

# Introduction to Linux for Scientists

A Crash Course to the Command Line

---

Dr Richard N.L. Terrett

2019-06-04

Computational Quantum Chemistry Group, Research School of Chemistry, Australian National University, Canberra, Australia

# Introduction

---

# Who am I?

## Position

Postdoctoral Fellow at RSC

## Role

Computational chemist, unofficial IT guy

## Linux Experience

Used headless Linux compute nodes for about 10 years.

Daily driver since about 2016.

Previously a Windows enthusiast.

Not an expert, but I do use Linux for everything.

# Background

- Thanks to Nic for inviting me, to Ananthan for coordinating the event, and to Erin for beta-testing and assistance.
- This workshop is designed to get you up to speed with connecting to a Linux server, navigating the Linux filesystem, performing simple filesystem operations, running and managing programs, and transferring data to and from the server.
- As we don't have a lot of time, it's a whirlwind tour.
- Often, simply knowing that something is possible is important information in its own right, to be pursued at your own pace.
- I will be available for a while afterwards for discussion.
- The **slide deck** will be available. Make sure to grab a **cheat sheet** too!

## Computer Literacy

Ability to use a computer to perform scientific research. Good familiarity with either Windows or MacOS. Prior Linux experience is great!

## Command Line Experience

Basic knowledge of command line interfaces and/or REPLs<sup>1</sup> (such as the R, Python, or Julia REPLs).

## Hardware/Software

Ability to connect to `nimbl.anu.edu.au` through `ssh` using a public/private RSA keypair.

---

<sup>1</sup>Read-Evaluate-Print Loop

# What is Linux?

- Free and Open Source Software (FOSS) operating system (OS).
- Linux Kernel + GNU CoreUtils.
- Collaborative effort spanning decades.

# Linux is Everywhere

- Linux is ubiquitous in scientific computation.
- Linux is the operating system of *all* of the top 500 supercomputers.
- Android is based on the Linux Kernel.
- Linux is very popular as a server OS.

Linux is very unpopular as a desktop OS. Why?

- Perceived or actual user-unfriendliness and lack of familiarity.
- Most desktop computers are sold with Windows or MacOS pre-installed.
- Lack of native Linux versions of key software (Microsoft Office, Adobe CC, many AAA video game titles).<sup>2</sup>

---

<sup>2</sup>However many alternatives of varying degrees of competence exist, and emulation layers are getting very good.



## Some remarks

- Linux is '*free as in freedom*' (*libre*) and '*free as in beer*' (*gratis*). You can modify it and redistribute it under the terms of the GNU Public License.
- Linux is very powerful, but also assumes you know what you are doing.
- Linux will not typically stop you from doing damage. Think before you act, and back up your work.

## Connecting to a server

---

You will be remotely logging into a 'headless' server (`nimbl.anu.edu.au`) to perform computations.

Headless systems are systems without a monitor or keyboard/mouse, or systems that are otherwise not intended to be used as desktop computers.

# Network terms

## Local host

The computer you are currently sitting at, wherever you are.

## Remote host

A remote computer that you can connect to through a network.

## Hostname

A human-readable name for a computer on a network  
(e.g. `nimbl.anu.edu.au`, `www.google.com`, `system.local`).

## IP address

A machine-readable address for a computer on a network  
(like `192.168.1.120` or `fe80::8c9d:eead:4614:4211`).

## Port

A numeric address that is used to split network traffic to/from a computer based on purpose.

ssh is a program and protocol for connecting to other computers across a network. The idea is that ssh allows you to access a remote command *shell* on another computer, to execute commands at the command line, as if you are sitting at it.

## Opening an ssh session

```
1 ssh [username]@[remotehost]
```

Where [username] is your username, and remotehost is the hostname (or IP address) of the computer you are connecting to. If you have not configured the use of an ssh key, you will typically be prompted for a password. After authenticating, you will be presented with a login session that allows you to run commands on the remote host.

## Closing an ssh session

To disconnect, we can use `exit` to close the shell, at which point we will be returned to the shell from which we originally ran `ssh`. `ssh` sessions can be nested.

```
1 [user@remote_server ~]$ exit
  logout
3 Connection to remote_server closed.
  [user@my_laptop ~]$
```

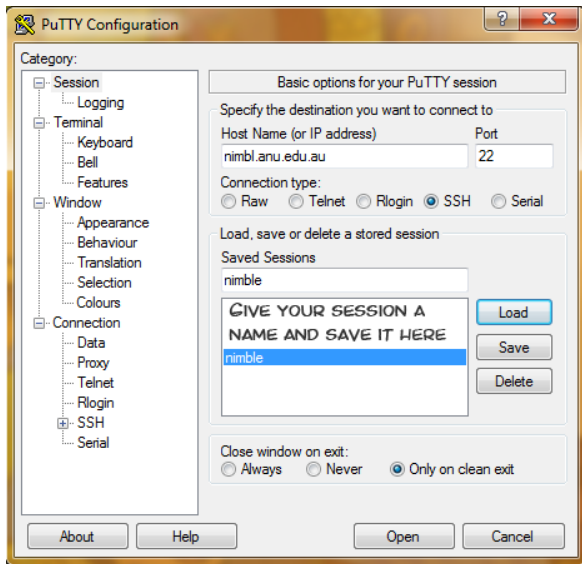
## X11 forwarding

- `ssh` supports something called '*X11 forwarding*', which allows you to run graphical programs as if you were seated at the remote host. This requires you to be running an X11 server.
- Linux comes with an X11 server. MacOS has the XQuartz server available at <https://www.xquartz.org>
- Windows has several X11 servers available, such as Xming, Cygwin/X, Xwin32, and VcXsrv. YMMV.
- X11 forwarding can be enabled by using `ssh -X` or by a setting in PuTTY.
- X11 forwarded user interfaces may be quite slow compared to the native experience, and certain 3D applications may not function properly.

