

File Commands	System Info
ls - directory listing	date - show the current date and time
ls -al - formatted listing with hidden files	cal - show this month's calendar
cd dir - change directory to <i>dir</i>	uptime - show current uptime
cd - change to home	w - display who is online
pwd - show current directory	whoami - who you are logged in as
mkdir dir - create a directory <i>dir</i>	finger user - display information about <i>user</i>
rm file - delete <i>file</i>	uname -a - show kernel information
rm -r dir - delete directory <i>dir</i>	cat /proc/cpuinfo - cpu information
rm -f file - force remove <i>file</i>	cat /proc/meminfo - memory information
rm -rf dir - force remove directory <i>dir</i> *	man command - show the manual for <i>command</i>
cp file1 file2 - copy <i>file1</i> to <i>file2</i>	df - show disk usage
cp -r dir1 dir2 - copy <i>dir1</i> to <i>dir2</i> ; create <i>dir2</i> if it doesn't exist	du - show directory space usage
mv file1 file2 - rename or move <i>file1</i> to <i>file2</i> if <i>file2</i> is an existing directory, moves <i>file1</i> into directory <i>file2</i>	free - show memory and swap usage
ln -s file link - create symbolic link <i>link</i> to <i>file</i>	whereis app - show possible locations of <i>app</i>
touch file - create or update <i>file</i>	which app - show which <i>app</i> will be run by default
cat > file - places standard input into <i>file</i>	Compression
more file - output the contents of <i>file</i>	tar cf file.tar files - create a tar named <i>file.tar</i> containing <i>files</i>
head file - output the first 10 lines of <i>file</i>	tar xf file.tar - extract the files from <i>file.tar</i>
tail file - output the last 10 lines of <i>file</i>	tar czf file.tar.gz files - create a tar with Gzip compression
tail -f file - output the contents of <i>file</i> as it grows, starting with the last 10 lines	tar xzf file.tar.gz - extract a tar using Gzip
Process Management	tar cjf file.tar.bz2 - create a tar with Bzip2 compression
ps - display your currently active processes	tar xjf file.tar.bz2 - extract a tar using Bzip2
top - display all running processes	gzip file - compresses <i>file</i> and renames it to <i>file.gz</i>
kill pid - kill process id <i>pid</i>	gzip -d file.gz - decompresses <i>file.gz</i> back to <i>file</i>
killall proc - kill all processes named <i>proc</i> *	Network
bg - lists stopped or background jobs; resume a stopped job in the background	ping host - ping <i>host</i> and output results
fg - brings the most recent job to foreground	whois domain - get whois information for <i>domain</i>
fg n - brings job <i>n</i> to the foreground	dig domain - get DNS information for <i>domain</i>
File Permissions	dig -x host - reverse lookup <i>host</i>
chmod octal file - change the permissions of <i>file</i> to <i>octal</i> , which can be found separately for user, group, and world by adding:	wget file - download <i>file</i>
<ul style="list-style-type: none"> ● 4 - read (r) ● 2 - write (w) ● 1 - execute (x) 	wget -c file - continue a stopped download
Examples:	Installation
chmod 777 - read, write, execute for all	Install from source: ./configure
chmod 755 - rwx for owner, rx for group and world	make
For more options, see man chmod .	make install
SSH	dpkg -i pkg.deb - install a package (Debian) rpm -Uvh pkg.rpm - install a package (RPM)
ssh user@host - connect to <i>host</i> as <i>user</i>	Shortcuts
ssh -p port user@host - connect to <i>host</i> on port <i>port</i> as <i>user</i>	Ctrl+C - halts the current command
ssh-copy-id user@host - add your key to <i>host</i> for <i>user</i> to enable a keyed or passwordless login	Ctrl+Z - stops the current command, resume with fg in the foreground or bg in the background
Searching	Ctrl+D - log out of current session, similar to exit
grep pattern files - search for <i>pattern</i> in <i>files</i>	Ctrl+W - erases one word in the current line
grep -r pattern dir - search recursively for <i>pattern</i> in <i>dir</i>	Ctrl+U - erases the whole line
command grep pattern - search for <i>pattern</i> in the output of <i>command</i>	Ctrl+R - type to bring up a recent command
locate file - find all instances of <i>file</i>	!! - repeats the last command
	exit - log out of current session
	* use with extreme caution.





Why Linux?

Dr. Gareth Roy (x6439)
gareth.roy@glasgow.ac.uk

- A brief history lesson
- Why Linux?
- The parts of an Operating System
- The Command Line

- A brief history lesson
- Why Linux?
- The parts of an Operating System
- The Command Line

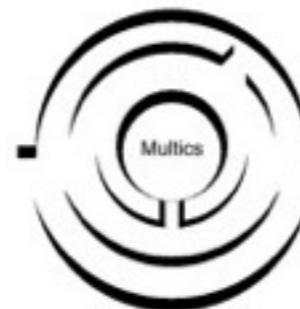
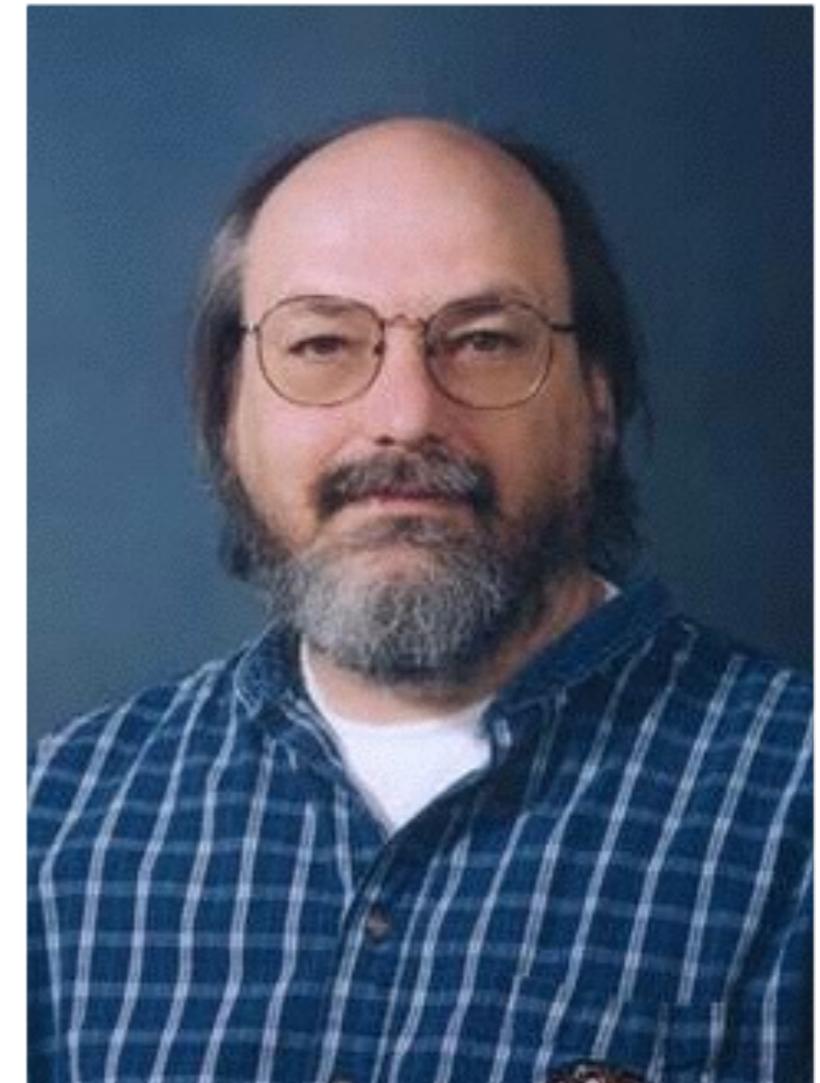


IBM

DATA CENTRE

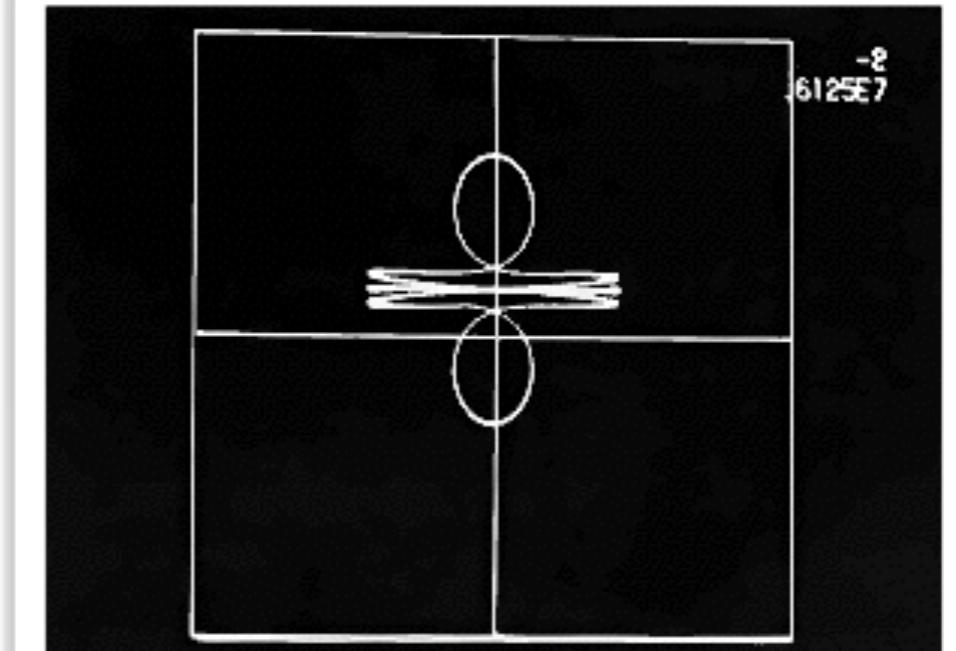
Kenneth Lane Thomson (1943 -)

- Received his Bachelor's and Master's degrees in E&EE from the University of California at Berkley
- In 1966 was hired by Bell Labs (owned by AT&T) to work on the MULTICS project.
- MULTICS (**M**ultiplexed **I**nformation and **C**omputing **S**ervice) was a mainframe time sharing operating system.
- Worked on developing the OS until Bell Labs withdrew from the consortium in 1969.



Space Travel

- While working on MULTICS, Ken created a game called Space Travel.
- It allowed the player to travel around a 2D solar system.
- Originally intend for MULTICS, Ken re-wrote it to run on an obsolete PDP-7 located in the lab.
- Ken wanted a better system to run his game on so in the Summer of 1969 he took one month to write:
 - Kernel
 - Shell
 - Editor
 - Assembler



Birth of UNIX

- Bell Labs left the MULTICS consortium in 1969.
- Bell Labs purchased new PDP-11, and development continued on what would become UNIX.
- In 1970 Ken created the ‘B’ programming language which was the precursor to C.
- Joined by Dennis Ritchie in 1972, Ken re-wrote the UNIX kernel in C.
- UNIX was presented to the world in 1973.



Ken Thomson and Dennis Ritchie working on the PDP-11 used to develop UNIX

What is UNIX

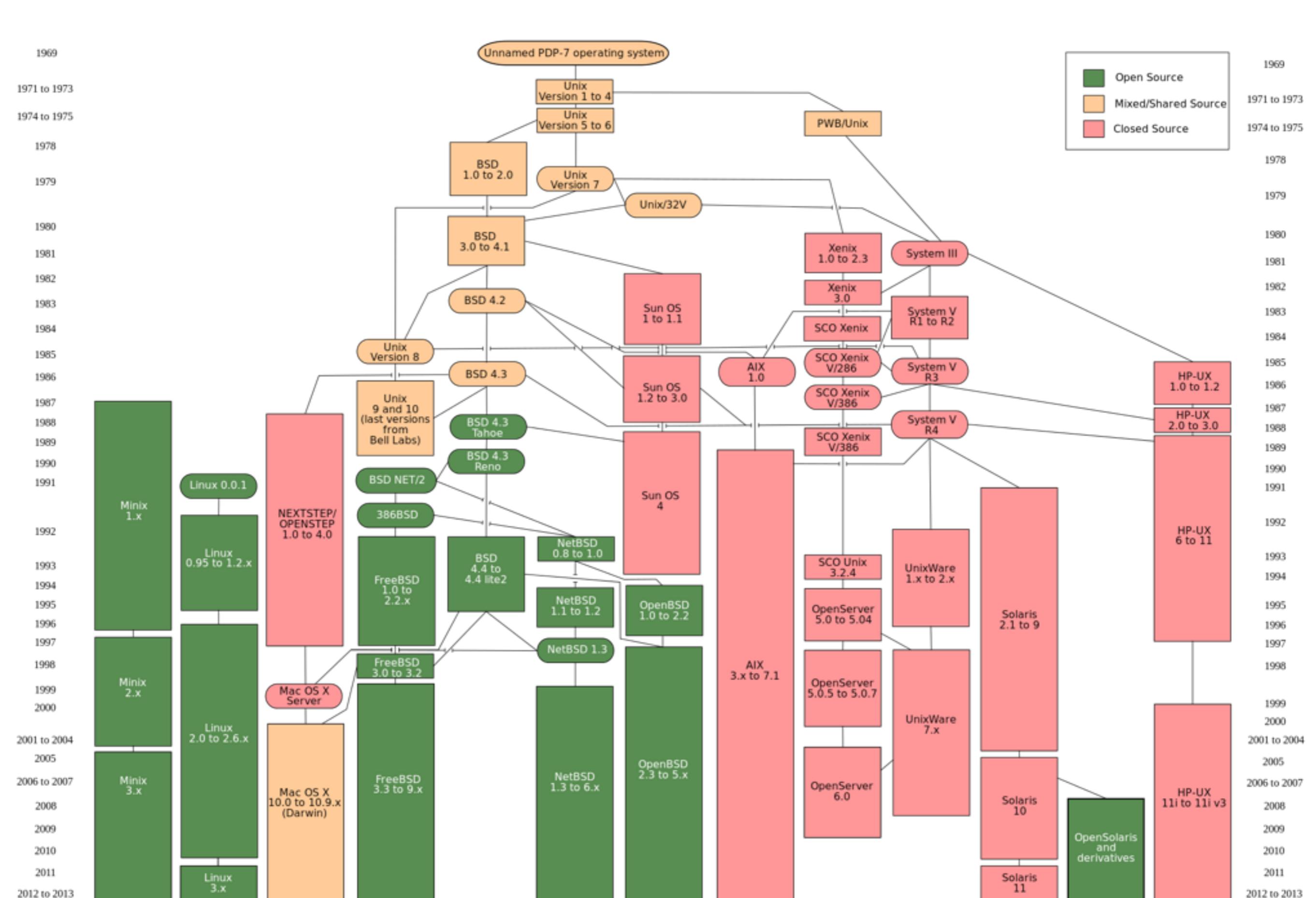
- UNIX is a Multi-user, Multi-tasking operating system (at present a family of Operating Systems).
- Any UNIX system has some key features:
 - the use of plain text for storing data
 - a hierarchical file system
 - devices and processes represented as files
 - the use of a large number of software tools, small programs that can be composed as opposed to using a single monolithic program
- In 2015 to be called a UNIX requires certification by the OpenGroup

A screenshot of a Mac OS X terminal window titled "2. bash". The window shows a command-line session where the user has navigated to their home directory and listed its contents. The output is as follows:

```
Last login: Tue Jan 13 13:14:55 on ttys001
Folkvangr:- gareth$ pwd
/Users/gareth
Folkvangr:- gareth$ ls
Applications Documents Installers Music Public
Backups Downloads Library Personal University
Desktop Dropbox Movies Pictures
Folkvangr:- gareth$
```

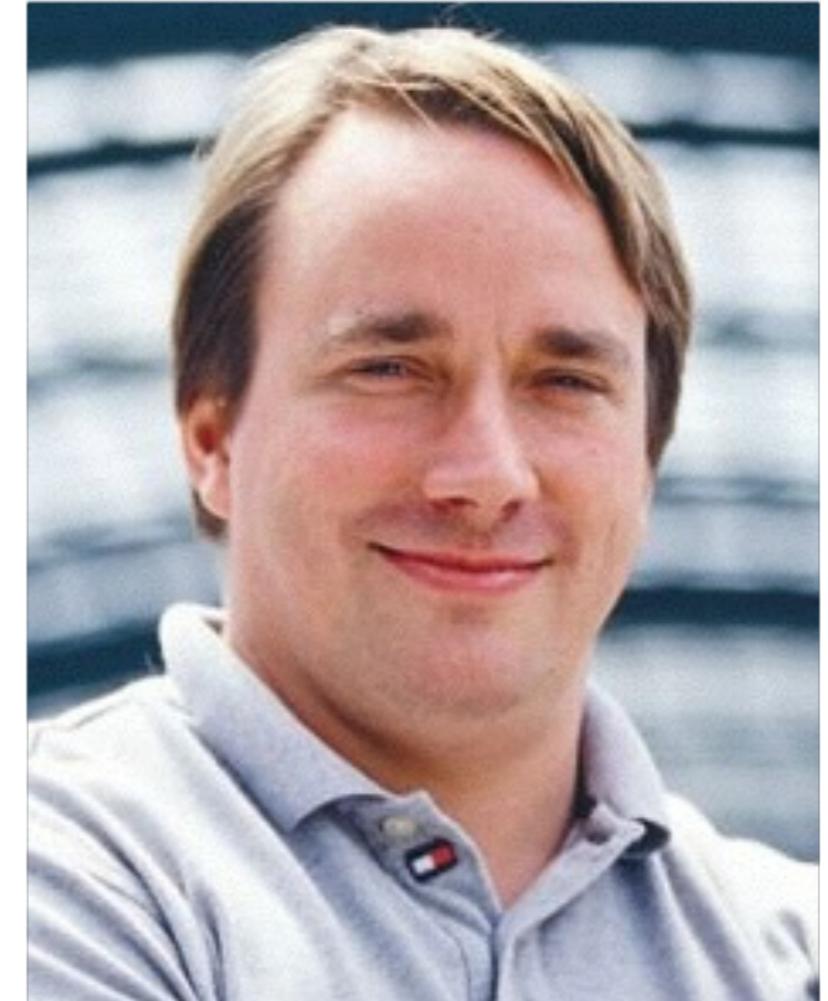
The window has a dark background with light-colored text and standard Mac OS X window controls.

<http://www.unix.org/version4/>



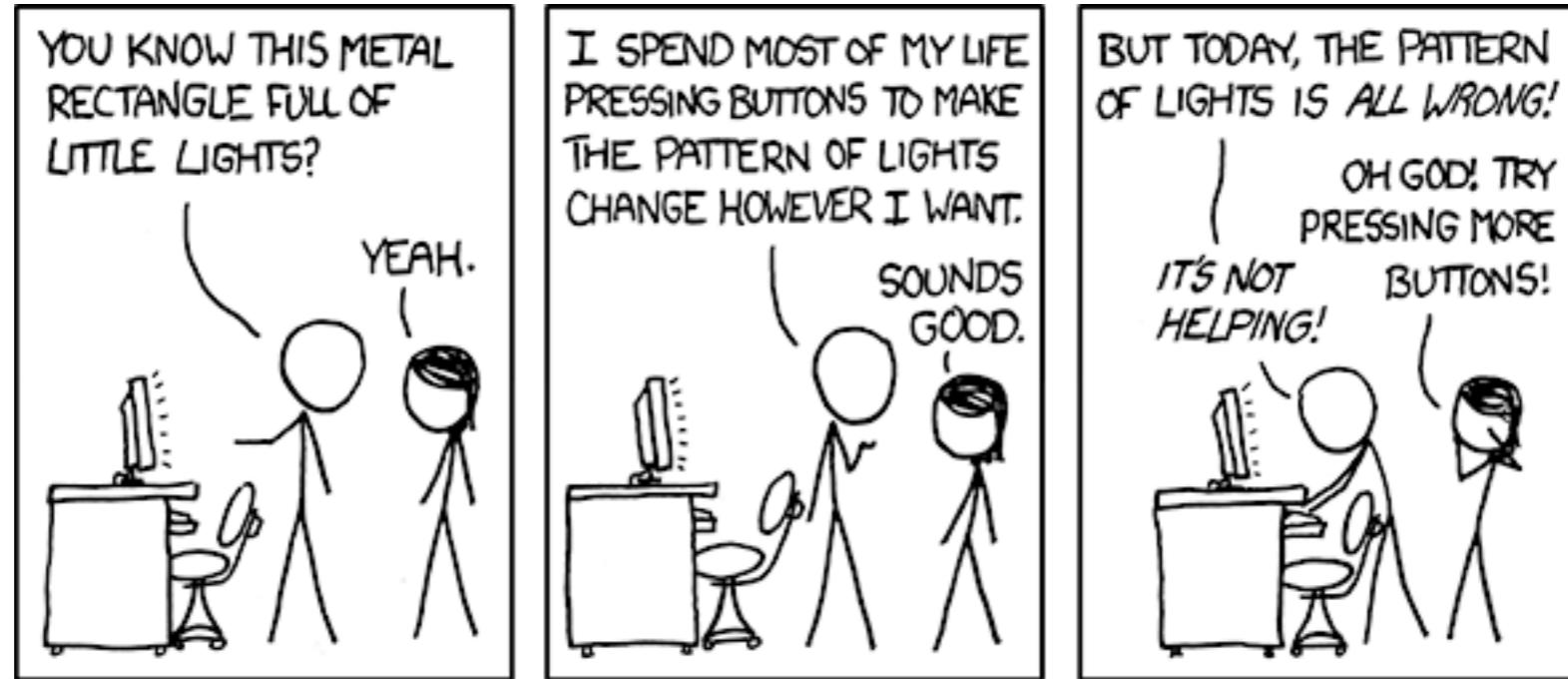
Linus Benedict Torvalds (1969 -)

- After becoming frustrated with the License associated with Minix, Linus began work on his own operating system.
- In 1991, Linux v.0.01 was released on comp.os.minix
- Originally intended to be called Freax, renamed by a co-worker (Ari Lemmke) when it was placed on the server.
- Linux was only the OS Kernel itself and so a complete Linux OS heavily leveraged the GNU utilities and tools.
- This later led to Linux being released under a GPL license becoming open source.