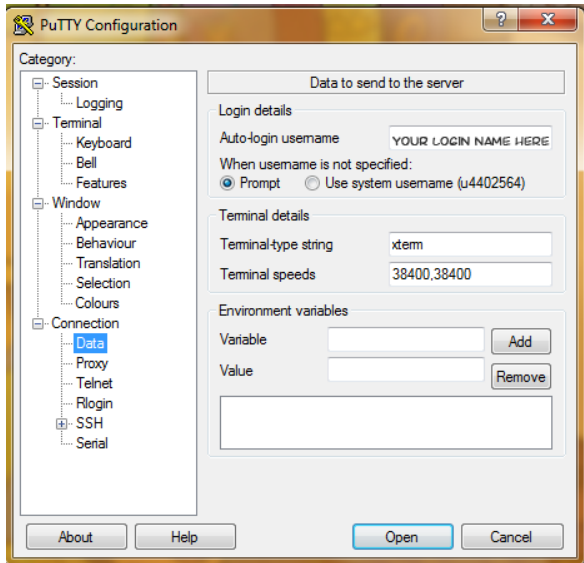


- If you have Xming/XQuartz, start it now.
- Xming is an X11 server for Windows. XQuartz is an X11 server for Mac.
- If you are on Windows, start up PuTTY. Mac/Linux users, open a terminal and use the `ssh -X` command previously mentioned.
- The following PuTTY configuration images have been provided by Dr Erin Walsh.

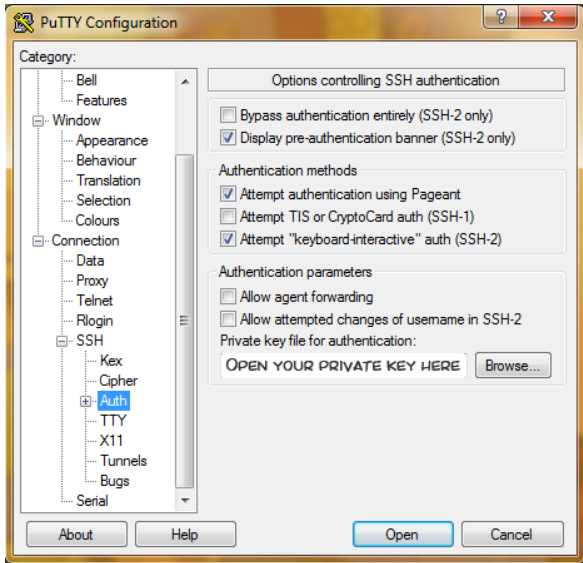
# PuTTY: Specifying a remote host



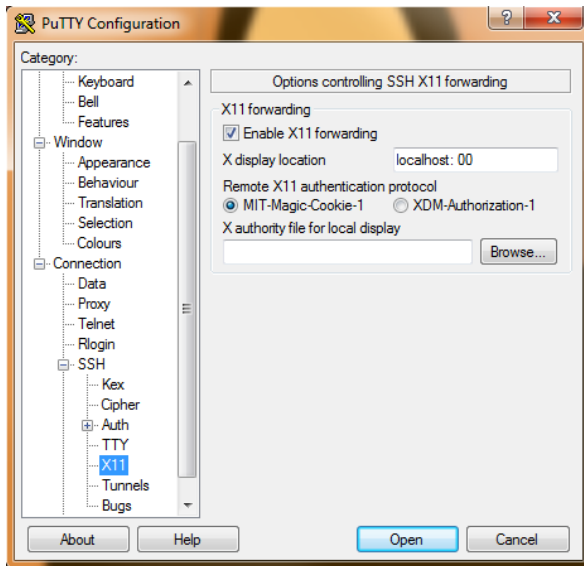
# PuTTY: Specifying a username



# PuTTY: Specifying a private key



# PuTTY: Enabling X11 forwarding



## X11 Forwarding Example

I'm going to start `firefox localhost:8787` on `nimbl`. It will hopefully appear on screen with the `rstudio-server` login screen. Hopefully you can do this too!

- Typing in a password every time you connect to a remote host with `ssh` can be tiresome.
- Low-quality passwords can represent a security risk.
- Asymmetric key cryptography, also known as public-key cryptography comes to the rescue!
- Asymmetric key cryptography is an enormous topic, we will only discuss the bare basics.

- Generate a private/public keypair with `ssh-keygen` or PuTTYgen.
- Your *public* key goes into the `~/.ssh/authorized_keys` file of the *remote host*.
- Your *private* key stays in the `~/.ssh/` directory of your *local host*, or in some other location specified in PuTTY.
- Don't mix these up.
- Your key can be encrypted with a password for extra security. This is technically a form of two-factor authentication.



- When used correctly, `ssh` keys can provide a highly convenient authentication method with much greater security than passwords.
- By default, `ssh-keygen` produces a keypair with 256 B of Shannon entropy.
- An 8-character *randomly generated* alphanumeric password (numbers and lower- and uppercase letters) contains less than 6 B of entropy.

## What happens if someone else has my *public key*?

- They can write messages to you that only you can read.
- They can securely authenticate your identity.
- These are good things!
- Your public key is public because *you are supposed to give it to other people*.

## What happens if someone else has my *private key*?

- They can impersonate you.
- They can log in to systems as you.
- This is generally considered to be bad.


## I lost control of my private key! What do I do?

- Don't panic.
- Notify your system administrator and other relevant personnel.
- Remove the corresponding public key from `authorized_keys` on any system that uses it, or ask a system administrator to do so. This is like changing the locks on your house.
- Don't reuse that keypair.
- Generate a new keypair.