"I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones."

- A brief history lesson
- Why Linux?
- The parts of an Operating System
- The Command Line



xkcd.com

Why Linux?

- Linux is a free, with no cost required to obtain a complete fully featured OS.
- Linux source code is freely available and can be modified to fit user needs.
- Linux is a multi-user, multi-tasking operating system based on the tested principles of UNIX.
- Linux is ubiquitous in scientific computing with 485 of the top 500 supercomputers in the world running Linux.
- Scientific code bases for Cosmology, Particle Physics, Nuclear, Atmospheric, Fluid Dynamics all run on Linux platforms.

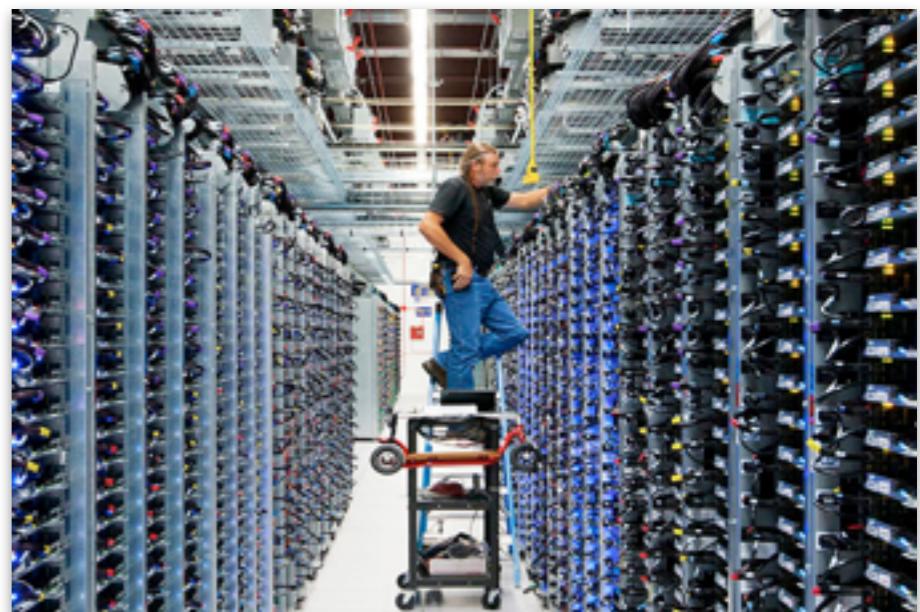
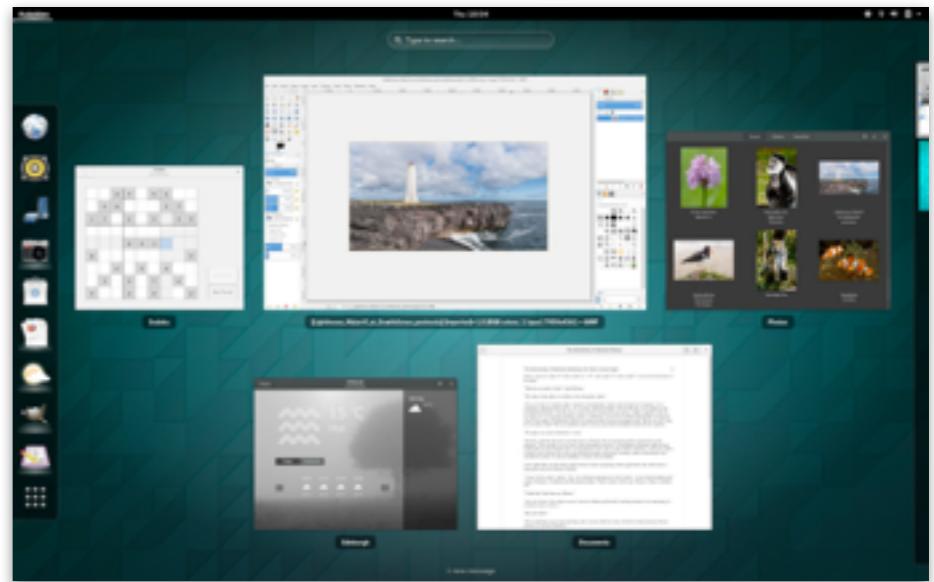
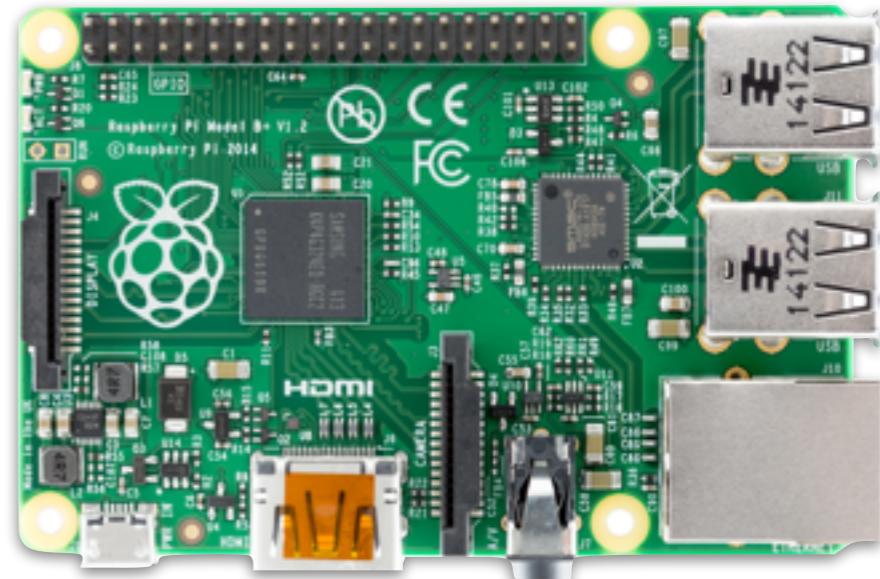


Tianhe-2 (MilkyWay-2)

- 3,120,000 CPU Cores
- 1,024,000 GB of RAM
- 33,862.7 TFlop/s
- 17,808 kW

Why Linux?

- Due to it's open nature Linux has been ported to many platforms and architectures.
- It scales from embedded devices such as the Raspberry PI to powering companies such as Google and Facebook.
- Linux is on your phone (Android), it's on your router (dd-wrt), it's on your TV (lots), in your car (BMW), it runs the city of Munich, and your data is already stored there if you use Facebook/Google/DropBox etc...
- It runs much of the worlds internet services with, until recently, most deployments running LAMP.



Why NOT Linux?

- Rough around the edges, oft times documentation is poor and no single source for information.
- Driver support can be weak for new hardware, and particularly for hardware that has closed data sheets (WiFi).
- Many applications target other platforms, so no Photoshop, MS Office or Assassin's Creed.
- Many problems need to be resolved on the command line rather than through more intuitive wizards.



So I've convinced you, how do you get it?

- Because Linux is the kernel you need other pieces of software to make an OS.
- Since most/all of it is open source many people have taken Linux and created different variants, called Distributions.
- There are two main core Distributions - Debian and Fedora (created by RedHat).
- Many other distros exists but in general many of these are “Forks” of either Debian or Fedora.

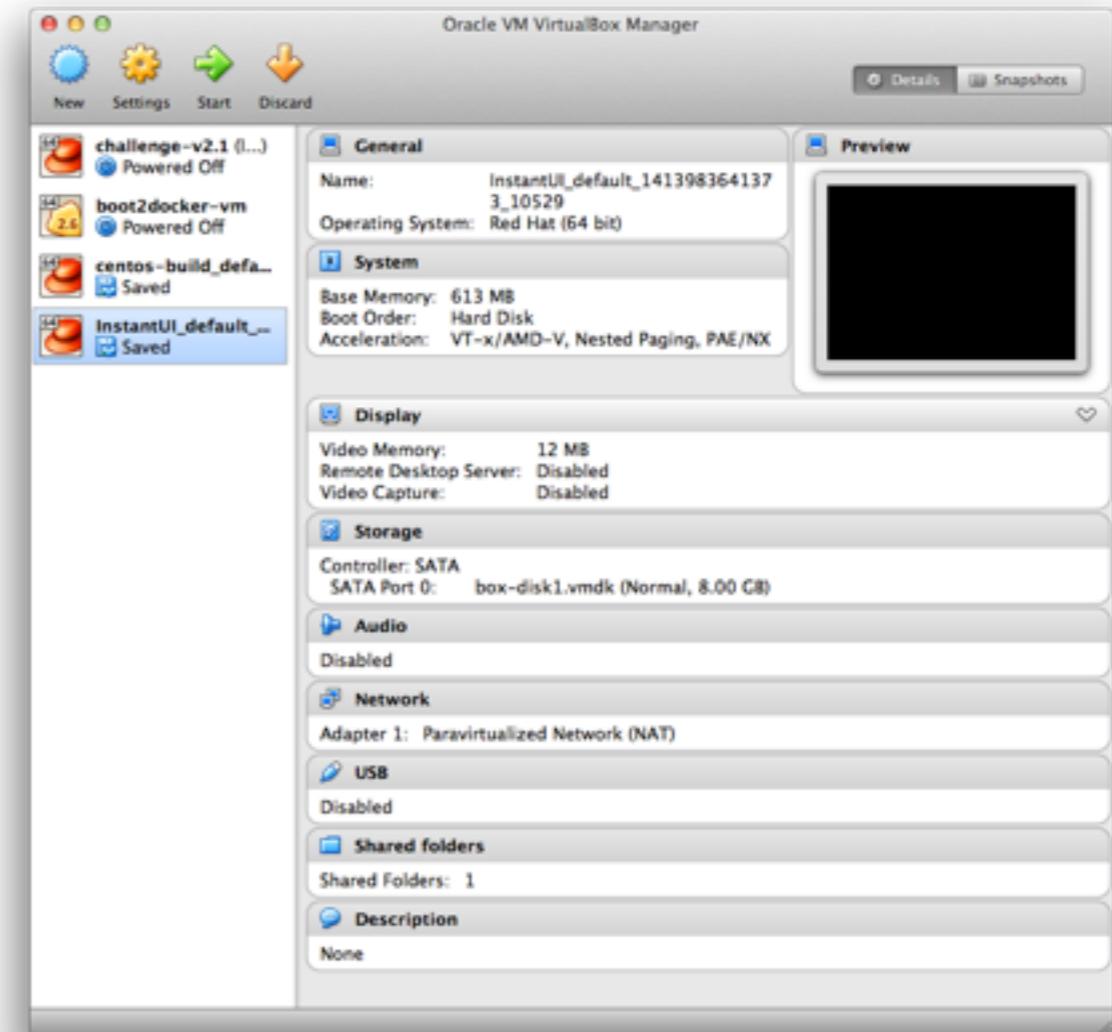


A screenshot of the DistroWatch.com website. The header includes the site's name, a search bar, and links for various languages. The main content area features news headlines, a sidebar for latest distributions, and a central column with a weekly newsletter summary. The footer contains links to ZDNet articles and a "Get The DistroTak!" button.

distrowatch.com

Installing Linux in a VM

- Simplest way to check out Linux.
- VirtualBox is free and available for most platforms (Linux, Windows and OS X).
- Download an ISO of your favourite Distribution and point VirtualBox at it.
- Be sure to give the VM enough memory and enable graphic acceleration for best performance.
- You can also install “guest additions” to allow you to mount directories from your host to the VM.

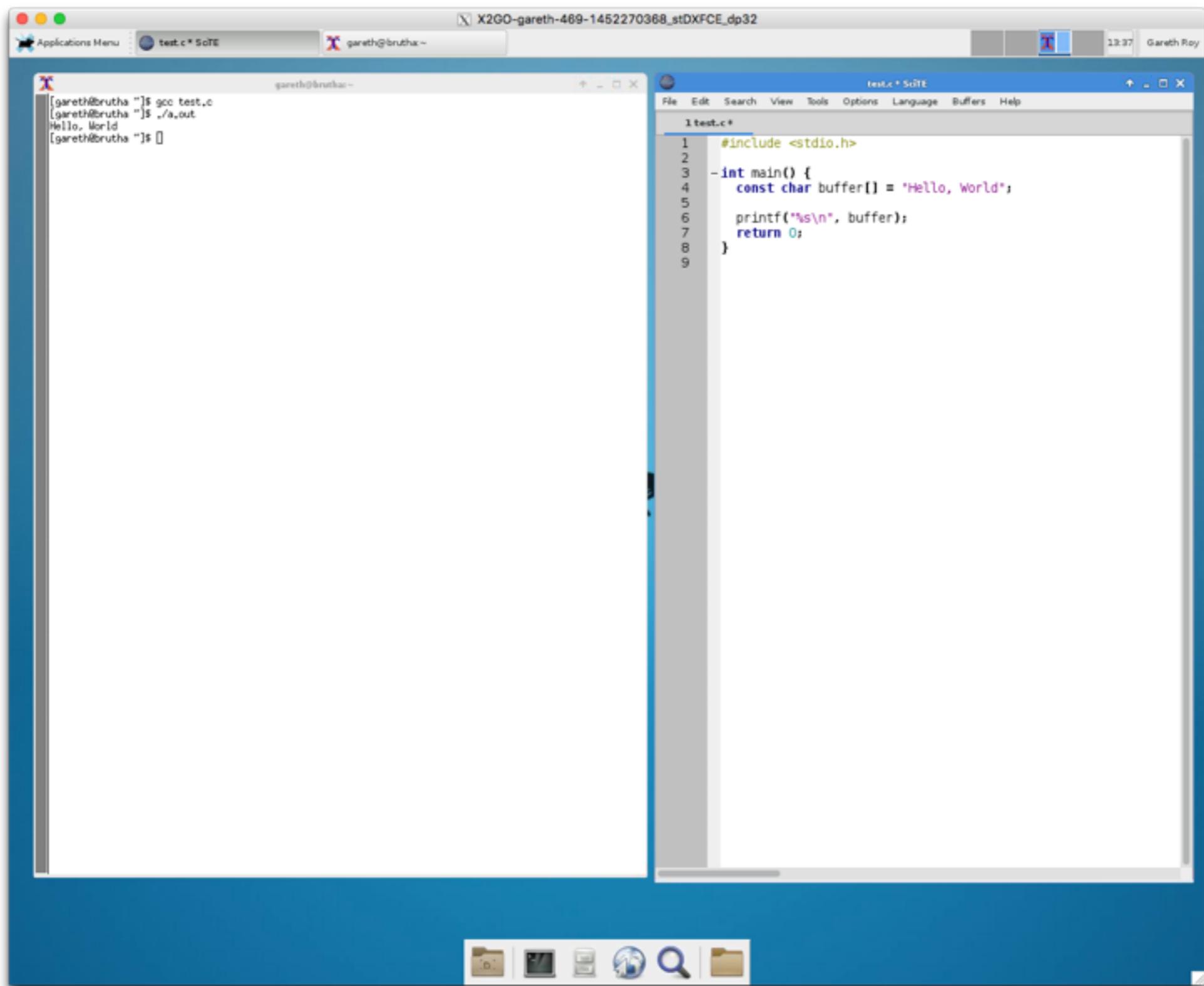


What you'll be using for this course

- For this course you'll be given access to remote linux server.
- You'll be using a distribution known as CentOS.
- This is a fork of Redhat's RHEL, popular in business and academia.
- You'll be able to log into the remote server and get a XFCE desktop via a piece of software called X2GO
- Instructions are up on Moodle2

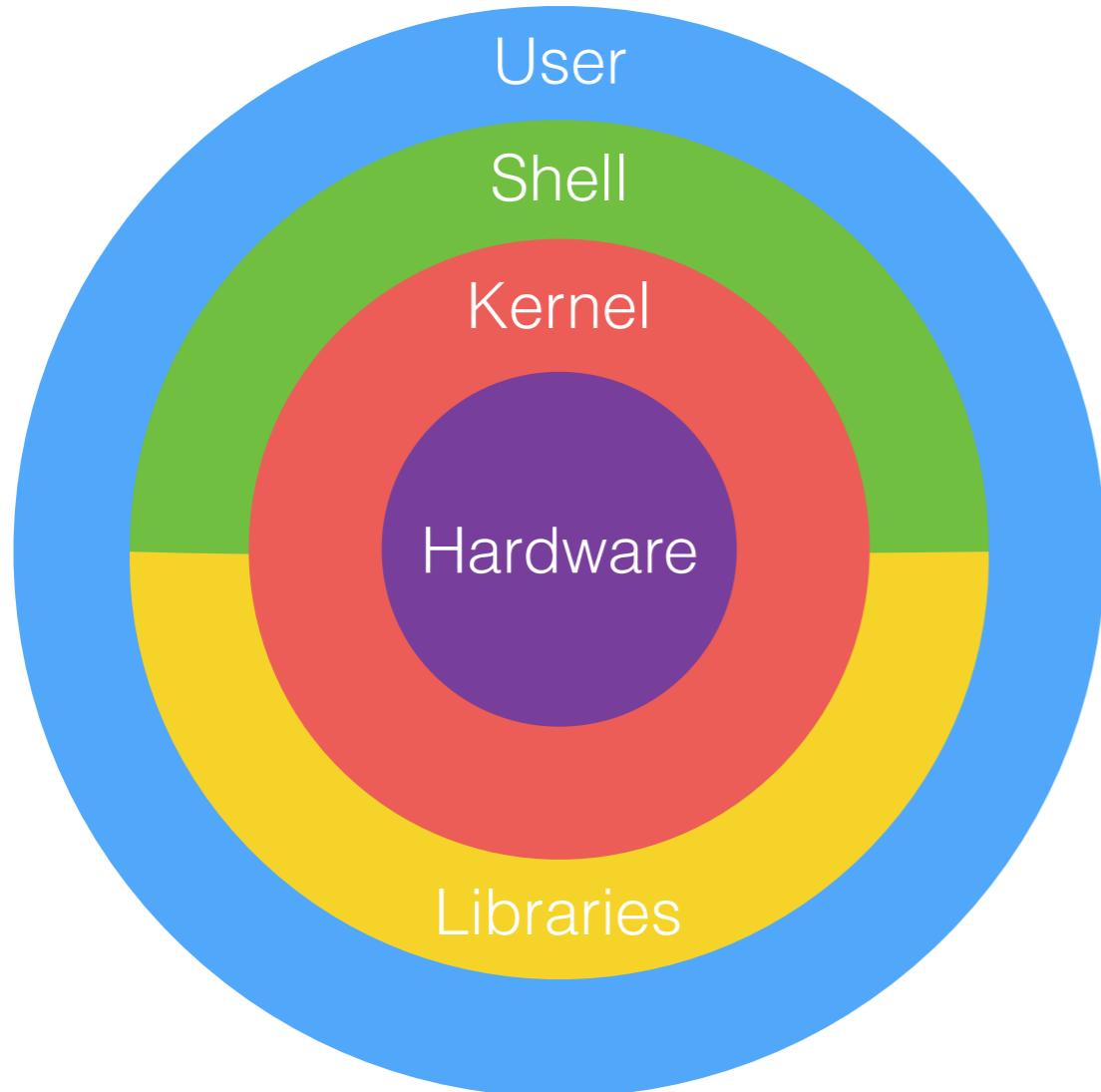


Accessing brutha.physics.gla.ac.uk



- A brief history lesson
- Why Linux?
- The parts of an Operating System
- The Command Line

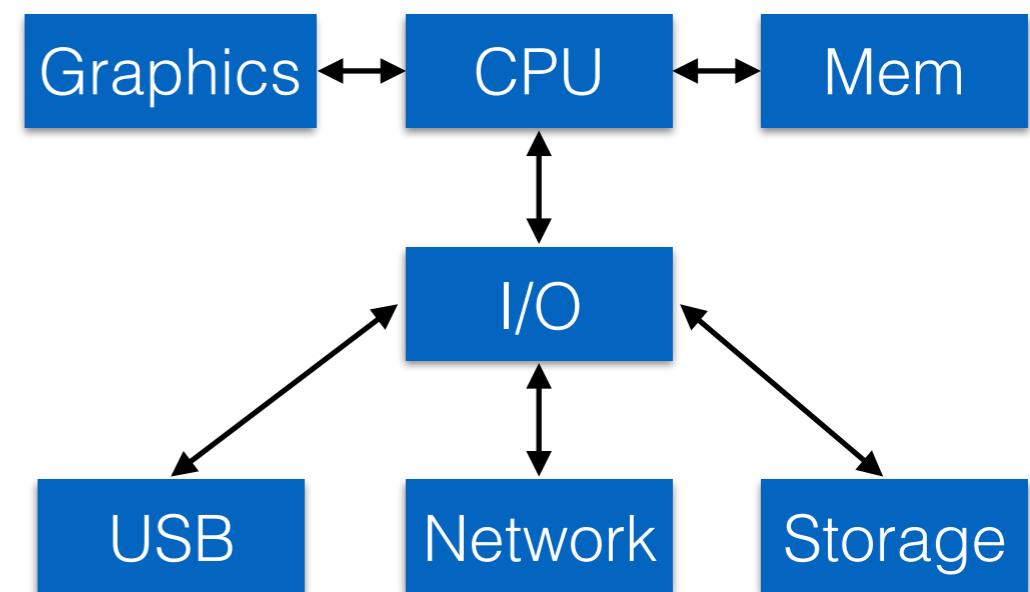
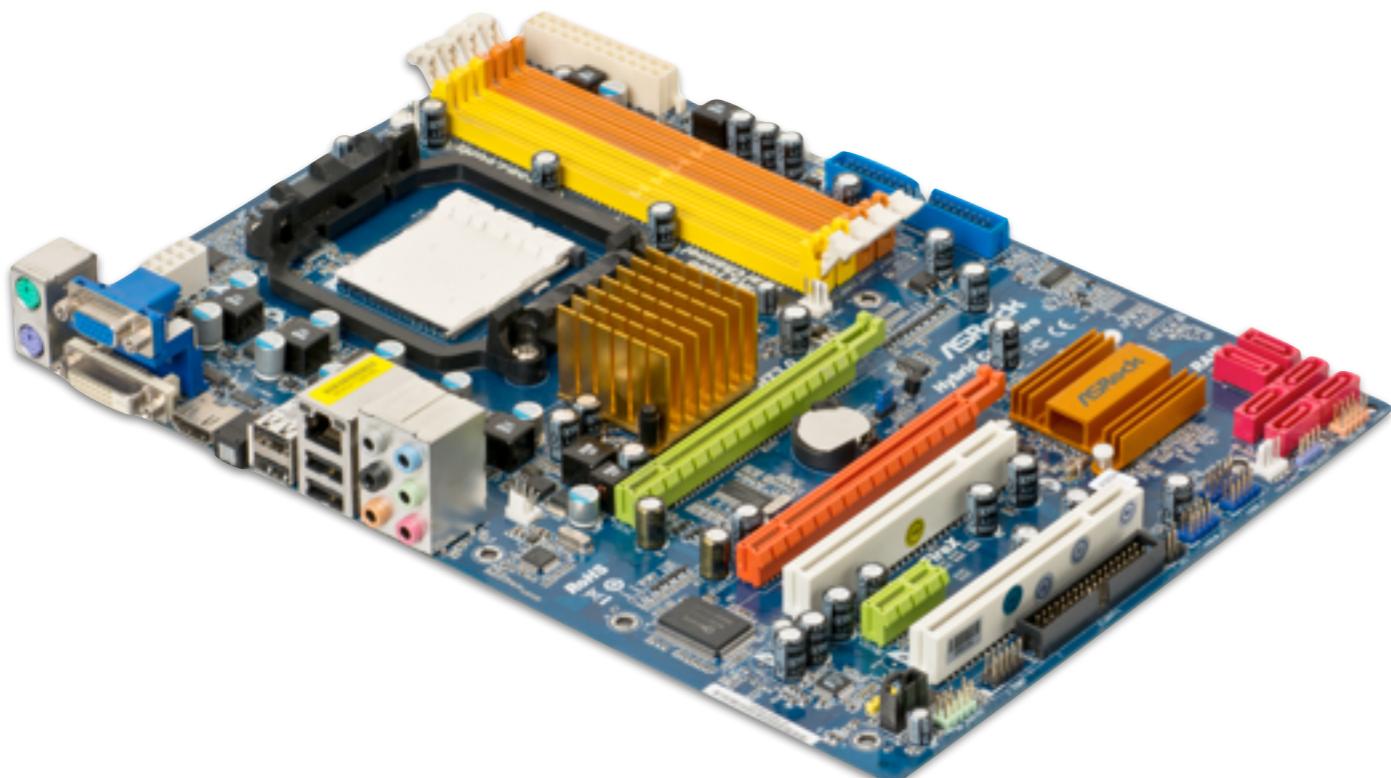
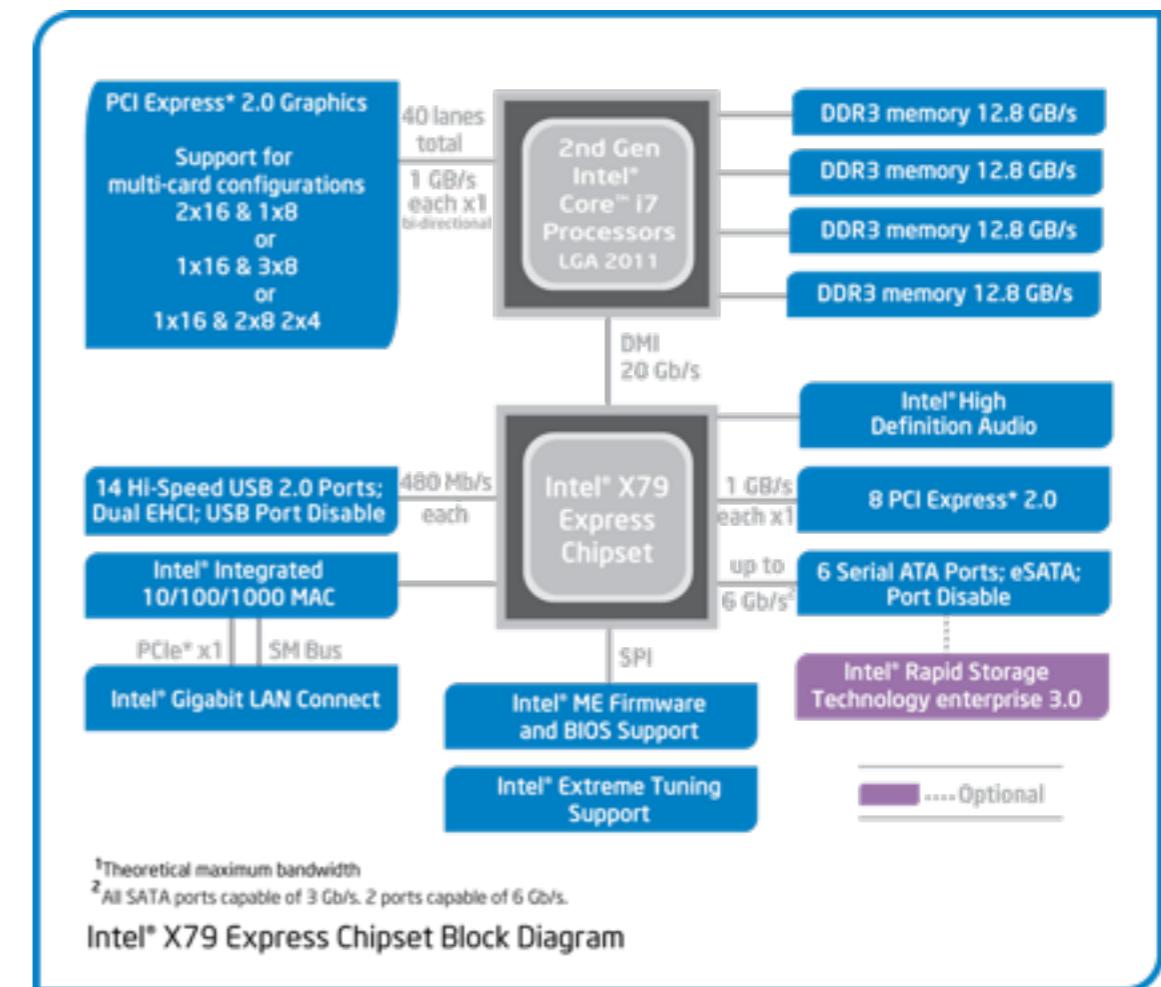
Linux Architecture



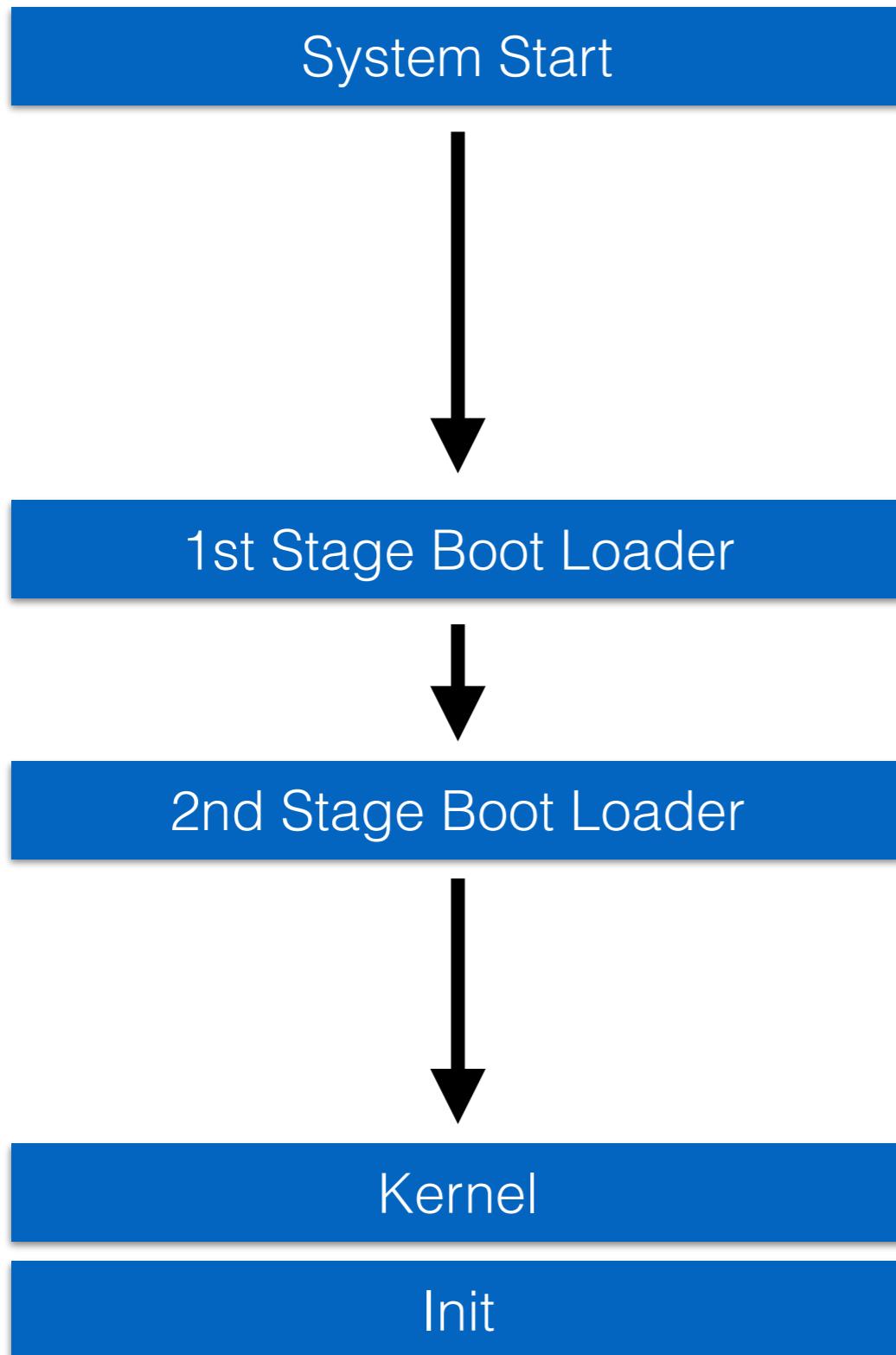
- The Kernel is the core part of Linux, it includes all the drivers need to interact with the underlying hardware.
- The Shell allows interactive access to system resource through a set of builtin commands or by running commands found on the filesystem
- Additionally access to system resources can be via system libraries that expose access to the underlying Kernel (c.f. glibc)
- User applications will usually be started by the shell, or by the init process at system startup.

Hardware

Component	Size	Purpose
CPU	GHz	Numeric / Logic / Shift / Vector Operations
Memory	GB	Running Software and Data storage.
Network	Gbit	Data transmission, TCP/IP packet decoding.
Storage	TB	Data storage preserved without power requirements
Graphics	MOps	VGA Display, 2D/3D Acceleration, Floating point ops.



Boot Process



Power On Self Test (POST)

- Verify BIOS code
- Verify CPU and Registers
- Find and Verify Memory
- Initialise the BIOS
- Discover and Initialise Hardware

Master Boot Record

- 512 bytes located on first sector
- Small program that loads 2nd Stage

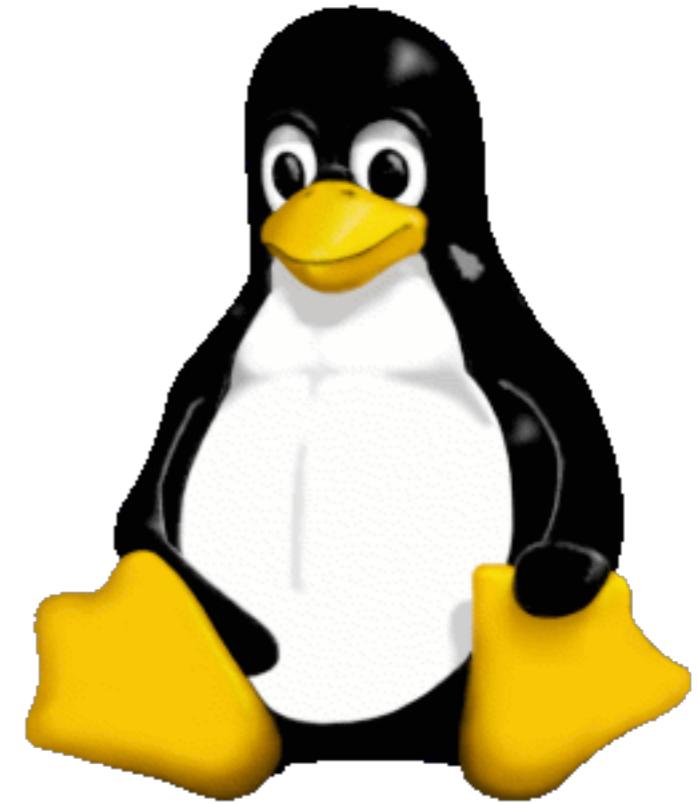
Loads the Kernel

- Loads required components into Mem.
- Enters Protected mode
- Transfers control to the Kernel

Kernel starts and brings up system services. Once devices have been initialised and important subsystems started, starts init services.

Kernel

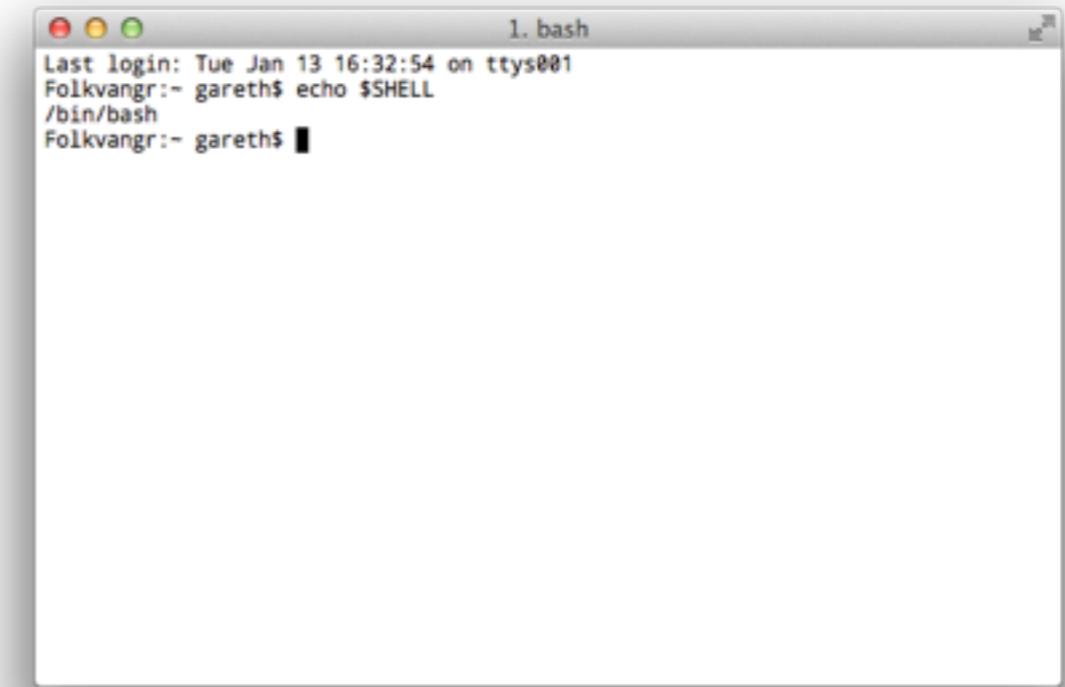
- The Kernel looks after most interactions with Hardware and all of the process running on the system
- It provides:
 - Process scheduler
 - Inter-Process Communication
 - Memory Management
 - A Virtual Filesystem
 - Networking
 - Device Mapper
 - Sound Subsystem
 - etc...



```
isapnp: No Plug & Play device found
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Initializing RF netlink socket
Starting kswapd
UFS: Disk quotas vqmot_6.5.1
vesafb: framebuffer at 0xe0000000, mapped to 0xc2000000, size 51200k
vesafb: mode is 800x600x8, linelength=800, pages<3
vesafb: protected mode interface info at a5f3:1f5f
vesafb: scrolling: redraw
Console: switching to colour frame buffer device 100x32
fb0: VESA VGA frame buffer device
fb0: 256 Unix98 pels configured
Uniform Multi-Platform E-IDE driver Revision: 6.31
ide: Assuming 50MHz system bus speed for PIO modes; override with idebus=<x>
hd: Generic 1234, ATAPI CD-ROM drive
ide0 at 0x1f0-0x1f7,0x3f6 on irq 14
hd: ATAPI 4X CD-ROM drive, 512k Cache
Uniform CD-ROM driver Revision: 3.12
FDC 0 is an 8272A
RMDISK driver initialized: 16 RMD disks of 4096k size 1024 blocksize
Cronyx Ltd. Synchronous PPP and CISCO HDLC (c) 1994
Linux port (c) 1998 Building Number Three Ltd & Jan "Yengo" Kasprzak.
SCSI subsystem driver Revision: 1.00
scsi0 : SCSI host adapter emulation for IDE ATAPI devices
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP, IGMP
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 2048 bind 2048)
NET4: Unix domain sockets 1.0-SMP for Linux NET4.0.
RMDISK: Compressed image found at block 0
```

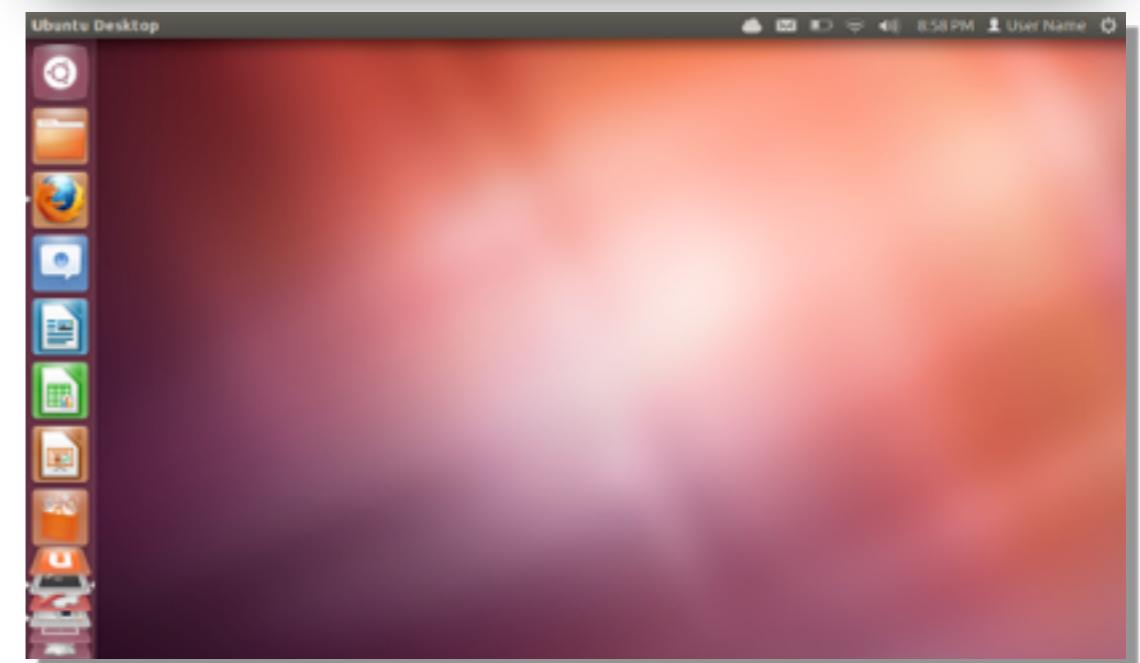
Shell

- In Linux the shell handles interaction between Users and the system.
- It can be via a text interface known as the command line as well as through a graphical interface, for instance the “gnome-shell”
- It allows other processes/applications to be started such as editors, compilers, office suites.
- We'll cover the command line in more detail later in this set of slides.