# The Linux Command Line

---

# Introduction to the CLI

- The Linux **command line interface** (CLI) is how you will do most of your interaction with the server.
- Type in a line and hit  to submit a command.
- Linux is CASE SENSITIVE. Capitalisation matters in commands and filenames.

```
user@system:~$ echo "Hello World!"  
Hello world!  
user@system:~$
```

## Hopefully you're in...

You should see a prompt like:

1

```
[your_user_name@nimbl ~]$
```

## Hopefully you're in...

Run the following command verbatim to copy the demo files into your home directory:

```
1 [your_user_name@nimbl ~]$ cp -r /home/richard_terrett/demo ~/
```



The command line is provided by a program called a **shell**. Shells can provide more or fewer features, and typically behave fairly similarly, but might have slight differences. There are several shells available, including:

`bash`

The Bourne Again Shell. Default shell on many distributions.

`zsh`

Highly customisable shell with rich package ecosystem.<sup>3</sup> **My favourite.**

`fish`

The Friendly Interactive Shell. Many conveniences for newcomers.

---

<sup>3</sup>Check out 'oh-my-zsh'!

## Important terms

`alias`

A command that abbreviates another command or function. Defined with the `alias` program.

`.bashrc`

A file that stores settings for `bash` and is executed when `bash` is started.<sup>4</sup>

### Environment variable

A variable defined within the scope of the current shell session, and accessible to programs run by that shell.

### Argument/switch

An option that is supplied to a command. Arguments can dramatically change the behaviour of programs.

---

<sup>4</sup>Other shells and programs have their own `*rc` files (e.g. `.zshrc`)

## Getting help

There are a few ways of getting help within the shell:

`man [program]`

`man` opens the *manpage* of `[program]`. *manpages* are detailed manuals of how to operate a program, however they may be a bit verbose. Not every program has a *manpage*.

`-h/--help` **switch**

Supplying this switch to a program will give a short outline of the command line arguments of that program, if such documentation exists.

`apropos [search_string]`

Searches the manpage database for `[search_string]`. Useful for finding commands.

# \$PATH

\$PATH is an environment variable that stores a list of everywhere your shell looks for commands to execute. Here is my \$PATH on nimbl:

```
1 [richard_terrett@nimbl /]$ echo $PATH
/usr/lib64/qt-3.3/bin:/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/
usr/sbin:/sbin:/opt/dell/srvadmin/bin:/usr/local/bin/ts:/home/
richard_terrett/bin
3
```

# The Linux Filesystem

---

# Filesystem terms

## Filesystem

The hierarchical structure of files and directories (folders) available to a computer, and/or how it is internally represented.

## Directory

The term Linux uses for what in Windows would be called a 'Folder'.

## Symlink

A 'symbolic link', similar to what Windows would call a 'Shortcut'.

## Mount

Attach a subsidiary filesystem to another filesystem so that it can be accessed.

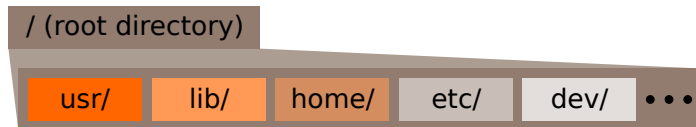
## Mountpoint

Where a mounted filesystem appears in the filesystem hierarchy.

## Directory structure tour: Root

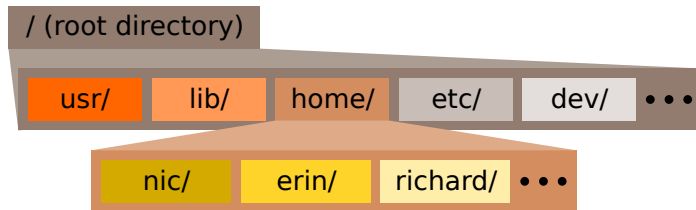
/ (root directory)

## Directory structure tour: Inside the root directory

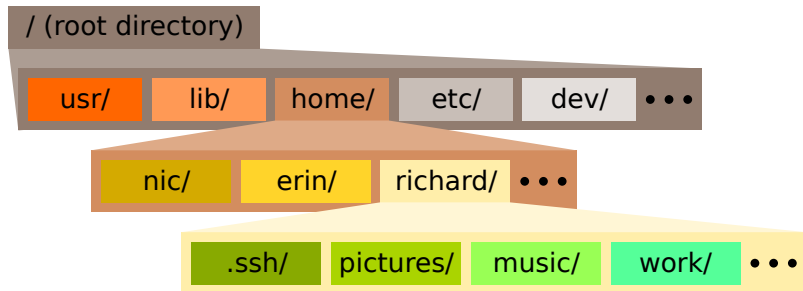




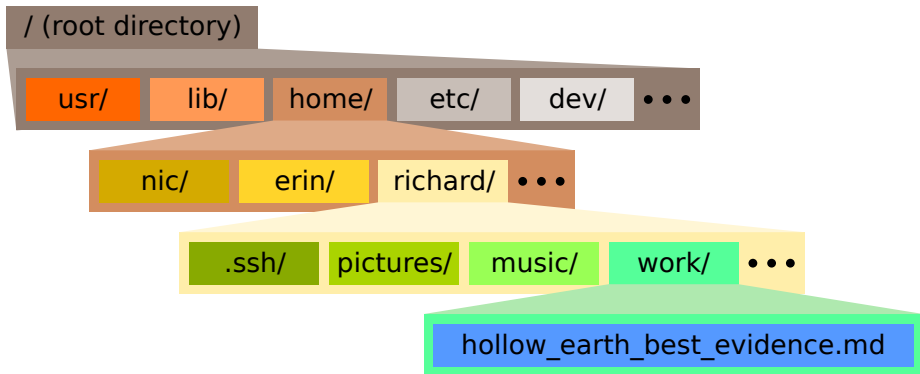
## Directory structure tour: Home directories