A screenshot of a terminal window titled "1. bash". The window shows a command-line session:

```
Last login: Tue Jan 13 16:32:54 on ttys001
Folkvangr:- gareth$ echo $SHELL
/bin/bash
Folkvangr:- gareth$
```



System Libraries

- Running programs also need to interact with system resources.
- This is carried out via system libraries.
- There are a wide range of these most of which are beyond the scope of this course.
- You will however use the standard C libraries as part of the C programming component of this course.



```
gareth@brutha:/usr/lib32$ ls
crt1.o          libcrypt.a      libmcheck.a      libquadmath.so.0
crti.o          libcrypt.so     libm.so          libquadmath.so.0.0.0
crtn.o          libc.so         libnsl.a        libresolv.a
gconv           libdl.a         libnsl.so       libresolv.so
gcrt1.o         libdl.so        libnss_compat.so librt.a
libanl.a         libgcc.a       libnss_dns.so   librt.so
libanl.so        libgcc_s.so.1  libnss_files.so libstdc++.so.6
libBrokenLocale.a libgfortran.so.3 libnss_hesiod.so libstdc++.so.6.0.16
libBrokenLocale.so libgfortran.so.3.0.0 libnss_nisplus.so libthread_db.so
libbsd-compat.a libgomp.so.1    libnss_nis.so    libutil.a
libc.a          libgcc_s.so.1  libieee.a      libpthread.a
libcidn.so       libgomp.so.1.0.0 libm.a          libpthread_nonshared.a
libc_nonshared.a libmcheck.a    libpthread.so   Mlibcrt1.o
gareth@brutha:/usr/lib32$
```

Userland

- User processes are the final level of the Operating System.
- This is likely the most familiar and includes programs such as web browsers, office suites, editors etc.
- Most user orientated distributions ship with software that allows you to do most common tasks.



- A brief history lesson
- Why Linux?
- The parts of an Operating System
- The Command Line

The Command Line

- There are a variety of different shells available: Sh, BASH, CSh, TCSH, ZSh...
- C-style shells: C-like syntax for scripting
- Bash-style shells: slightly more consistent, and more widely adopted in the research-computing community.
- We'll use Bash in this course, but feel free to experiment!
- To open the terminal, click the ubuntu button and type "terminal"!

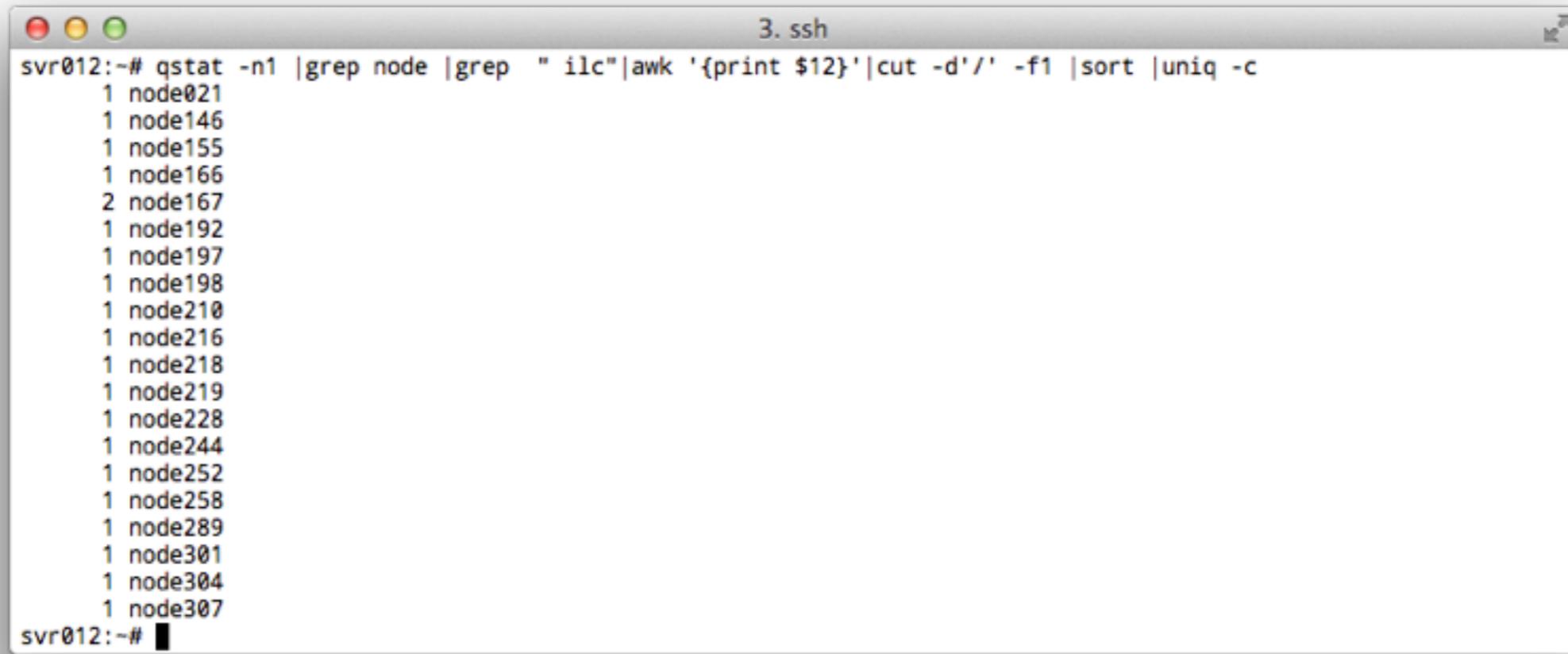


The Command Line

- The command line lets us input command at the shell prompt and receive output in a text form.
- Commands are of the form:

```
prompt# <command> <flags> <arguments>
```

```
prompt# du -h --max-depth=1
```
- Commands can be chained together to build up complex interactions and obtain the results we want.



A screenshot of a terminal window titled "3. ssh". The window contains a command-line session on a Linux system named "svr012". The user has run the following command:

```
svr012:~# qstat -n1 |grep node |grep " ilc" |awk '{print $12}' |cut -d'/' -f1 |sort |uniq -c
```

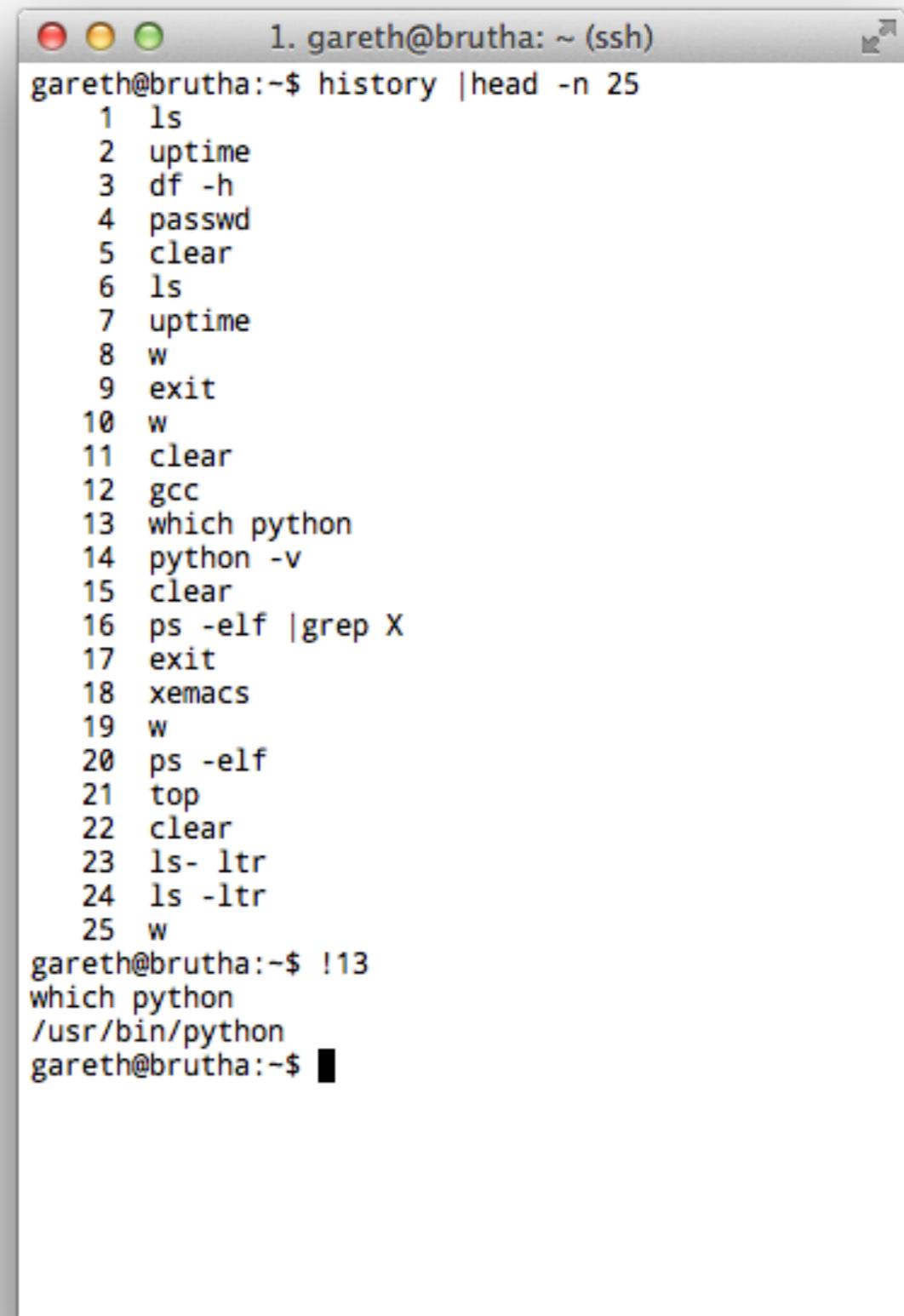
The output shows the count of each node ID:

Count	Node ID
1	node021
1	node146
1	node155
1	node166
2	node167
1	node192
1	node197
1	node198
1	node210
1	node216
1	node218
1	node219
1	node228
1	node244
1	node252
1	node258
1	node289
1	node301
1	node304
1	node307

The terminal prompt "svr012:~#" is visible at the bottom.

Useful Commands

- **history** - gives all commands you've entered
- **!!** - runs the last command
- **!<number>** runs the command at <number> i.e. !484 runs command 484
- **up/down arrow** cycles through previous commands
- **tab** auto completes names



A screenshot of a terminal window titled "1. gareth@brutha: ~ (ssh)". The window displays the output of the command "history | head -n 25", which shows a list of 25 commands run by the user. The commands include various system utilities like ls, uptime, df, passwd, clear, w, exit, ps, top, and xemacs. Below this, the user types "gareth@brutha:~\$!13" and presses enter, which executes the command "which python" and shows its path as "/usr/bin/python". The terminal window has a standard OS X style with red, yellow, and green close buttons.

```
gareth@brutha:~$ history | head -n 25
 1  ls
 2  uptime
 3  df -h
 4  passwd
 5  clear
 6  ls
 7  uptime
 8  w
 9  exit
10  w
11  clear
12  gcc
13  which python
14  python -v
15  clear
16  ps -elf |grep X
17  exit
18  xemacs
19  w
20  ps -elf
21  top
22  clear
23  ls -ltr
24  ls -ltr
25  w
gareth@brutha:~$ !13
which python
/usr/bin/python
gareth@brutha:~$ █
```

Useful Commands (2)

- **man** will give you the manual page for a particular command.
 - hitting the **spacebar** takes you to the next page and **q** quits
- **ctrl-r** allows you to search through previous commands
- **clear** empties the screen (useful for clearing your head sometimes).

The screenshot shows a terminal window with the title "1. sh". The content of the window is the man page for the "ls" command. It includes sections for NAME, SYNOPSIS, DESCRIPTION, and a list of options:

NAME
ls -- list directory contents

SYNOPSIS
ls [-ABCFGHLOPRSTUW@abcdefghijklmnopqrstuvwxyz] [file ...]

DESCRIPTION
For each operand that names a file of a type other than directory, ls displays its name as well as any requested, associated information. For each operand that names a file of type directory, ls displays the names of files contained within that directory, as well as any requested, associated information.

If no operands are given, the contents of the current directory are displayed. If more than one operand is given, non-directory operands are displayed first; directory and non-directory operands are sorted separately and in lexicographical order.

The following options are available:

- @ Display extended attribute keys and sizes in long (-l) output.
- 1 (The numeric digit ``one''). Force output to be one entry per line. This is the default when output is not to a terminal.
- A List all entries except for . and .. Always set for the super-user.
- a Include directory entries whose names begin with a dot (.).
- B Force printing of non-printable characters (as defined by ctype(3) and current locale settings) in file names as \xxx, where xxx is the numeric value of the character in octal.
- b As -B, but use C escape codes whenever possible.
- C Force multi-column output; this is the default when output is to a terminal.
- c Use time when file status was last changed for sorting (-t) or long printing (-l).
- d Directories are listed as plain files (not searched recursively).
- e Print the Access Control List (ACL) associated with the file, if present, in long (-l) output.
- F Display a slash ('/') immediately after each pathname that is a directory, an asterisk ('*') after each that is executable, an at

- A brief history lesson
- Why Linux?
- The parts of an Operating System
- The Command Line



The Filesystem

Dr. Gareth Roy (x6439)
gareth.roy@glasgow.ac.uk

- The Filesystem
- Navigating the filesystem
- Users & Permissions
- Working with files
- Archives

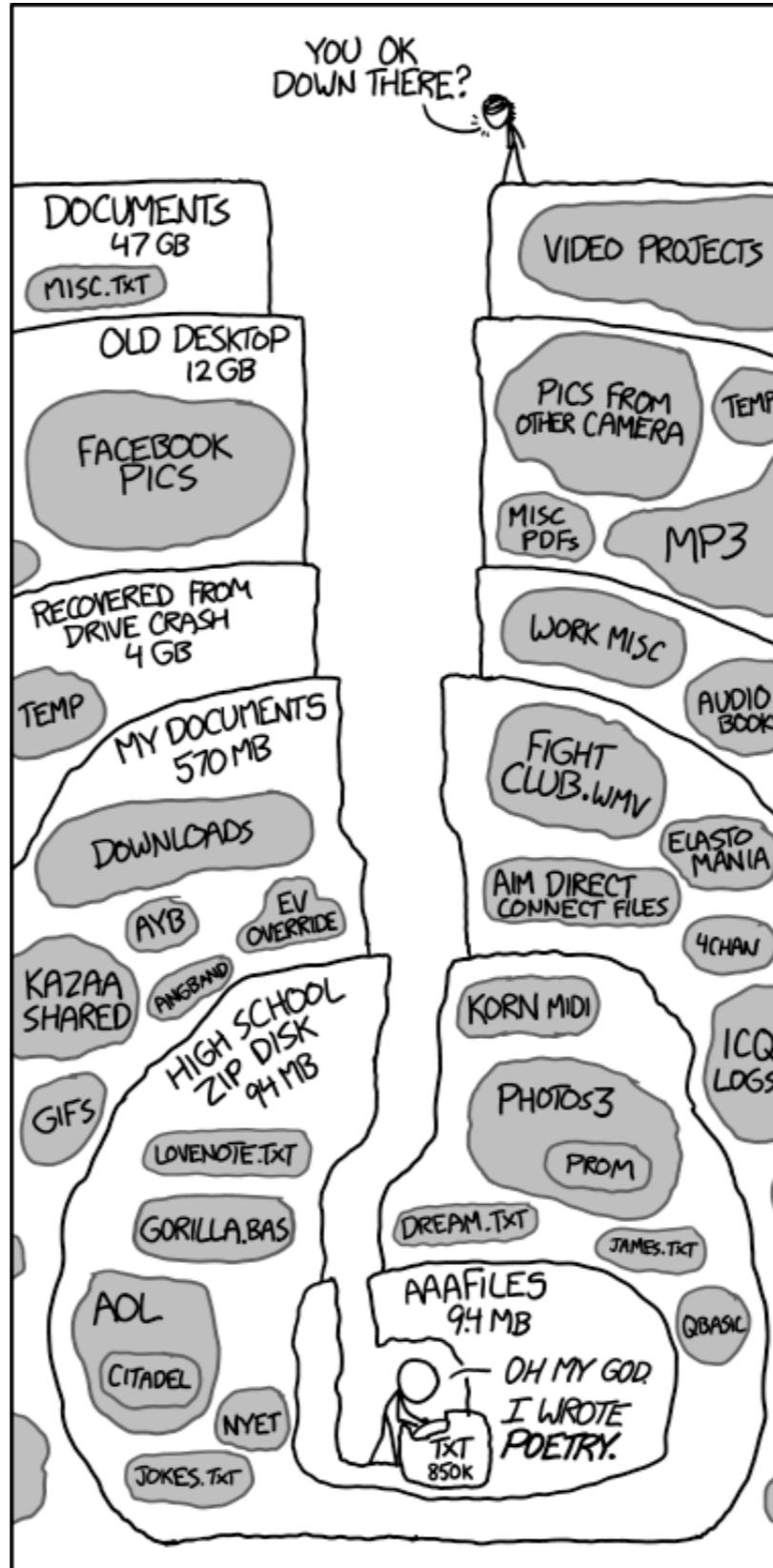
- The Filesystem
- Navigating the filesystem
- Users & Permissions
- Working with files
- Archives

"I think the major good idea in Unix was its clean and simple interface: open, close, read, and write."