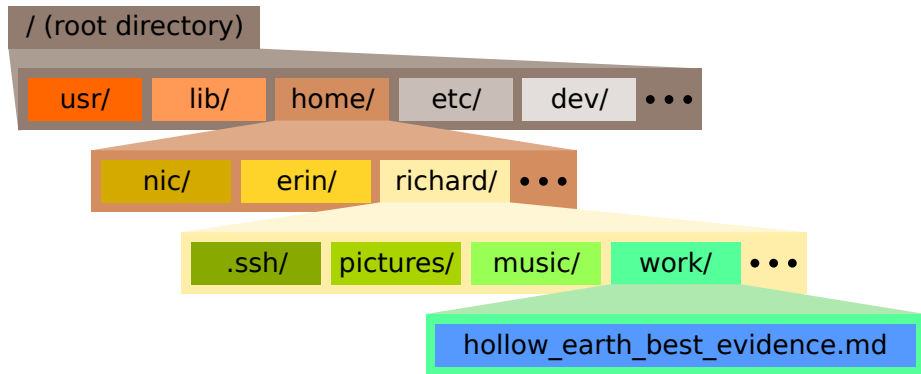
## Directory structure tour: Inside a home directory



## Directory structure tour: What's this?



## Directory structure tour: Full path



`/home/richard/work/hollow_earth_best_evidence.md`

# Special Directories

## Working Directory (.)

The directory that a user has navigated to at any point in time.

## Parent Directory (..)

The directory one hierarchical level above the working directory.

## Previous directory (-)

The last working directory you were in.

## Home Directory (~)

What it sounds like: your 'home' on the system. Everything below here is yours.

## Root directory (/)

The most fundamental level of the filesystem hierarchy.

## Moving around in the filesystem

`cd [path]`

Move to the specified path.

`cd ..`

Move up one directory.

`cd -`

Move to the previous directory you were in.

`cd ~`

Go home.

- Filenames in Linux are case sensitive.
- You can't have more than one file or directory with the same name in a directory.
- Suggestion: avoid special characters and spaces in filenames. Non-latin characters generally supported, but discouraged.
- Filenames with special characters or spaces can be wrapped in '"' or escaped with '\', but this can get awkward fast.

## Where are we in the filesystem?

`ls [directory]`

Lists the contents of the specified directory. If none is specified, defaults to the working directory (.). This will probably be the most frequent command you will use.

`tree [directory]`

Prints a hierarchical view of the directory structure beneath [directory]. Defaults to the working directory if none is specified.

`pwd`

Simply prints the path of the current working directory (~).






## Hidden files and directories

If a file or directory name begins with '.', it will be hidden. This means that it will not show up in tab completion or `ls` (unless you use the `-a` command line argument). This is used to keep your view of the filesystem tidy. It is not a security feature.

Useful things to know...

---

When you type a password in at the Linux command line, you typically will not get any visual feedback in the form of dots or asterisks. This is security best practice.

Your input is being accepted: press  to complete password entry, or  +  to delete the line (this works everywhere, by the way).

## Keyboard shortcuts

**ctrl** + **u** Erases the command line prompt.

**ctrl** + **w** Deletes everything before the caret.

**ctrl** + **p** Deletes everything after the caret.

**ctrl** + **r** Search command history.

**ctrl** + **c** Kill (keyboard interrupt) a running task and return to prompt.

**ctrl** + **z** Suspend a running task and return to prompt.

**ctrl** + **l** Visually clear the terminal.

**↑**, **↓** Move backwards and forwards in command history.

**ctrl** + **←**, **ctrl** + **→** Move backwards and forwards in line in delimited increments.

**→** Complete a command or path where possible.

## Common command line switches / arguments

- h/--help Provides a description and list of command line arguments for the program. Some programs use -h to produce more 'human readable' output, in which case use --help.
- a Prints all output, including output that would normally be hidden.
- r Applies command recursively over directory hierarchy.
- f Forces a command to be executed where it would otherwise stop.
- v Turns on verbose output, useful for debugging. Adding more vs can sometimes produce even more verbose output!

The `*` character is expanded in the shell as a wildcard, representing zero or more of any character.

It is equivalent to `.*` in a regular expression.

# The working directory is not in \$PATH by default

This will not work:

```
user@system:~$ program.exe
2 program.exe: command not found
```

This will:

```
user@system:~$ ./program.exe
2 Hello World!
```

Speculate on why this might be...

- Most users are limited users.
- Do not have complete access privileges to the host. Usually only have write access to `/home/[username]/`.
- Cannot modify system files.
- Limits the scope of damage a user or process operating on behalf of a user can do.



# Superusers

- *'root'* access. Complete access to whole filesystem, appropriate to system administrators (*'sysadmins'*).
- *'root'* privileges are potentially hazardous: with great power comes great responsibility.
- Most system administrators will only elevate their privileges on a command-by-command basis using `sudo` (*'superuser do'*).
- Being made a superuser represents absolute trust and responsibility (within the context of the system), and therefore should never be given or taken lightly. A superuser can wreck a server. Don't take it personally if you don't have superuser access.

# Filesystem Operations

---

`touch [file_path]` will create an empty file, if it does not exist. If it does exist, the timestamp is updated to the present.

`mv [source_path] [destination_path]` will move a file from one location to another. If files with the same name already exist, they will be overwritten.