— Ken Thompson

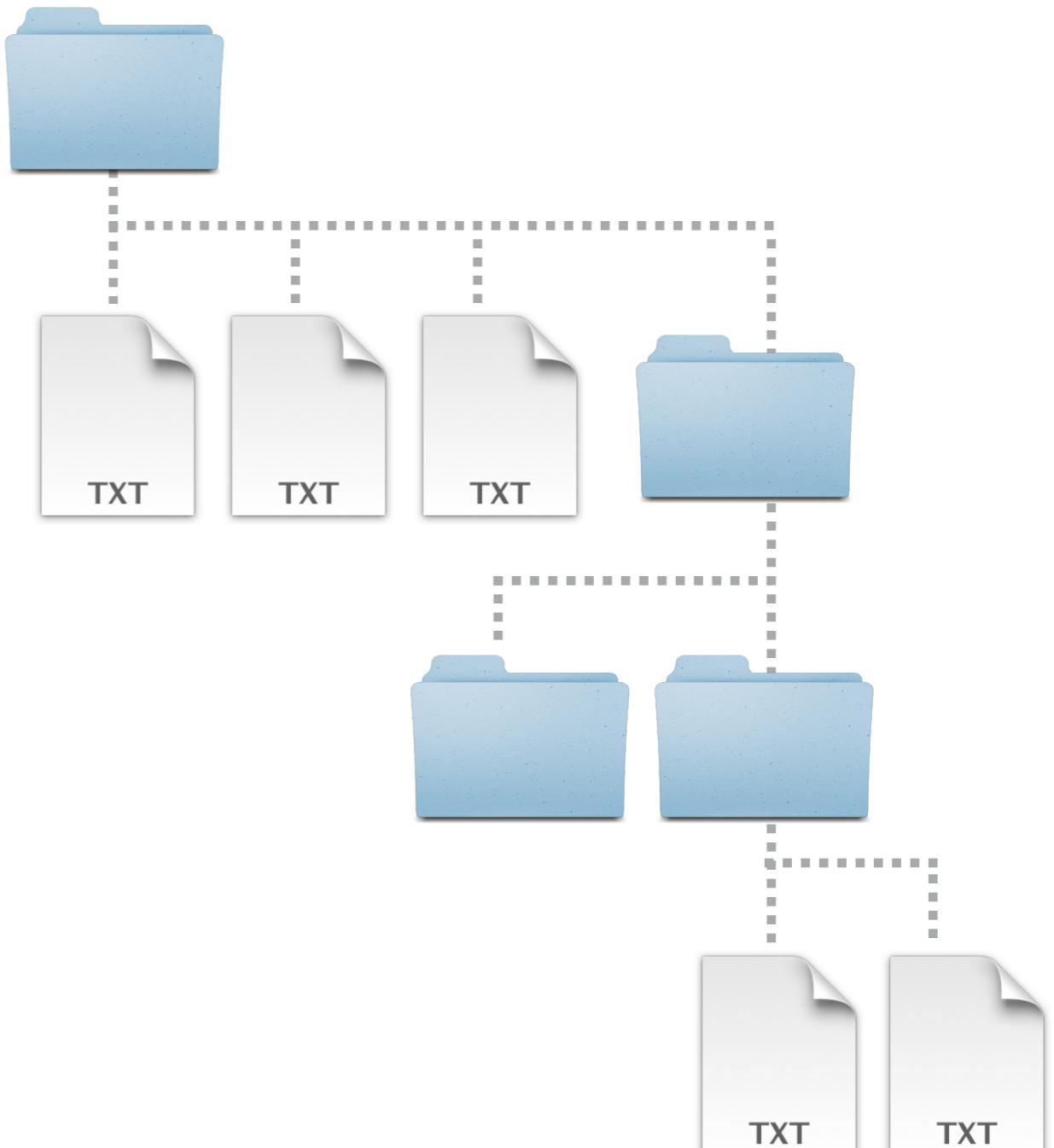
"On a UNIX system, everything is a file; if something is not a file, it is a process."

— Unknown



The Filesystem

- The Linux filesystem is hierarchical.
- It has a tree like structure that begins at the “root” node.
- Each node in the tree can be a file or directory (or link, socket...).
- A directory is a special file that contains a list of other files, and can include other directories.
- Each “file” in the filesystem has a name, which is case sensitive (afile is not the same as AFILE).
- Each “file” or node in the tree has a unique identifier called it's **inode**.

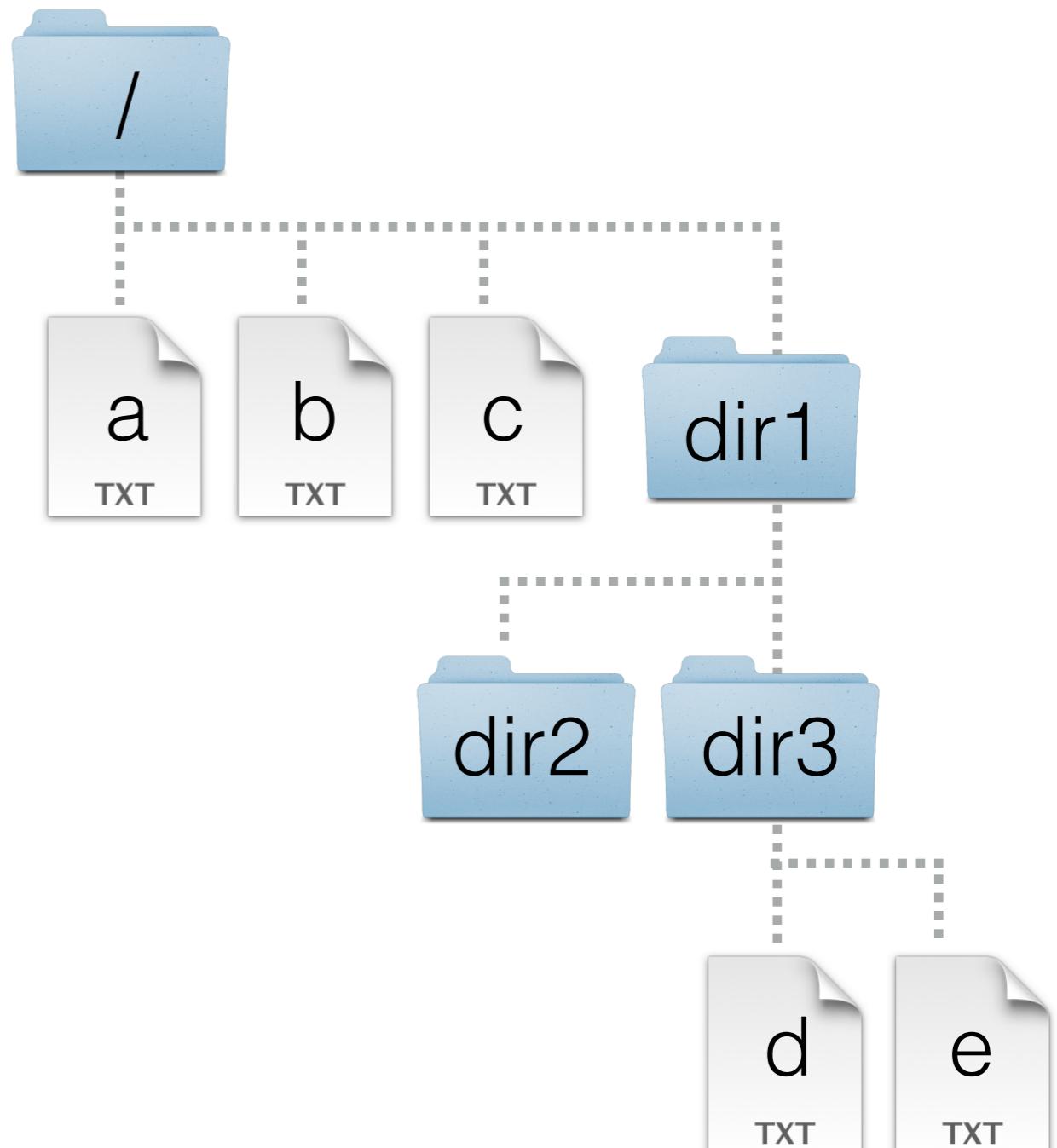


File types

Type	Symbol	Purpose
Regular	-	Standard text files
Directories	d	Files that are lists of files
Links	l	A special file that allows another file to appear in multiple locations in the filesystem.
Special File	c	Input / Ouput sources (usually located in /dev)
Sockets	s	inter process networking similar to TCP/IP
Named Pipes	p	Similar to a socket, allows process to communicate with one another.

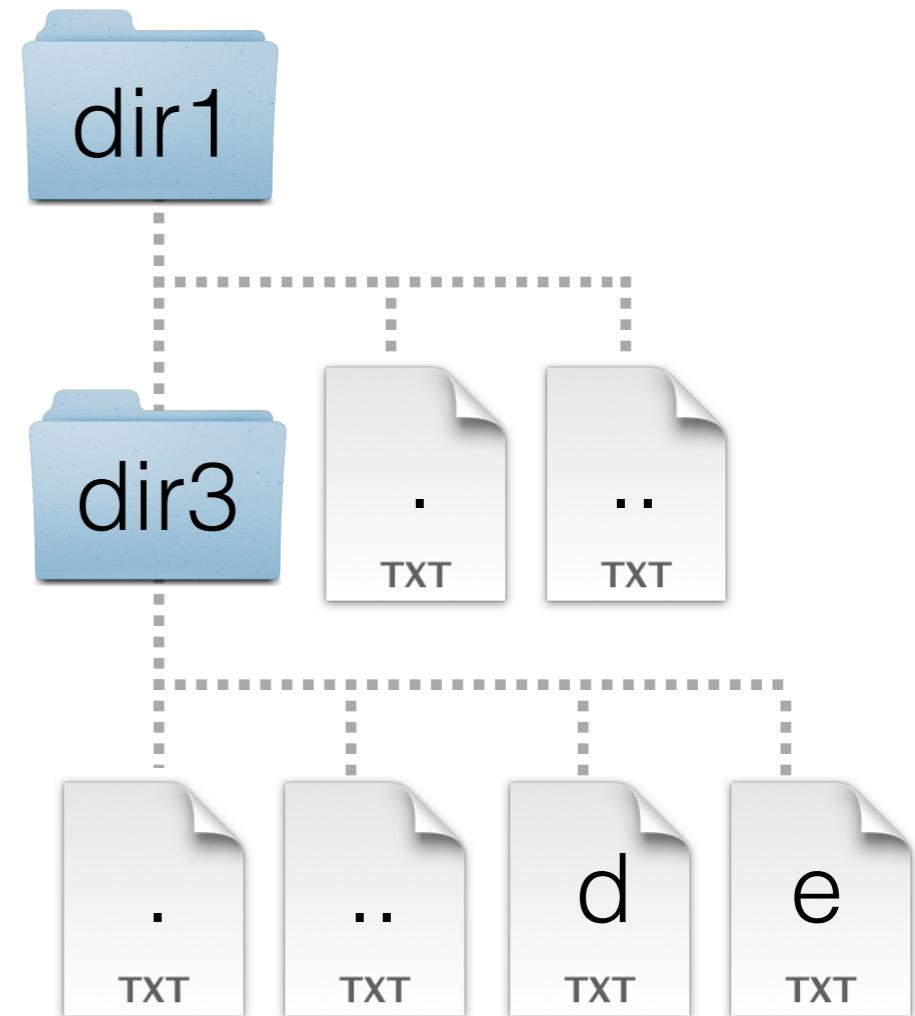
Absolute File Paths

- The location of each file is given by its **path**.
- The absolute path starts from the “root” node denoted by a **/**
- In this example the absolute path for the file ‘a’ would be **/a**
- Each time we descend into a directory we add an additional **/**
- So the absolute path for the file ‘e’ would be **/dir1/dir3/e**
- We can locate any file in the filesystem by using it’s absolute path.



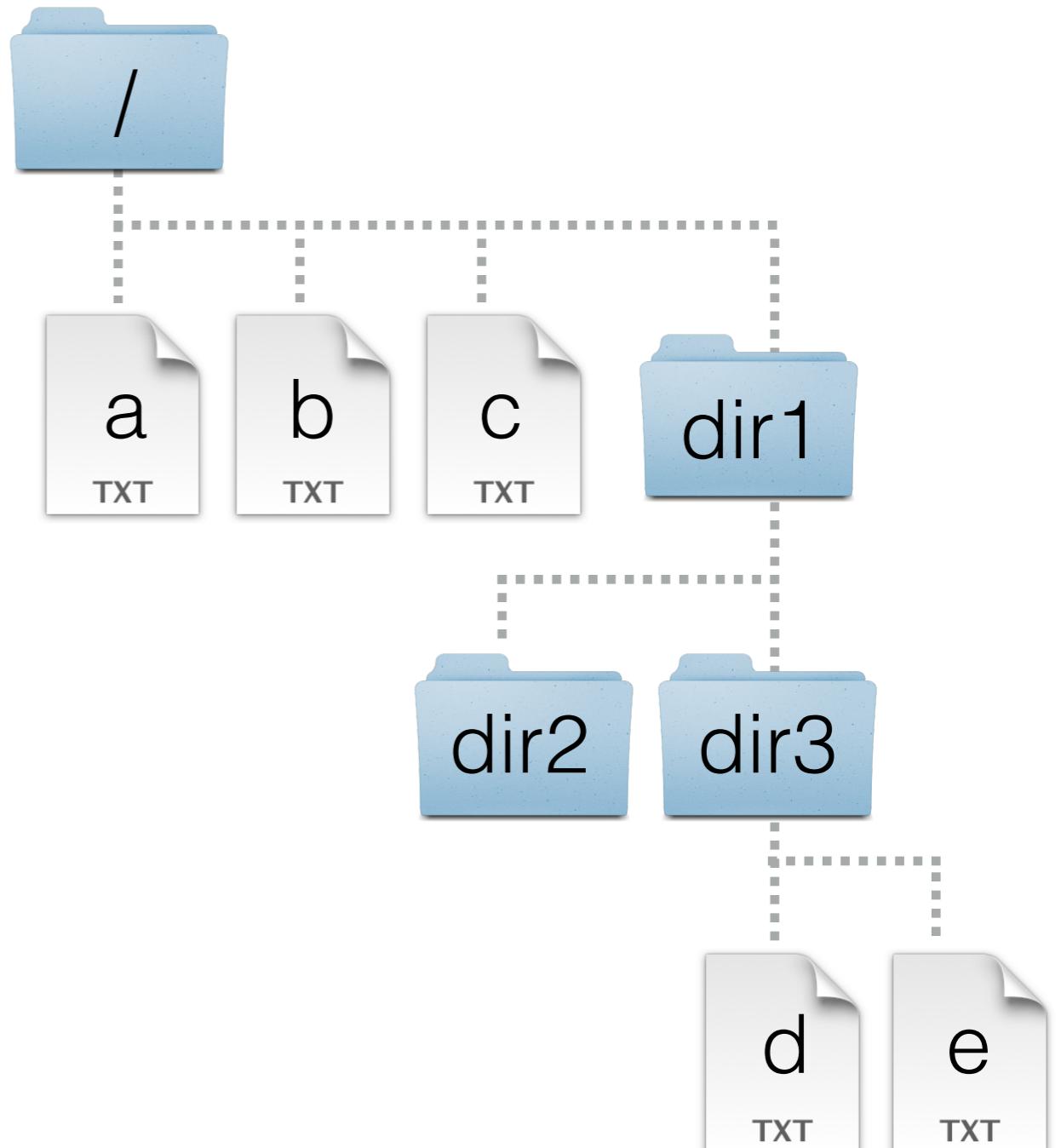
Hidden Files

- In Linux files can be hidden so they don't appear in a directory.
- Hidden files in Linux all begin with '.' character
- This is often used for configuration files, like **.vimrc**
- There are two special hidden files found in every directory.
- **.** - the current directory.
- **..** - the directory above this one in the hierarchy.



Relative File Paths

- Relative paths can be used to references files at other parts of the file system.
- This is useful as absolute paths can make a system inflexible or difficult to modify.
- For instance if we were presently working in **dir2** we could access file '**e**' via: **../dir3/e**
- This can be interpreted as a set of actions:
 - go up one level (..)
 - enter **dir3**
 - access file '**e**'



Standard Filesystem Locations

- **/etc** - holds a lot of configuration files, e.g. /etc/cups controls the configuration of the printer queues and which devices are made available at boot.
- **/dev** - holds links to the various pieces of hardware being managed by the kernel
- **/proc** - holds views into the current kernel operating parameters and processes
- **/var** - holds ‘variable’ data, e.g. logging files, mail, printer spools
- **/usr** - holds binaries, documentation, software libraries
- **/home** - is the collection of private directories for users of the system
- **/bin** - contains the binaries that are essential to the system (shells, cp, mv, rm, cat, ls, etc...)
- **/sbin** - contains the binaries that are essential to the system but only for use by administrators

- The Filesystem
- Navigating the filesystem
- Users & Permissions
- Working with files
- Archives

Looking Around

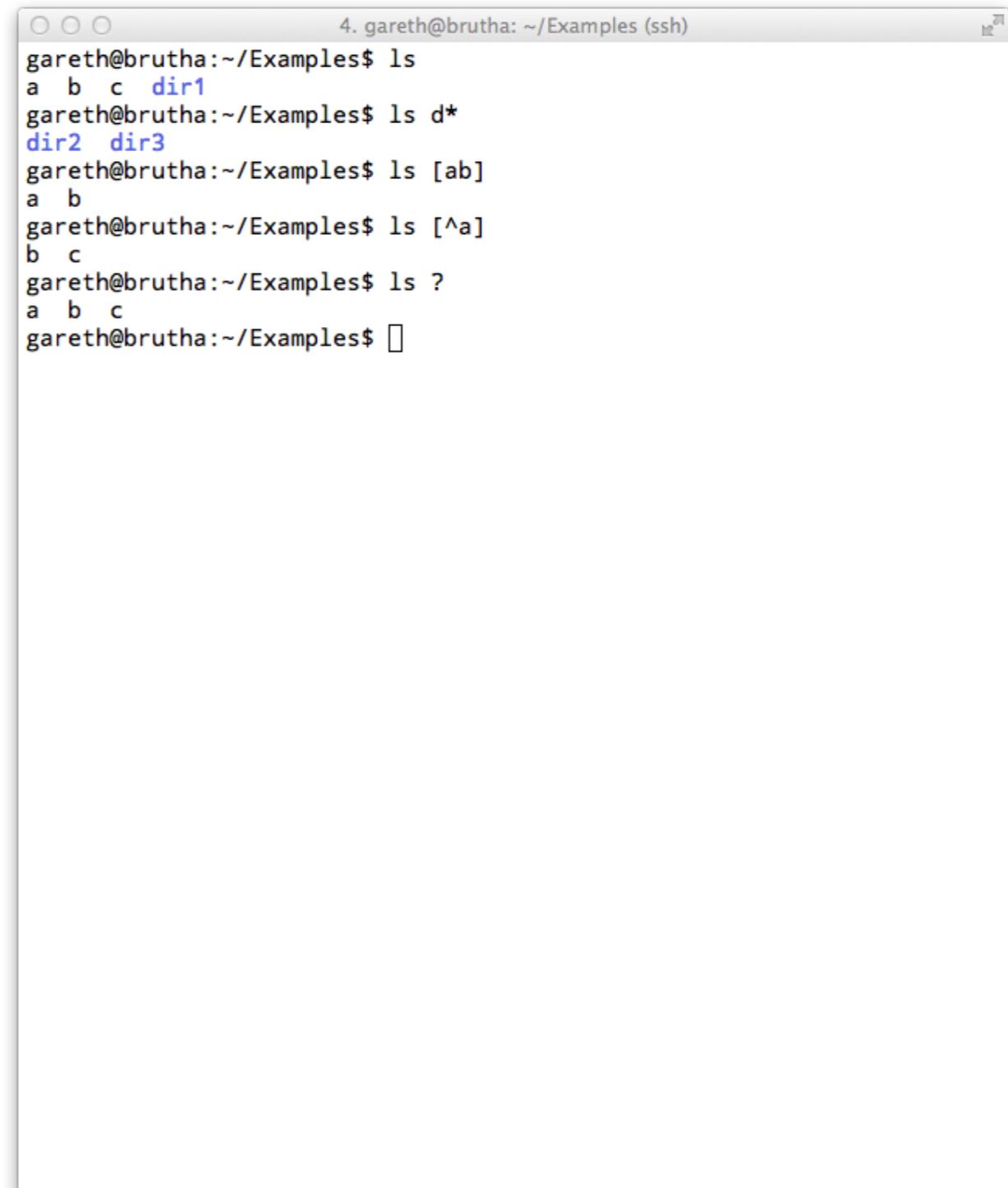
- **pwd** - tells us our current position in the filesystem (print working directory).
- **tree** - prints out the complete ‘tree’ of our current location, good for getting a visual representation of a directory.
- **ls** - lists the contents of a directory. By itself defaults to the current directory, but can be given an absolute or relative path.
- **ls -a** - the minus a option tells ls to show hidden files.
- **ls -l** - the minus l option tells ls print the contents of a directory in long format.

```
4. gareth@brutha: ~/Examples (ssh)
gareth@brutha:~/Examples$ pwd
/home/gareth/Examples
gareth@brutha:~/Examples$ gareth@brutha:~/Examples$ gareth@brutha:~/Examples$ tree
.
├── a
└── b
    ├── c
    └── dir1
        ├── dir2
        └── dir3
            ├── d
            └── e

3 directories, 5 files
gareth@brutha:~/Examples$ gareth@brutha:~/Examples$ gareth@brutha:~/Examples$ ls
a b c dir1
gareth@brutha:~/Examples$ gareth@brutha:~/Examples$ gareth@brutha:~/Examples$ ls -a
. ... a b c dir1
gareth@brutha:~/Examples$ gareth@brutha:~/Examples$ gareth@brutha:~/Examples$ ls -l
total 4
-rw-rw-r-- 1 gareth gareth 0 Jan 14 09:05 a
-rw-rw-r-- 1 gareth gareth 0 Jan 14 09:05 b
-rw-rw-r-- 1 gareth gareth 0 Jan 14 09:05 c
drwxrwxr-x 4 gareth gareth 4096 Jan 14 09:28 dir1
gareth@brutha:~/Examples$ 
```