`cp [source_path] [destination_path]` will copy a file from one location to another, creating two instances. The `-r` switch will make the command *recursive*, so that it operates on entire directory trees. If files with the same name already exist, they will be overwritten.

## `rm` (remove)

`rm [file_path]` will delete a file by removing it from the file allocation table. Supplying the `-r` switch will make it recursively delete directory trees. `rmdir` can also be used to remove a directory, but only if it is empty.

## Some words on filesystem operations.

`rm`, `mv`, and `cp` are potentially dangerous commands, especially when used with the `-r` switch.<sup>5</sup> Files that are overwritten or deleted at the command line are unlikely to be recoverable by normal means, and no *'Recycle Bin'* equivalent is typically in place. However, 'removed files' are still on-disk until they are overwritten by other data. Keep this in mind when decommissioning storage media that contains sensitive data, as that data may be recoverable using data-carving tools. If in doubt, physically destroy the media.

---

<sup>5</sup>`mv` doesn't have an `-r` switch as it always operates recursively.

Who and where are we?

---



Some commands will be useful to you in negotiating a multi-user, networked environment.

`whoami`

Prints your username.

`hostname`

Get the name of the computer you are connected to.

`ps`

Lists processes created by the current user.

`top/htop`

Starts a live task manager.

w

Summarises currently logged in users, what they are running, and the resources they are using.

host [ip-addr]

Gets the hostname of the IP address ip-addr from the DNS.

users

Prints all of the logged in users.

cat /etc/passwd

This file contains the usernames of all real and virtual users of a system.

ifconfig

Prints information about the network interfaces on nimbl.

# Security and Privileges

---

# The Linux Permissions Model

Every file has three sets of three permissions associated with it. These are the read (**r**), write (**w**), and execute (**x**) permissions for the user who owns the file (**u**), the group the file belongs to (**g**), and other users (**o**).

- Every file has one owner. The owner of a file can access it with **u** privileges.
- Every user is a member of at least one group. Users in a group can access a file in that group with **g** privileges.
- Everyone can access files at the **o** privilege level (although these might very well be empty.)

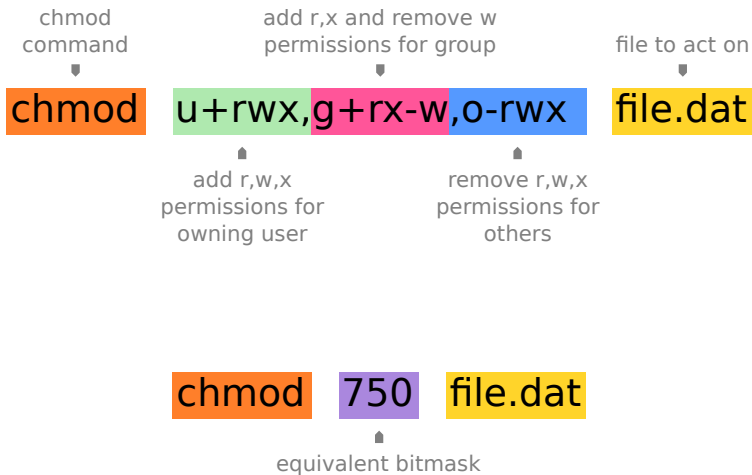
- Superusers can read and write to the filesystem in an unrestricted manner. This is intentional.
- Even if a user does not have read privileges to a file, they can still obtain some information about the file (existence, name, size, timestamp) if they have permission to enumerate the directory it is in.
- You can absolutely lock yourself out of your own file by changing its owner or group. :)

We can change the permissions on a file or folder with `chmod` ('change mode').

`chmod` accepts either a *permission string* or a *bitmask*.

```
[user@system ~/demo/permission] ls -l
2 total 0
   -rwxr-xr-x 1 user user 0 May 14 18:31 program.py
4  -rw-r--r-- 1 user user 0 May 14 18:31 public.mp3
   -rw----- 1 user user 0 May 14 18:31 secret.key
6  -rw-r----- 1 user user 0 May 14 18:31 sensitive.png
```

## chmod example





Value	Read	Write	Execute
0			
1			Yes
2		Yes	
3		Yes	Yes
4	Yes		
5	Yes		Yes
6	Yes	Yes	
7	Yes	Yes	Yes

## How does `chmod` affect directories?

`chmod`'s relationship with directories can be confusing:

- r** Enumerate directory contents.
- w** Create files in directory.
- x** Traverse directory (*i.e.* navigate into it).

Be aware that using the `-R` (recursive) switch with `chmod` will apply changes to both directories and files!

`chown [user] [path]` changes the ownership of a file or directory. You may need elevated privileges to change ownership.

`chgrp [group] [path]` changes the group of a file or directory. Type `groups [username]` to see which groups a user is a member of.

## Reading files

---

`cat` is short for '*catenate*'. When invoked with one or more filenames as their arguments, they will be printed sequentially to `stdout`, which is by default the terminal. This can be used to quickly check the contents of a file, or with output redirection can be used to concatenate several files into one.

tac is like cat but prints lines in reverse order. May not be installed.

echo just prints its arguments to stdout. Output redirection can be used to easily write simple text files. If the argument is a shell variable name (e.g. \$PATH), the value of that variable is printed.

Inside shell scripts, echo can be used as a *'print'* statement.

## head and tail

head and tail are like cat, but are restricted to the beginning and end of a file, respectively. The `-n [lines]` argument controls the number of lines that are printed, e.g. `tail -n 10` will print the last 10 lines of a file.



grep searches a text file for substrings that match a supplied '*regular expression*' (also known as a '*regex*'). In the simplest case, grep can just be used to search for string literals in a file. The `-B [lines]` and `-A [lines]` arguments can be used to print a number of lines before and/or after a match to provide context.

## A word on regular expressions

A regular expression is a type of grammar (in the computational linguistics sense) that can be used to search for strings belonging to a (potentially infinitely large) set satisfying certain constraints. This may sound complex, but regular expressions are genuinely life-changing to learn. Think the ability to search a file for anything that *looks like* a phone number, or an email address, or a DOI, or a HTML element, or a variant of the name 'Elizabeth', and then pull that data out automagically, all in a one-line expression. Very cool.

## find

`find` searches for patterns in filenames. `find` has a rather complex syntax which will not be discussed here, however piping `find` without arguments into `grep` is a simple way to search a directory hierarchy for substrings in a filename.

```
find | grep '[hH]ello'
```

2

`less` is an enhanced version of an older program called `more`.<sup>6</sup> `less` provides a paginated interface for reading text files. It is very efficient, and can read very large files with no problems.

---



<sup>6</sup>This is par for the course Linux 'humour'.

## Writing files

---

There are a large number of terminal editors available for Linux. Some are focused on being easy to use, others are almost operating systems in their own right.

`nano` is a terminal text editor that is often the default used by Linux distributions, on account of its user-friendliness and simplicity. It is an enhanced version of an older program called `pico`.<sup>7</sup>

`nano` lists a bunch of keyboard shortcuts at the bottom of the window. In this instance,  stands for .

---

<sup>7</sup>Are you starting to see a pattern here?

vim is a powerful editor that uses a *modal* control paradigm. In this paradigm the user switches between *insert* and *command* (or *normal*) modes to edit text and issue rich commands, respectively.<sup>8</sup>

vim is incredibly powerful (check out videos of vim experts at work), but has a very steep learning curve. vim is also very customisable and has a large plugin ecosystem. This slide deck was written using vim.

To exit from vim, hit `Esc` and then type `:q`.

---

<sup>8</sup>There are some other, additional modes.



emacs is the other power-editor. A holy war between vim and emacs has been raging for several decades now.<sup>9</sup> emacs has a different interaction philosophy from vim, using key combinations rather than modal control. It can be invoked at the terminal by `emacs -nw` and quit by `ctrl+x`, `ctrl+c`.

---

<sup>9</sup>No deaths, as far as I know.

## Editing files on the fly: sed and rename

---

sed is short for '*stream editor*'. Despite its unassuming nature, entire books have been written on this program. It accepts an input stream (typically a file), carries out some operation on it, and outputs the modified stream. Shell variables can be used in sed expressions.

A very useful sed command performs a find and replace operation on a file, without the need to open it in an editor:

```
1  user@system:~$ cat input.txt
   Hello World
3  user@system:~$ sed -i 's/World/Earth/g' input.txt
   user@system:~$ cat input.txt
5  Hello Earth
```

rename allows us to perform find and replace on file names instead of file contents. This allows the easy batch renaming of large numbers of files. Some systems have a different version of rename which uses a sed-like perl expression instead.

```
1 user@system:~$ touch hello-world.txt
2 user@system:~$ rename word earth hello-world.txt
3 user@system:~$ ls
4 hello-earth.txt
5
```

## The ins and outs of running programs

---

## Backgrounding a program

We may wish to start a program without it locking up the terminal to additional input. We can send a program to the background by adding an ampersand (&) to the command.

```
user@system:~$ ./long-running-script.py &
```

```
[1] 18550
```

```
user@system:~$
```

```
[1]+  Done
```

```
long-running-script.py
```

```
user@system:~$
```

## Suspending a program

- If we are running a program interactively, we can suspend it by hitting `ctrl` + `z`.
- We can restore the most recently suspended program to the foreground by running `fg`.
- We can restore the most recently suspended program to the background by running `bg`.
- We can also refer to jobs by running `fg` or `bg %[job_id]`. The `[job_id]` is reported when the job is suspended, or can be looked up by running `jobs`. If unambiguous, you can also foreground a task by the program name.

# Backgrounding example

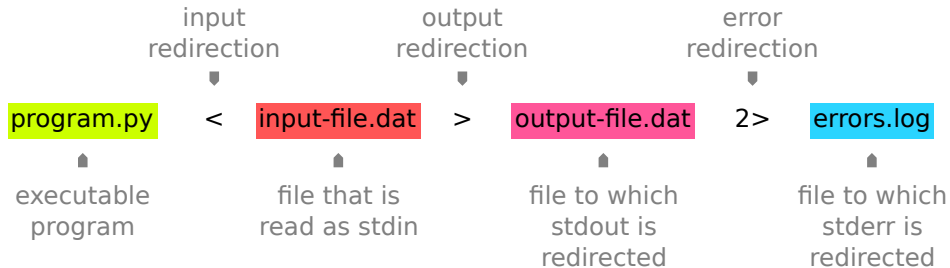
```
1  user@system:~$ vim &
   [1] 14581
3  user@system:~$ top &
   [2] 14584
5
   [1]+  Stopped                  vim
7  user@system:~$ jobs
   [1]-  Stopped                  vim
9  [2]+  Stopped                  top
   user@system:~$ fg %1
11 vim
```



## IO redirection

---

Often, we want to redirect input from a file into a program, and/or redirect output and error messages into files. Otherwise, these streams will be read from and written to the terminal.