File Globbing

- **ls** can also be used to list specific files, and the list can be filtered by using wildcards.
- Wildcards are symbols that have special meanings when combined with filenames.
 - * - matches all characters
 - ? - matches only one character.
 - [] - specifies alphanumeric values to match on i.e. [a-c]
 - ^ - negates a match
- For example we could match all the files that have a 'txt' extension by the following:
 - **ls *.txt**
- Or we could match on files whose name are only a single character
 - **ls ?**

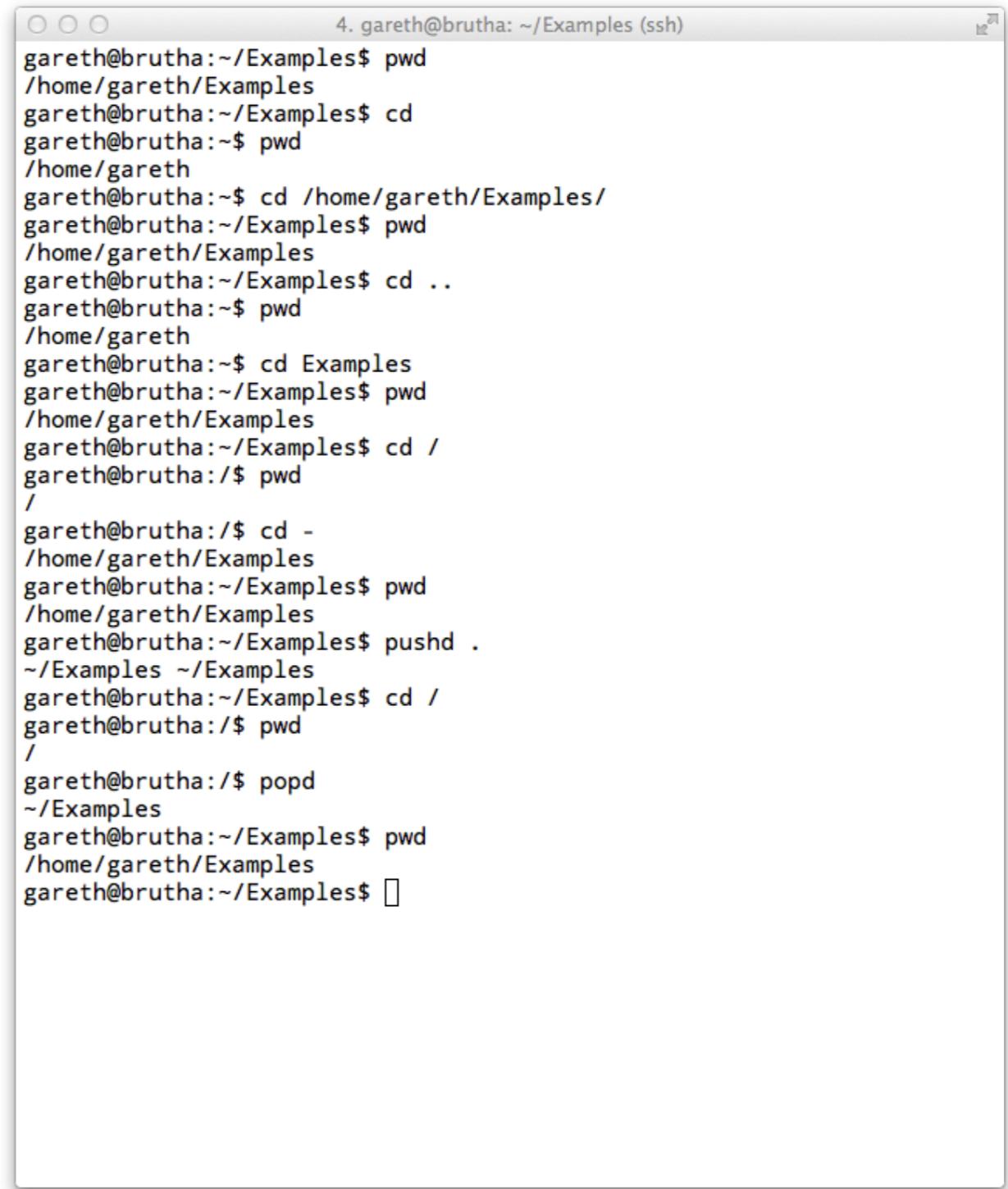


The screenshot shows a terminal window titled "4. gareth@brutha: ~/Examples (ssh)". It displays several commands and their outputs:

- gareth@brutha:~/Examples\$ ls
a b c dir1
- gareth@brutha:~/Examples\$ ls d*
dir2 dir3
- gareth@brutha:~/Examples\$ ls [ab]
a b
- gareth@brutha:~/Examples\$ ls [^a]
b c
- gareth@brutha:~/Examples\$ ls ?
a b c
- gareth@brutha:~/Examples\$

Moving Around

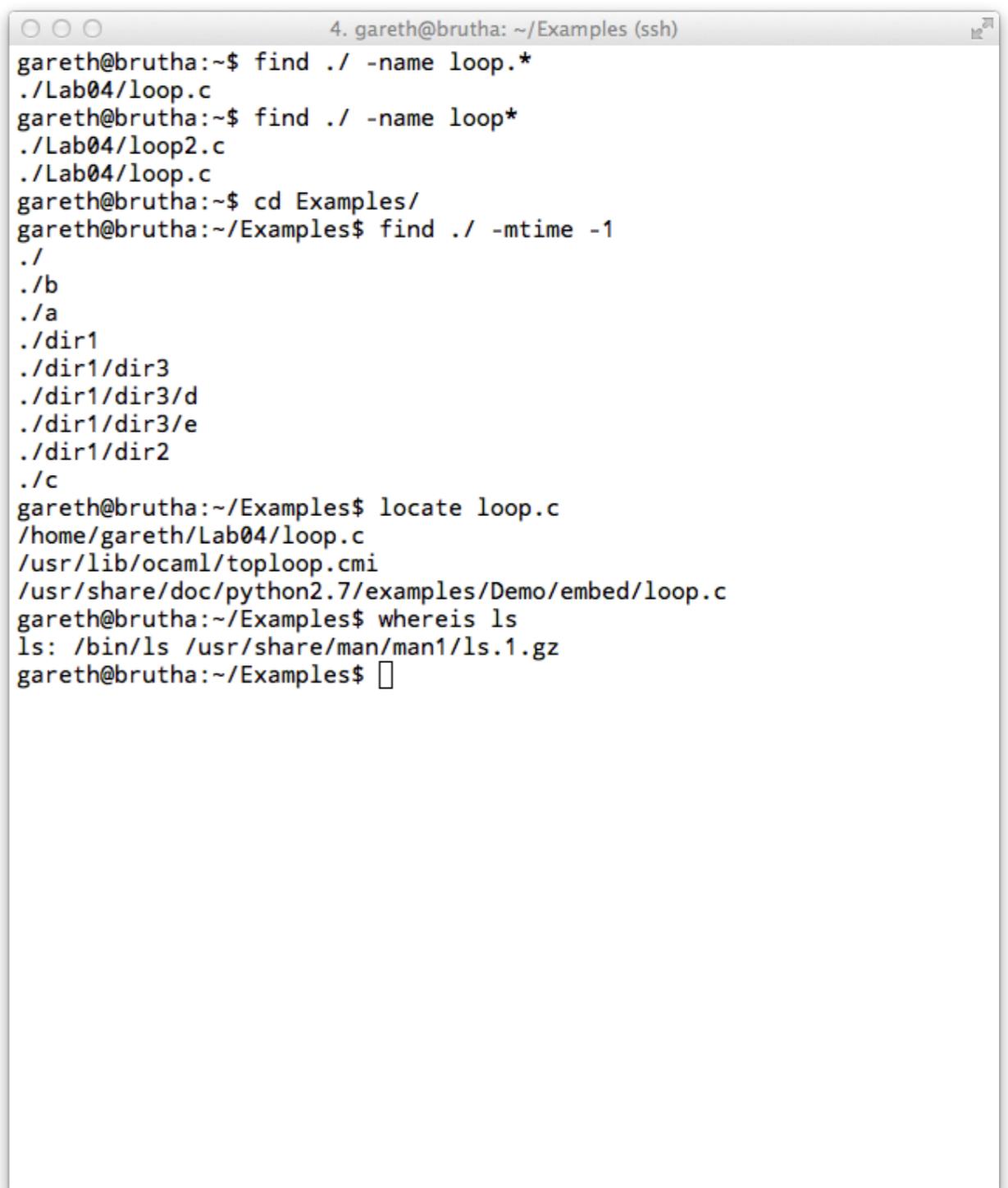
- **cd** - command to “change directory”
- **cd** - with no arguments will return you to your home directory.
- **cd <dir>** - changes you to a directory <dir>. This can be an absolute or relative path.
- **cd -** - will return you to the previous directory.
- **pushd** - stores a directory path you'd like to visit later
- **popd** - takes the first directory stored and changes to it.



```
4. gareth@brutha: ~/Examples (ssh)
gareth@brutha:~/Examples$ pwd
/home/gareth/Examples
gareth@brutha:~/Examples$ cd
gareth@brutha:~$ pwd
/home/gareth
gareth@brutha:~$ cd /home/gareth/Examples/
gareth@brutha:~/Examples$ pwd
/home/gareth/Examples
gareth@brutha:~/Examples$ cd ..
gareth@brutha:~$ pwd
/home/gareth
gareth@brutha:~$ cd Examples
gareth@brutha:~/Examples$ pwd
/home/gareth/Examples
gareth@brutha:~/Examples$ cd /
gareth@brutha:$ pwd
/
gareth@brutha:$ cd -
/home/gareth/Examples
gareth@brutha:~/Examples$ pwd
/home/gareth/Examples
gareth@brutha:~/Examples$ pushd .
~/Examples ~/Examples
gareth@brutha:~/Examples$ cd /
gareth@brutha:$ pwd
/
gareth@brutha:$ popd
~/Examples
gareth@brutha:~/Examples$ pwd
/home/gareth/Examples
gareth@brutha:~/Examples$
```

Finding Files

- **find** - used to search through the filesystem to find a file.
 - `find ./ -name Lab04`
 - `find ./ -mtime +2`
 - `find ./ -mtime -3`
- **locate** - accesses a database of files (usually built once a day), is faster than find but only as good as the database.
- Use find to get newer files, and if you want to filter on things like modification or creation time
- **whereis** - will return the location of programs and manpages
 - `whereis ls`

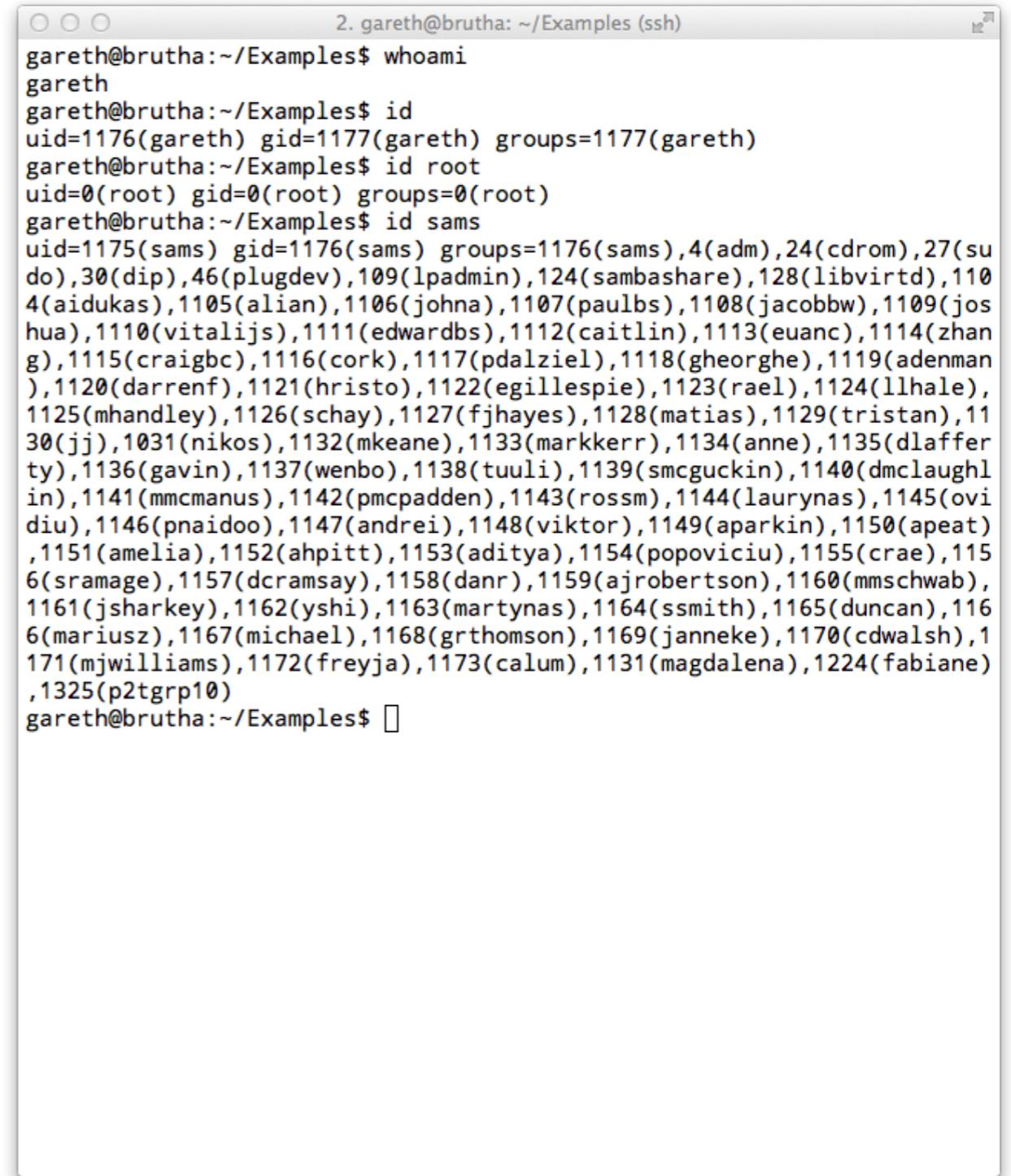


```
4. gareth@brutha: ~/Examples (ssh)
gareth@brutha:~$ find ./ -name loop.*
./Lab04/loop.c
gareth@brutha:~$ find ./ -name loop*
./Lab04/loop2.c
./Lab04/loop.c
gareth@brutha:~/Examples/
gareth@brutha:~/Examples$ find ./ -mtime -1
./
./b
./a
./dir1
./dir1/dir3
./dir1/dir3/d
./dir1/dir3/e
./dir1/dir2
./c
gareth@brutha:~/Examples$ locate loop.c
/home/gareth/Lab04/loop.c
/usr/lib/ocaml/toploop.cmi
/usr/share/doc/python2.7/examples/Demo/embed/loop.c
gareth@brutha:~/Examples$ whereis ls
ls: /bin/ls /usr/share/man/man1/ls.1.gz
gareth@brutha:~/Examples$ 
```

- The Filesystem
- Navigating the filesystem
- Users & Permissions
- Working with files
- Archives

Users & Groups

- Linux is a multi-user operating system.
- Each User has a unique ID (**uid**) and is associated with a set of groups. Each group also has a unique id (**gid**)
- What a user can do is restricted by the rights associated with each user or group.
- **whoami** - tells you what your username is.
- **id** - tells you about your id and group info.
- Linux systems have the concept of a “root” (or super) user who can do anything on the system
- This right can also be delegated to a specific user, often referred to **sudoer** (super user do).



A screenshot of a terminal window titled "2. gareth@brutha: ~/Examples (ssh)". The window displays the output of several commands:

```
gareth@brutha:~/Examples$ whoami
gareth
gareth@brutha:~/Examples$ id
uid=1176(gareth) gid=1177(gareth) groups=1177(gareth)
gareth@brutha:~/Examples$ id root
uid=0(root) gid=0(root) groups=0(root)
gareth@brutha:~/Examples$ id sams
uid=1175(sams) gid=1176(sams) groups=1176(sams),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sambashare),128(libvirtd),1104(aidukas),1105(alian),1106(johna),1107(paulbs),1108(jacobbw),1109(joshua),1110(vitalijs),1111(edwardbs),1112(caitlin),1113(euanc),1114(zhang),1115(craigbc),1116(cork),1117(pdalziel),1118(gheorghe),1119(adenman),1120(darrenf),1121(hristo),1122(egillespie),1123(rael),1124(ljhale),1125(mhandley),1126(schay),1127(fjhayes),1128(matias),1129(tristan),1130(jj),1031(nikos),1132(mkeane),1133(markkerr),1134(anne),1135(dlafferty),1136(gavin),1137(wenbo),1138(tuuli),1139(smcguckin),1140(dmclaughlin),1141(mmcmanus),1142(pmcpadden),1143(rossm),1144(laurynas),1145(ovidiu),1146(pnaidoo),1147(andrei),1148(viktor),1149(aparkin),1150(apeat),1151(amelia),1152(ahpitt),1153(aditya),1154(popoviciu),1155(crae),1156(sramage),1157(dcramsay),1158(danr),1159(ajrobertson),1160(mmschwab),1161(jsharkey),1162(yshi),1163(martynas),1164(ssmith),1165(duncan),1166(mariusz),1167(michael),1168(grthomson),1169(janneke),1170(cdwalsh),1171(mjwilliams),1172(freyja),1173(calum),1131(magdalena),1224(fabiane),1325(p2tgrp10)
gareth@brutha:~/Examples$
```

Permissions

- All restrictions in a Linux system are handled at the file level.
- Each file or directory has the notion of ownership by a user or by a group.
- Additionally restrictions are placed on what can be done to a file.
- ls -l will show the contents of a directory in long format. This shows:
 - permissions
 - no. of links
 - owner
 - group owner
 - size of file in bytes
 - date modified
 - name

```
gareth@brutha:~/Examples$ ls  
a b c d dir1  
gareth@brutha:~/Examples$ ls -ltr  
total 4  
-rw-rw-r-- 1 gareth gareth 0 Jan 14 09:05 a  
-rw-rw-r-- 1 gareth gareth 0 Jan 14 09:05 b  
drwxrwxr-x 4 gareth gareth 4096 Jan 14 09:28 dir1  
-rw-rw-r-- 1 gareth gareth 0 Jan 14 13:59 c  
lrwxrwxrwx 1 gareth gareth 11 Jan 15 08:56 d -> dir1/dir3/d  
gareth@brutha:~/Examples$
```

```
gareth@brutha:~/Examples$ ls -ltr  
total 4  
-rw-rw-r-- 1 gareth gareth 0 Jan 14 09:05 a  
-rw-rw-r-- 1 gareth gareth 0 Jan 14 09:05 b  
drwxrwxr-x 4 gareth gareth 4096 Jan 14 09:28 dir1  
-rw-rw-r-- 1 gareth gareth 0 Jan 14 13:59 c  
lrwxrwxrwx 1 gareth gareth 11 Jan 15 08:56 d -> dir1/dir3/d
```

Permissions

Filetype → -rwxrw-rw-

User Group Other

- First part is filetype
 - **d** - directory
 - **l** - link
 - **-** - normal file
- Next is permissions, split into three parts
 - **User** permissions
 - **Group** permissions
 - **Other** permissions (everyone on the system)
- Permissions are
 - **r** - read access
 - **w** - write access
 - **x** - execute

chmod [who][op][what] filename

- Can change permissions of a file using the **chmod** command
- who can be:
 - **u** - user permissions
 - **g** - group permissions
 - **o** - other permissions
 - **a** - all permissions
- op can be:
 - **+** - grant permissions
 - **-** - remove permissions
- what is one of the three permission types (**r,w,x**)

- The Filesystem
- Navigating the filesystem
- Users & Permissions
- Working with files
- Archives

Common Tasks

- create a file or directory
 - move a file or directory
 - copy a file or directory
 - deleting a file or directory
 - look at the contents of a file
 - look at the beginning of a file or end of a file
 - find a line of text in a file