## Redirection examples

Redirect output and error streams to files:

```
./program.exe < [in] > [out] 2> [err]
```

## Redirection examples

Append output stream to file and discard error messages:

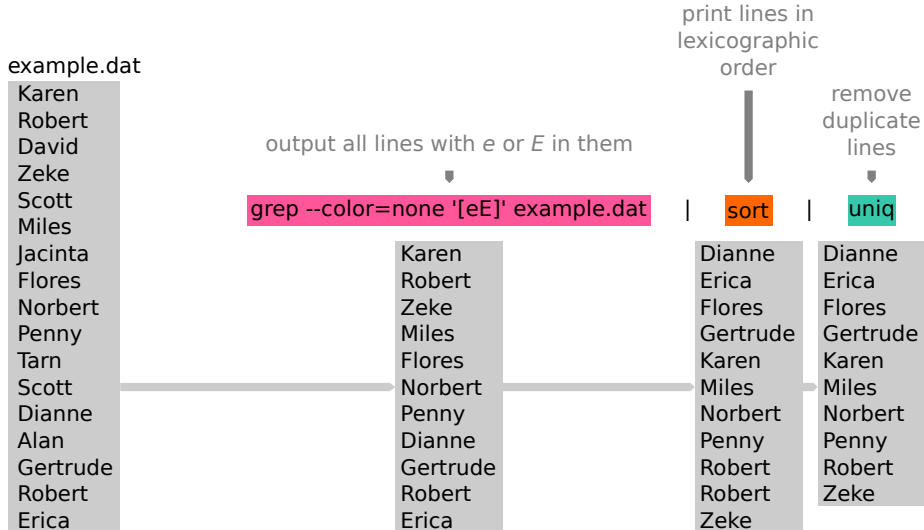
```
./program.exe < [in] >> [out] 2> /dev/null
```

## Redirection examples

Append output and error streams to same file:

```
./program.exe < [in] >> [combined] 2>&1
```

# Pipes example



## Being a good citizen of a multi-user environment

- Multiple users can work on a server at the same time. This requires some cooperativity on the part of users as resources are finite.
- Often, resource management systems (such as PBS, Slurm, or Torque) are used to fairly distribute computational resources to users.
- However, more *ad hoc* systems rely upon dialogue and consideration.



Niceness refers to the priority a process has in the kernel scheduler. This can range from low ( $-20$ ) to high ( $19$ ), with high niceness being given lower priority. This is useful even in single-user environments to ensure that critical processes do not have resources taken by noncritical processes.

- We can run a process with a desired niceness level by running it with `nice -n [niceness] [command]`.
- We can change the niceness of a running process with `renice -n [niceness] -p [PID]`.
- Niceness is shown in the NI column in `top` and `htop`.

Sometimes, a job may run longer than we intended. This is bad if it locks up resources that other users need, stalls a queue, or if the process is just spinning its wheels forever due to some non-terminating error condition or infinite loop. We can use `timeout` `[timelimit]` `[command]` to kill a process after some time. `[timelimit]` is a number followed by a unit suffix (s/m/h/d).

# Managing Long-Running Tasks

---

## Watching Job Output

Sometimes we might want to watch the output to a logfile in real-time. Running `tail` with the `-f` switch (`tail -f [logfile.log]`) will provide a live display of the output!

## But I Want To Go Home!

Sometimes we may have a task that will take a very long time to run.

We could just leave the `ssh` session open.

However, if we lose our connection, log out, or power down our computer, the task will end because the parent shell has exited.

`disown [job_id]` detaches the specified process from the shell so that it survives shell exit. If no `[job_id]` is specified, the last task you started will be disowned.

Obviously, if important output is being printed to the terminal, you should make sure that is is redirected into a file.

## Task Spooler (ts)

Whilst not installed by default, `ts` is a super useful lightweight queue system.

`ts [command]`

Adds `[command]` to the end of the queue.

`ts`

Displays the status of the queue, including queued, running, and finished jobs and the temporary directories the jobs are using.

`ts -r [task-id]`

Removes a queued job with `[task-id]`.

`ts -k [task-id]`

Kills a running job with `[task-id]`.

`ts -C [task-id]`

Clears running jobs from the queue.



ts has many other options: run `ts -h` for the list of arguments.

ts obviates any concerns about detaching jobs from the shell: you can disconnect and the jobs will keep running automatically.

If we want to do stream redirection with ts, we need to make the context of the redirection clear. An easy way to do this is to run our command as an inline bash script, *e.g.*:

```
ts bash -c "program.exe < input > output 2> logfile"
```

## Killing processes

---

## Killing jobs with `kill`

- `kill [PID]` will ask the process with that PID nicely to terminate (using the SIGTERM signal (15)).
- `kill -9 [PID]` will tell the process with that PID to terminate, not so nicely (using the SIGKILL signal (9)).

## Killing jobs with `pkill`

- `pkill [name]` will kill all running processes with that program name.
- `pkill -n [name]` will kill the most recent process with that program name.

Check out `pkill -h` for the full list of options.

## Killing jobs with `htop`

You can kill a process in `htop` by selecting it and hitting `k`, `↵`.

## Moving data across the network

---

## scp (secure copy)

scp is a program that copies files from a local host to a remote host or *vice versa*. Under the hood, it uses ssh to achieve this. It behaves in much the same way as cp except it accepts a `user@hostname:` qualification to the file path.