File Creation

- A file can be created using:
 - **touch** <filename>
- Multiple files can be created by giving a list:
 - **touch** {a,b,c}
- Directories are created using:
 - **mkdir** <filename>
- We can create multiple directories in the same was as files:
 - **mkdir** {dir1,dir2,dir3}
- We can create a set of nested directories using the **-p** option. This creates all the directories not present:
 - **mkdir -p** this/is/a/new/directory

```
2. gareth@brutha: ~/Examples2 (ssh)
gareth@brutha:~/Examples2$ ls
gareth@brutha:~/Examples2$ touch a
gareth@brutha:~/Examples2$ ls
a
gareth@brutha:~/Examples2$ touch {b,c,d}
gareth@brutha:~/Examples2$ ls
a b c d
gareth@brutha:~/Examples2$ ls -l
total 0
-rw-rw-r-- 1 gareth gareth 0 Jan 15 10:18 a
-rw-rw-r-- 1 gareth gareth 0 Jan 15 10:18 b
-rw-rw-r-- 1 gareth gareth 0 Jan 15 10:18 c
-rw-rw-r-- 1 gareth gareth 0 Jan 15 10:18 d
gareth@brutha:~/Examples2$ mkdir dir2
gareth@brutha:~/Examples2$ ls
a b c d dir2
gareth@brutha:~/Examples2$ mkdir {dir1,dir3}
gareth@brutha:~/Examples2$ ls
a b c d dir1 dir2 dir3
gareth@brutha:~/Examples2$ mkdir -p this/is/a/new/directory
gareth@brutha:~/Examples2$ tree
.
├── a
├── b
├── c
├── d
└── dir1
    ├── dir2
    └── dir3
    └── this
        └── is
            └── a
                └── new
                    └── directory
8 directories, 4 files
gareth@brutha:~/Examples2$
```

Moving a File

- A file can be moved by:
 - **mv <src> <destination>**
- Where <src> and <destination> can be either absolute or relative paths.
- Multiple files can be moved by giving a list:
 - **mv {a,b,c,d} dir1**
- Or by using a file glob:
 - **mv dir1/* .**
- The move command can also be used to rename a file:
 - **mv a this_was_a**

```
2. gareth@brutha: ~/Examples2 (ssh)
gareth@brutha:~/Examples2$ tree
.
├── a
├── b
├── c
└── d
└── dir1

1 directory, 4 files
gareth@brutha:~/Examples2$ mv a dir1/
gareth@brutha:~/Examples2$ tree
.
├── b
├── c
└── d
└── dir1
    └── a

1 directory, 4 files
gareth@brutha:~/Examples2$ mv {b,c,d} dir1/
gareth@brutha:~/Examples2$ tree
.
└── dir1
    ├── a
    ├── b
    ├── c
    └── d

1 directory, 4 files
gareth@brutha:~/Examples2$ mv dir1/* .
gareth@brutha:~/Examples2$ tree
.
├── a
├── b
├── c
└── d
└── dir1

1 directory, 4 files
gareth@brutha:~/Examples2$ 
```

Copying a File

- A file can be copied by:
 - **cp** <src> <destination>
- Where <src> and <destination> can be either absolute or relative paths.
- Multiple files can be copied by giving a list or a glob:
 - **cp** {a,b,c,d} dir1
 - **cp** dir1/* .
- Copying directories is a little different, you need to tell copy that you want to recursively copy all the files, so:
 - **cp -r** dir1 dir2

```
2. gareth@brutha: ~/Examples2 (ssh)
.
├── a
└── b
    └── dir1

1 directory, 2 files
gareth@brutha:~/Examples2$ cp a dir1/
gareth@brutha:~/Examples2$ tree
.
├── a
└── b
    └── dir1
        └── a

1 directory, 3 files
gareth@brutha:~/Examples2$ cp {a,b} dir1/
gareth@brutha:~/Examples2$ tree
.
├── a
└── b
    └── dir1
        ├── a
        └── b

1 directory, 4 files
gareth@brutha:~/Examples2$ cp dir1 dir2
cp: omitting directory `dir1'
gareth@brutha:~/Examples2$ cp -r dir1 dir2
gareth@brutha:~/Examples2$ tree
.
├── a
└── b
    └── dir1
        ├── a
        └── b
    └── dir2
        ├── a
        └── b

2 directories, 6 files
gareth@brutha:~/Examples2$ 
```

Deleting a File

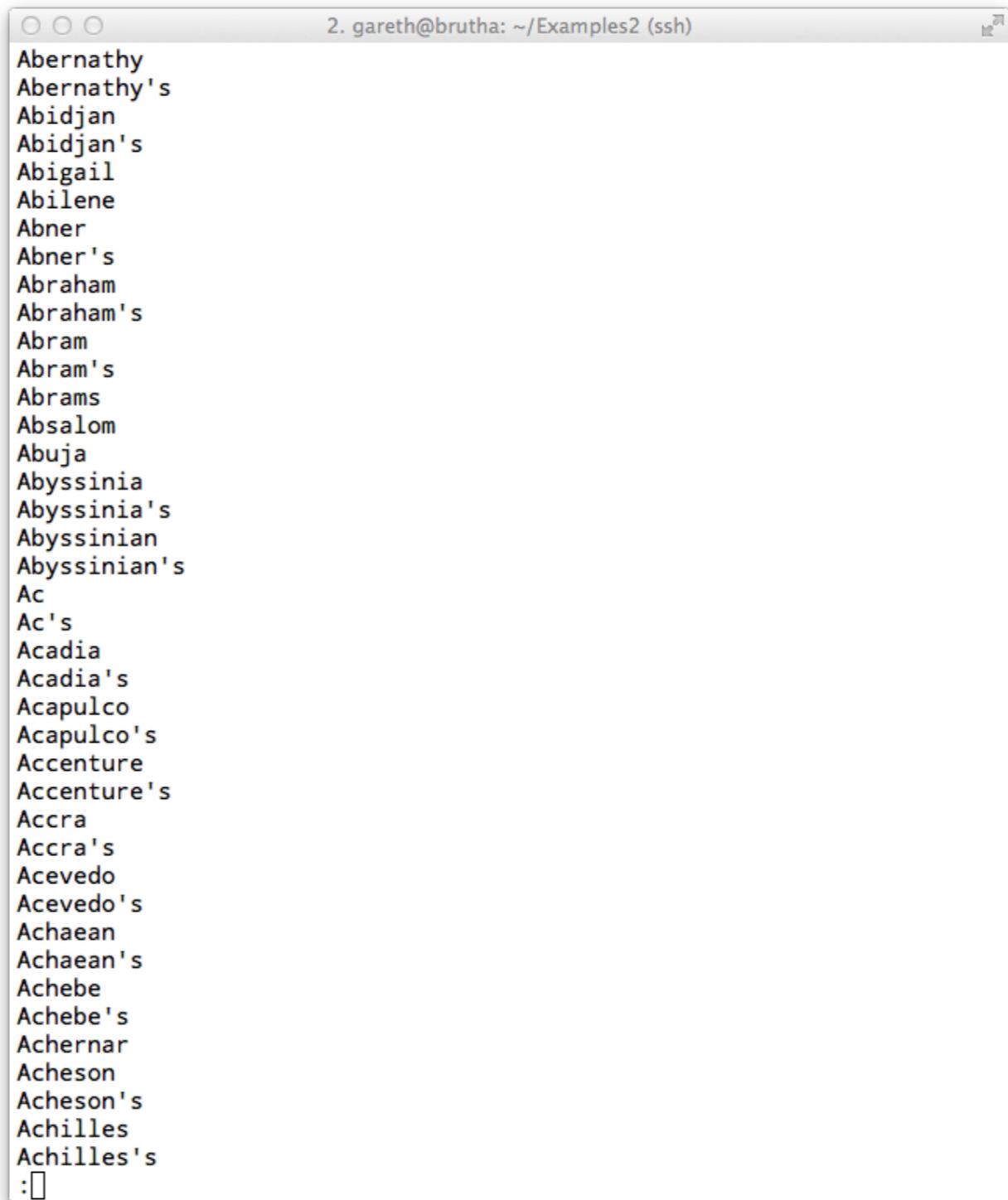
- A file can be deleted or removed by:
 - **rm <filename>**
- Where **<filename>** can be either an absolute or relative path.
- Multiple files can be removed by giving a list or a glob:
 - **rm {a,b,c,d}**
 - **rm dir1/***
- Like copying removing directories is a little different, you need to tell rm that you want to recursively delete all the files, so:
 - **rm -r dir1**
- Some files are also write protected (-r--r--r--) and you may have to remove with force:
 - **rm -f <filename>**
 - **rm -rf dir1**

```
2. gareth@brutha: ~/Examples2 (ssh)
gareth@brutha:~/Examples2$ ls
a b c d dir1
gareth@brutha:~/Examples2$ rm a
gareth@brutha:~/Examples2$ ls
b c d dir1
gareth@brutha:~/Examples2$ rm {b,c,d}
gareth@brutha:~/Examples2$ ls
dir1
gareth@brutha:~/Examples2$ rm dir1
rm: cannot remove `dir1': Is a directory
gareth@brutha:~/Examples2$ rm -r dir1/
gareth@brutha:~/Examples2$ ls
gareth@brutha:~/Examples2$ touch a
gareth@brutha:~/Examples2$ chmod -w a
gareth@brutha:~/Examples2$ ls -l
total 0
-r--r--r-- 1 gareth gareth 0 Jan 15 11:13 a
gareth@brutha:~/Examples2$ rm a
rm: remove write-protected regular empty file `a'? n
gareth@brutha:~/Examples2$ rm -f a
gareth@brutha:~/Examples2$ ls
gareth@brutha:~/Examples2$ 
```

CAUTION: When a file is removed it is gone, there is no wastebasket.

Looking at Files

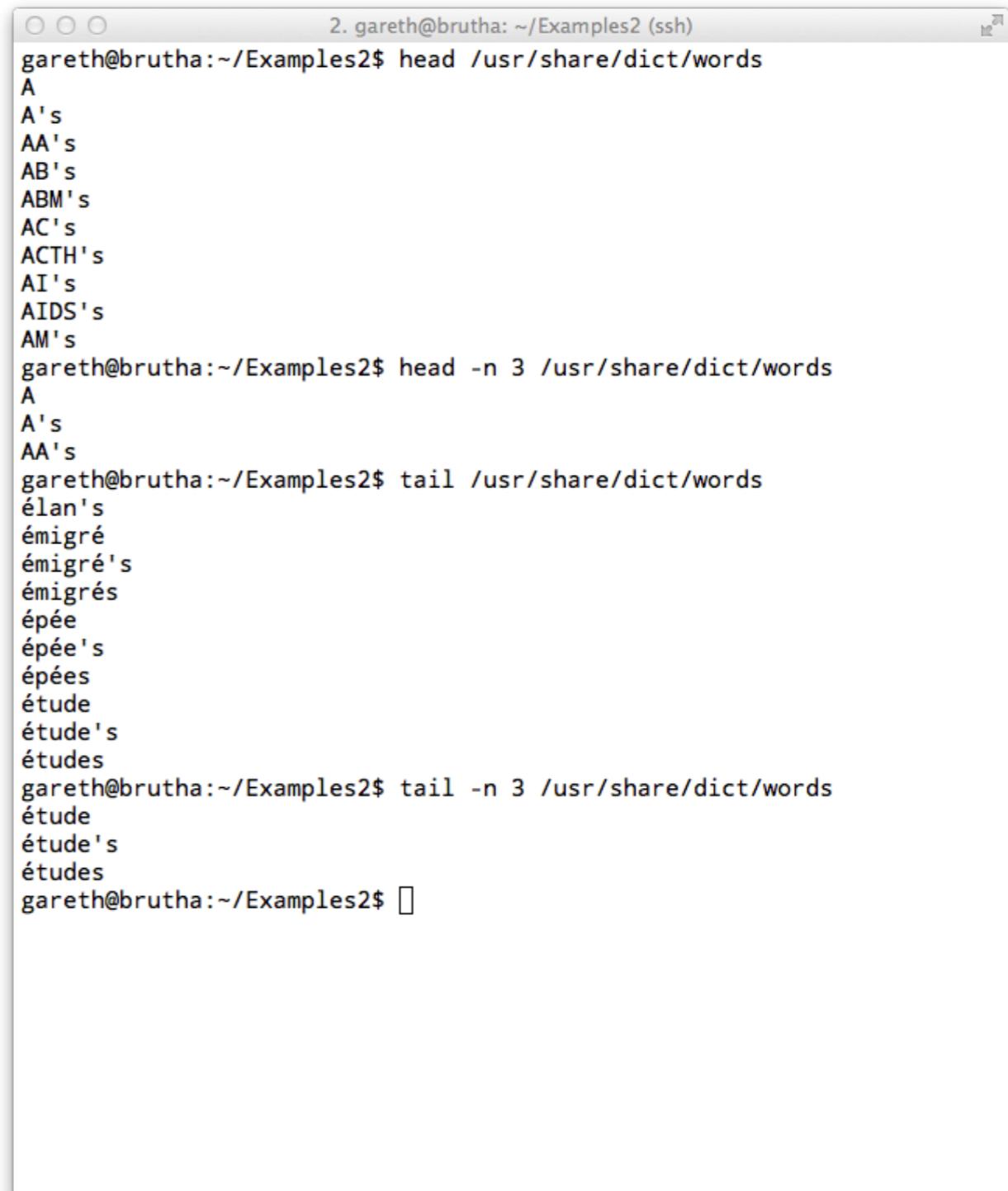
- They type of file can be found by using:
 - **file** <filename>
- The contents of file can be written to the terminal by using:
 - **cat** <filename>
- cat shows the whole contents of the file, we can see one page at a time using less or more:
 - **less** <filename>
 - **more** <filename>
- Of note, more will load the whole file into memory while less will only access the amount it needs to display.



A screenshot of a terminal window titled "2. gareth@brutha: ~/Examples2 (ssh)". The window displays a list of names and their possessives, likely from a file named "names.txt". The names listed are: Abernathy, Abernathy's, Abidjan, Abidjan's, Abigail, Abilene, Abner, Abner's, Abraham, Abraham's, Abram, Abram's, Abrams, Absalom, Abuja, Abyssinia, Abyssinia's, Abyssinian, Abyssinian's, Ac, Ac's, Acadia, Acadia's, Acapulco, Acapulco's, Accenture, Accenture's, Accra, Accra's, Acevedo, Acevedo's, Achaean, Achaean's, Achebe, Achebe's, Achernar, Acheson, Acheson's, Achilles, Achilles's, and a final colon followed by a blank line. The text is black on a white background, and the terminal has a standard OS X-style interface with a title bar and scroll bars.

Looking at bits of files

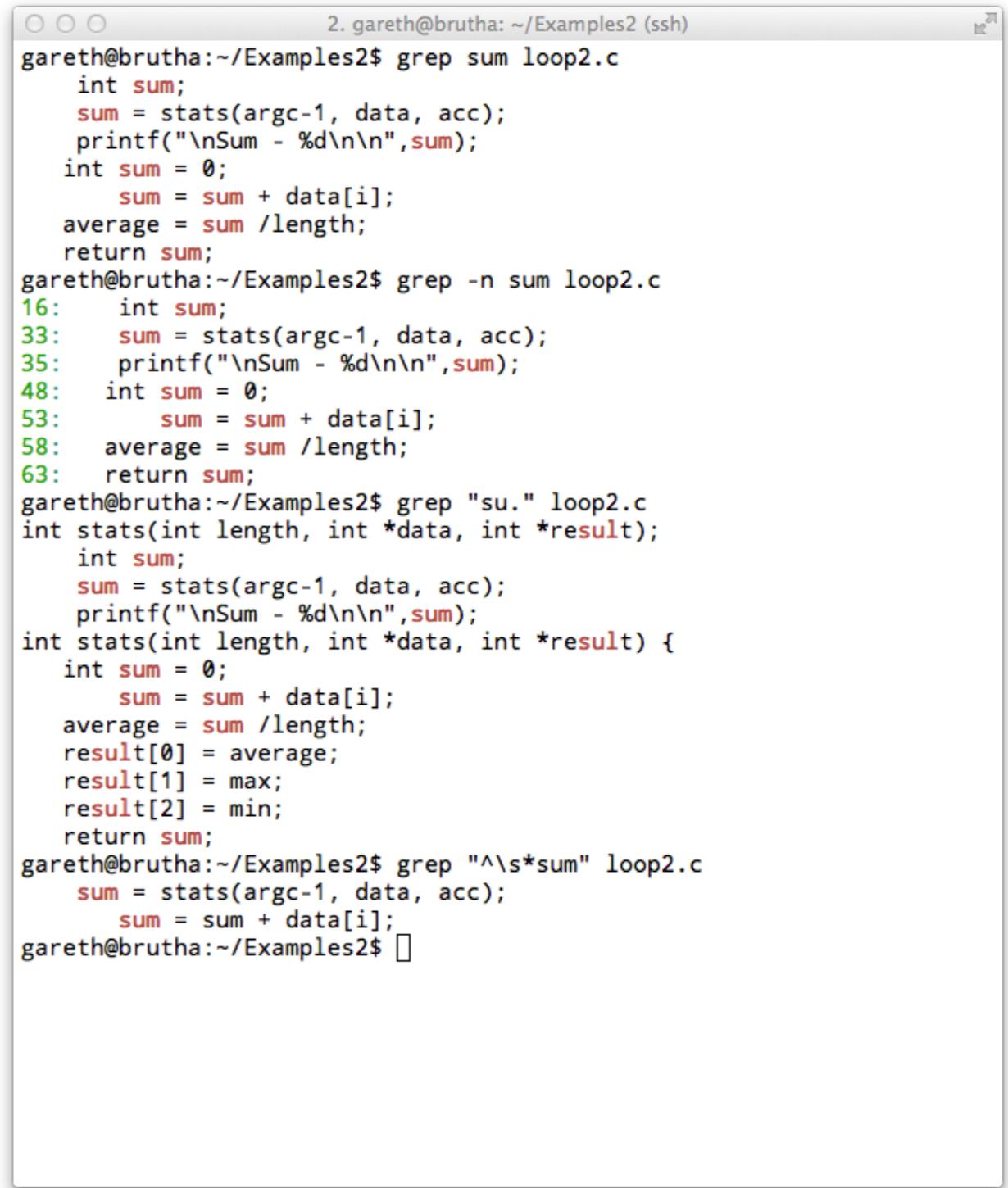
- For very large files (especially things like log file) you may only want to look at the start or end of the file.
- We can do this by using:
 - **head <filename>**
- This will print out the first ten lines of a file. If we want to print more or less than that we can specify, so:
 - **head -n 3 <filename>**
- As with head, we can also look at the end of a file with tail:
 - **tail <filename>**
 - **tail -n 3 <filename>**
- Tail can also be used to watch the end of a file and print out anything that is appended. This can be useful for watching things like logs:
 - **tail -f <filename>**



```
2. gareth@brutha: ~/Examples2 (ssh)
gareth@brutha:~/Examples2$ head /usr/share/dict/words
A
A's
AA's
AB's
ABM's
AC's
ACTH's
AI's
AIDS's
AM's
gareth@brutha:~/Examples2$ head -n 3 /usr/share/dict/words
A
A's
AA's
gareth@brutha:~/Examples2$ tail /usr/share/dict/words
élan's
émigré
émigré's
émigrés
épée
épée's
épées
étude
étude's
études
gareth@brutha:~/Examples2$ tail -n 3 /usr/share/dict/words
étude
étude's
études
gareth@brutha:~/Examples2$ 
```

Searching in a File

- Often you'd like to search for a particular word or phrase in a text file (or a particular variable name in a piece of C code).
- We can do this by using the grep command:
 - **grep <expression> <filename>**
- grep takes something called a “regular expression” and searches for it through a file.
- To search for the word “sum” in a C program called loop.c, we would write
 - **grep “sum” loop.c**
- You can get the line numbers for each line that matches by using the -n option:
 - **grep -n “sum” loop.c**
- You can search for more complicated things by using patterns:
 - * - match 0 to as many of the preceding character
 - . - match exactly one character
 - [a-z] - match any lowercase character
 - [0-9] - match any digit
 - ^ - match the start of the line
 - \$ - match the end of the line