**Pull:**                remote host                remote path                local path  
scp -r `backdoor@cia.gov:~/KFC_spices.docx` `~/Downloads/read-later`

**Push:**                local path                remote host    remote path  
scp `"Suzanne Vega - Tom's Diner.mp3"` `user@laptop:~/Music/`

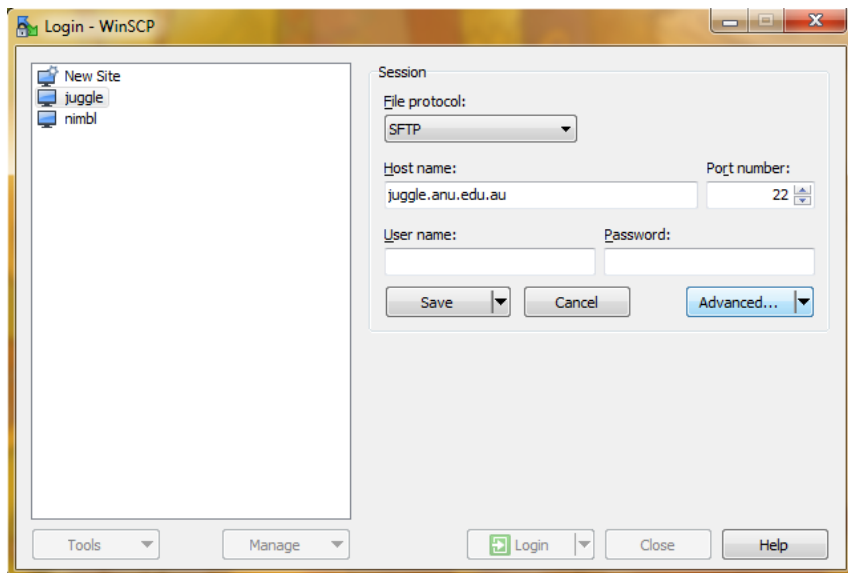
rsync is a program like scp, but is focused on moving large amounts of data efficiently, e.g. for archival purposes. It is smart enough not copy data that already exists at the destination path, and can use compression to considerably speed up the transfer of certain types of data. rsync can be used as a drop in replacement for scp in many circumstances.



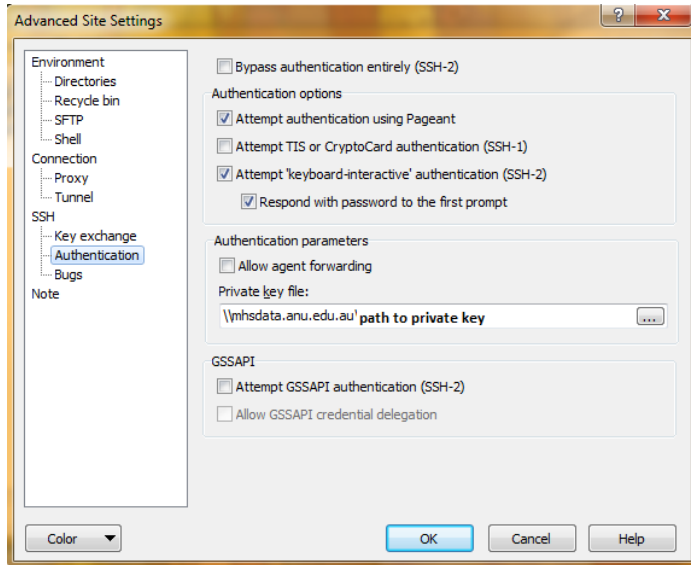
`sshfs` allows a user to mount a remote filesystem through `ssh` as if it were locally attached. Super useful, but does not come standard on most distros. Will not be covered here, but definitely worth investigating.

WinSCP is a GUI program for Windows that allows you to read and write files over SCP.  
Erin has helpfully provided a number of screenshots.

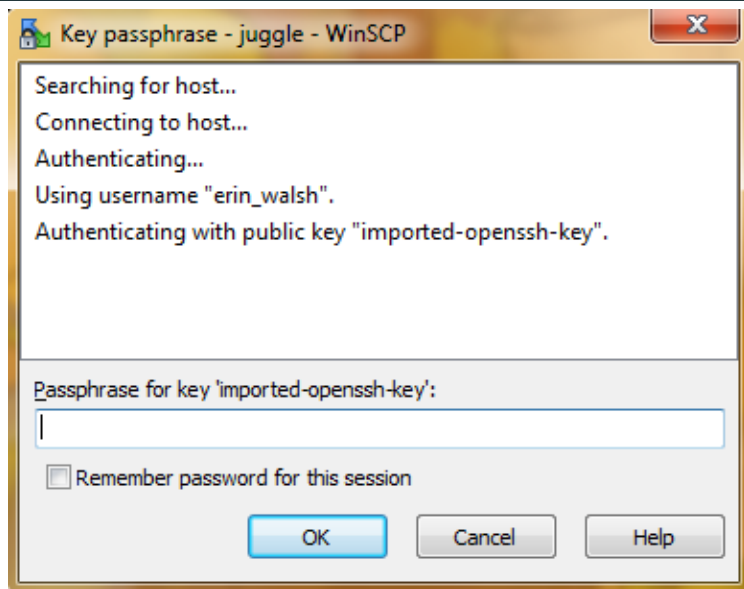
## WinSCP: Specifying a remote host



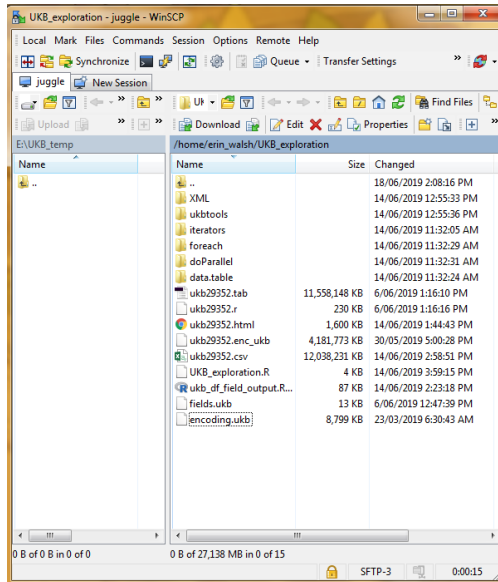
## WinSCP: Specifying a private key



## WinSCP: Unlocking your private key



# WinSCP: File explorer



*'Freedom is worth the inconvenience'*

— Richard Stallman