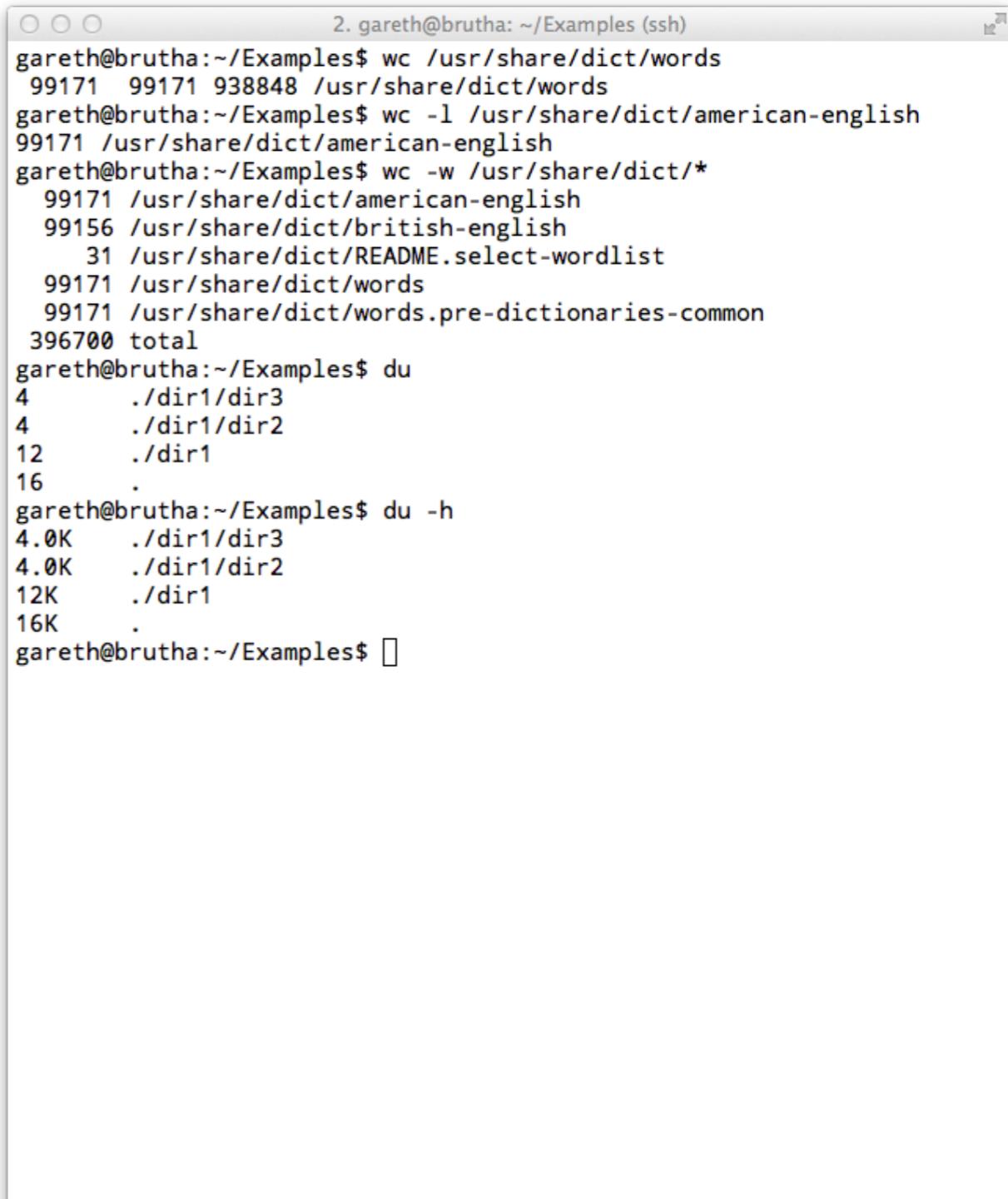


The screenshot shows a terminal window with the following session:

```
2. gareth@brutha: ~/Examples2 (ssh)
gareth@brutha:~/Examples2$ grep sum loop2.c
    int sum;
    sum = stats(argc-1, data, acc);
    printf("\nSum - %d\n\n",sum);
    int sum = 0;
    sum = sum + data[i];
    average = sum /length;
    return sum;
gareth@brutha:~/Examples2$ grep -n sum loop2.c
16:     int sum;
33:     sum = stats(argc-1, data, acc);
35:     printf("\nSum - %d\n\n",sum);
48:     int sum = 0;
53:     sum = sum + data[i];
58:     average = sum /length;
63:     return sum;
gareth@brutha:~/Examples2$ grep "su." loop2.c
int stats(int length, int *data, int *result);
    int sum;
    sum = stats(argc-1, data, acc);
    printf("\nSum - %d\n\n",sum);
int stats(int length, int *data, int *result) {
    int sum = 0;
    sum = sum + data[i];
    average = sum /length;
    result[0] = average;
    result[1] = max;
    result[2] = min;
    return sum;
gareth@brutha:~/Examples2$ grep "\^s*sum" loop2.c
    sum = stats(argc-1, data, acc);
    sum = sum + data[i];
gareth@brutha:~/Examples2$ 
```

Other Useful Commands

- Sometimes you want to find out how many words to lines are in a file. To do this use **wc**:
 - **wc <filename>**
 - **wc -l <filename>** (for lines)
 - **wc -w <filename>** (for words)
- You can sort the contents of a file using the **sort** command:
 - **sort <filename>**
- If you want to find out how much storage your files are taking up you can do:
 - **du <directory>**
 - **du -h <directory>**



```
2. gareth@brutha: ~/Examples (ssh)
gareth@brutha:~/Examples$ wc /usr/share/dict/words
 99171  99171 938848 /usr/share/dict/words
gareth@brutha:~/Examples$ wc -l /usr/share/dict/american-english
99171 /usr/share/dict/american-english
gareth@brutha:~/Examples$ wc -w /usr/share/dict/*
 99171 /usr/share/dict/american-english
 99156 /usr/share/dict/british-english
      31 /usr/share/dict/README.select-wordlist
 99171 /usr/share/dict/words
 99171 /usr/share/dict/words.pre-dictionaries-common
396700 total
gareth@brutha:~/Examples$ du
4      ./dir1/dir3
4      ./dir1/dir2
12     ./dir1
16     .
gareth@brutha:~/Examples$ du -h
4.0K   ./dir1/dir3
4.0K   ./dir1/dir2
12K    ./dir1
16K    .
gareth@brutha:~/Examples$ 
```

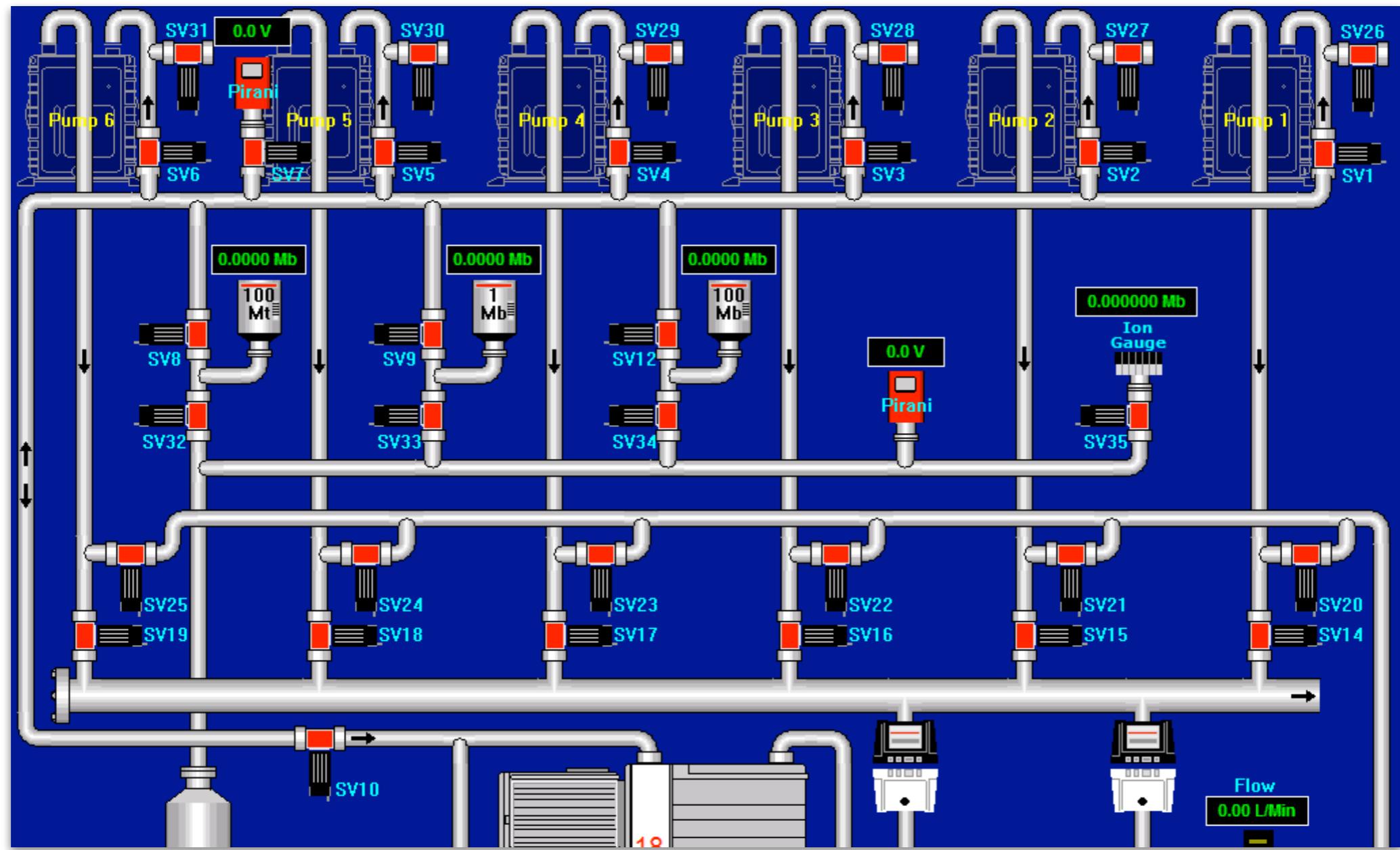
- The Filesystem
- Navigating the filesystem
- Users & Permissions
- Working with files
- Archives

Tarballs

- Sometimes you'd like to package up all your files in a single file.
- Similar to zipping or raring a file on Windows or OS X
- On Linux the default way for doing this is to create a “tarball”
- **tar** (tape archive) is a tool that was used for writing files to tape.
- It can be used to put all the files in a directory tree into a single file and preserve the paths.
 - **tar cvf <filename> <files>**
 - **tar tvf <filename>**
 - **tar xvf <filename>**
- You may also want to compress the archive, which can be done via tar or using gzip:
 - **tar zcvf <filename> <files>**
 - **gzip blob.tar**

```
2. gareth@brutha: ~/Examples (ssh)
gareth@brutha:~/Examples$ ls
a b c d dir1
gareth@brutha:~/Examples$ tar cvf blob.tar *
a
b
c
d
dir1/
dir1/dir3/
dir1/dir3/d
dir1/dir3/e
dir1/dir2/
gareth@brutha:~/Examples$ tar tvf blob.tar
-r--r--r-- gareth/gareth    0 2015-01-14 09:05 a
-rw-rw-r-- gareth/gareth    0 2015-01-14 09:05 b
-rw-rw-r-- gareth/gareth    0 2015-01-14 13:59 c
lrwxrwxrwx gareth/gareth    0 2015-01-15 08:56 d -> dir1/dir3/d
drwxrwxr-x gareth/gareth    0 2015-01-14 09:28 dir1/
drwxrwxr-x gareth/gareth    0 2015-01-14 09:06 dir1/dir3/
-rw-rw-r-- gareth/gareth    0 2015-01-14 09:05 dir1/dir3/d
-rw-rw-r-- gareth/gareth    0 2015-01-14 09:06 dir1/dir3/e
drwxrwxr-x gareth/gareth    0 2015-01-14 09:05 dir1/dir2/
gareth@brutha:~/Examples$ gzip blob.tar
gareth@brutha:~/Examples$ ls -l
total 8
-r--r--r-- 1 gareth gareth    0 Jan 14 09:05 a
-rw-rw-r-- 1 gareth gareth    0 Jan 14 09:05 b
-rw-rw-r-- 1 gareth gareth  262 Jan 15 12:07 blob.tar.gz
-rw-rw-r-- 1 gareth gareth    0 Jan 14 13:59 c
lrwxrwxrwx 1 gareth gareth    11 Jan 15 08:56 d -> dir1/dir3/d
drwxrwxr-x 4 gareth gareth 4096 Jan 14 09:28 dir1
gareth@brutha:~/Examples$ 
```

- The Filesystem
- Navigating the filesystem
- Users & Permissions
- Working with files
- Archives



Processes

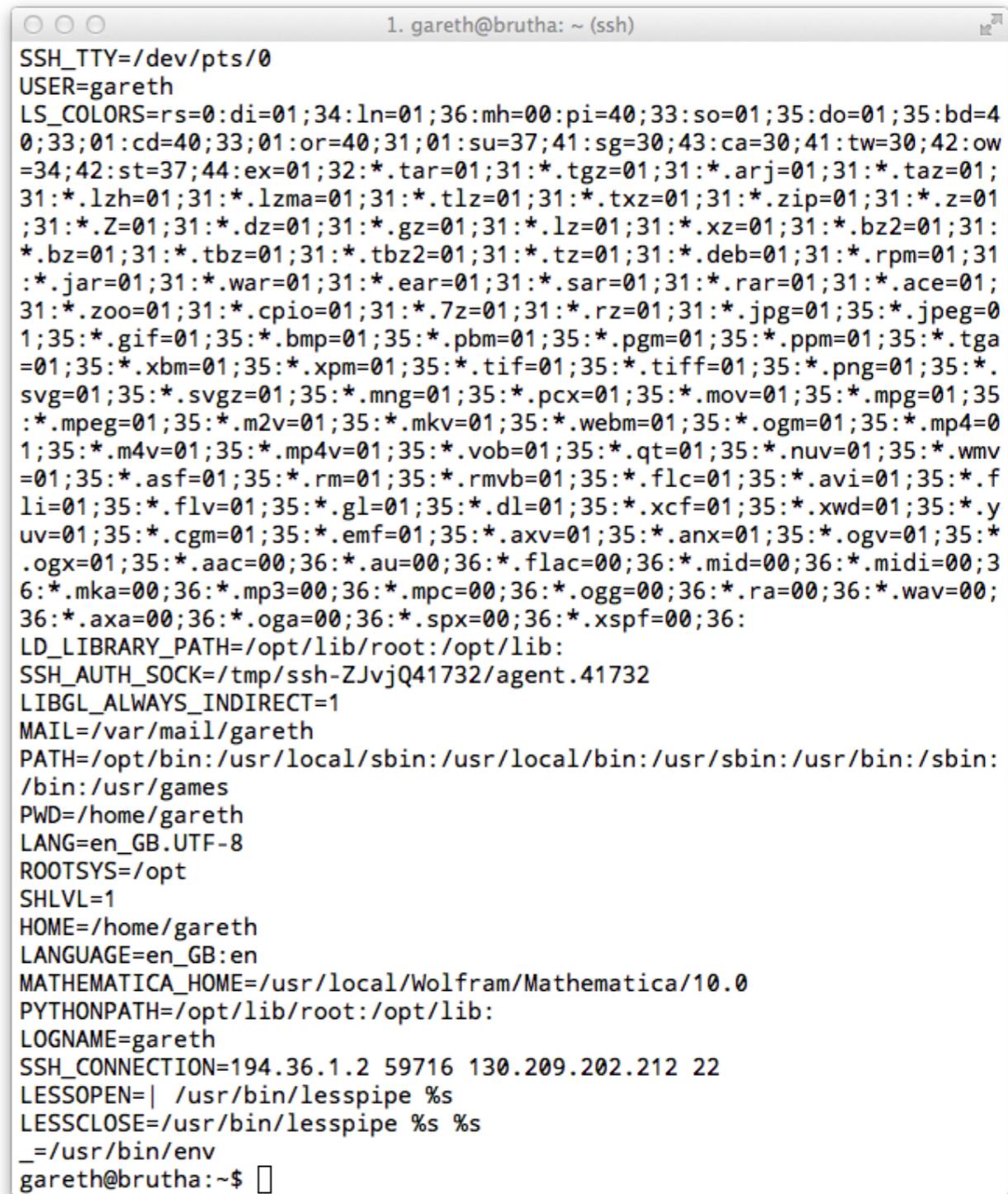
Dr. Gareth Roy (x6439)
gareth.roy@glasgow.ac.uk

- Variables and the Shell
- Processes
- Working with Processes
- IO Streams
- Pipes and Redirections

- Variables and the Shell
- Processes
- Working with Processes
- IO Streams
- Pipes and Redirections

Variables

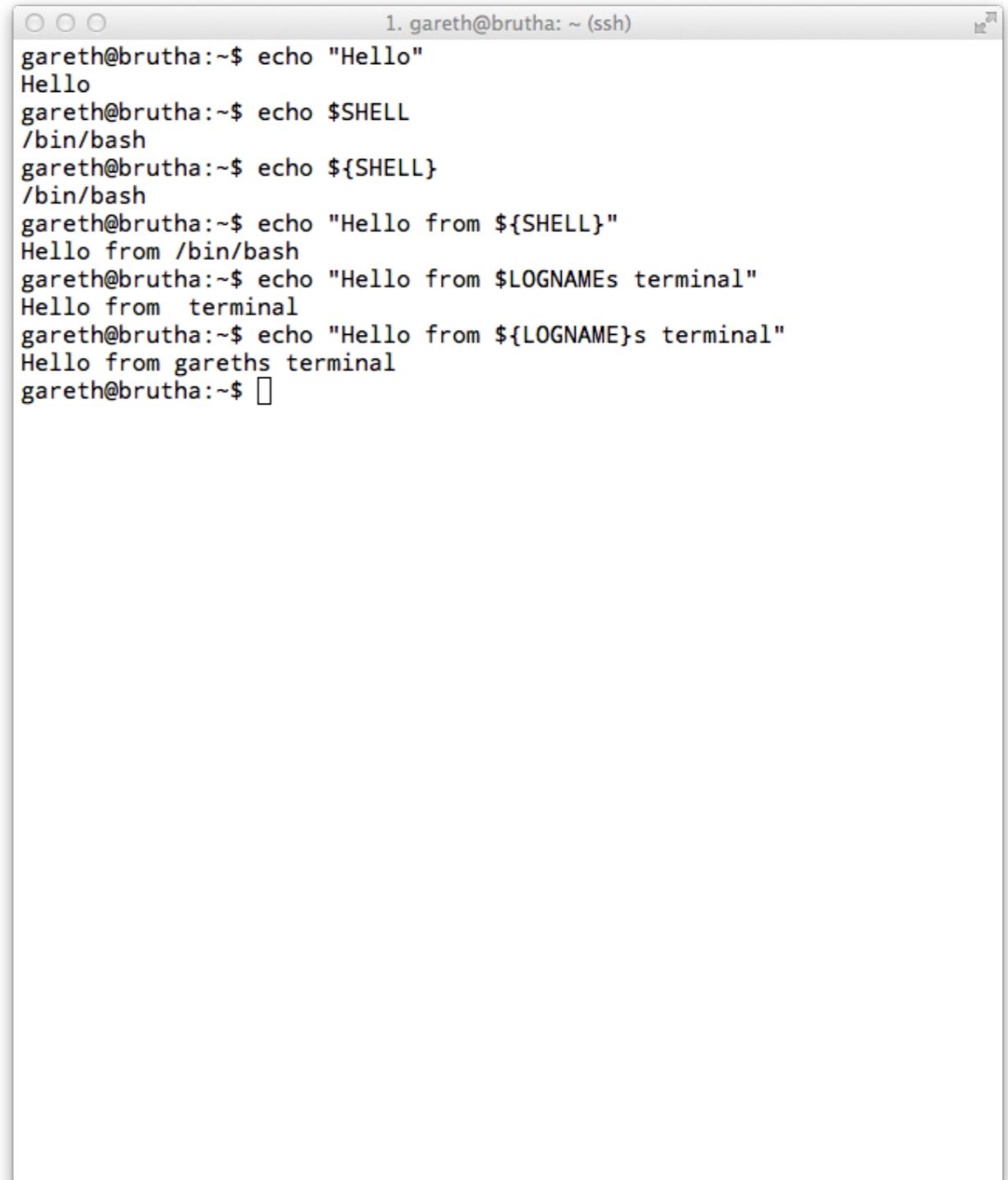
- The “shell” stores parameters in variables.
These are referred to as environment variables.
- In these examples we assume our shell is **bash**.
- To assign a value to a bash variable on the command line we would do:
 - **export SOME_VAR=“some value”**
 - **Note:** there are no spaces between the “=” and the left and right hand side of the assignment and variables are case sensitive.
- **bash** and most shells use these variables to store configuration information.
- By convention most environment variable names are all in uppercase.
- You can see all the variables set in a running shell by using the **env** or **export** commands.



A screenshot of a terminal window titled "1. gareth@brutha: ~ (ssh)". The window displays a list of environment variables. The variables include SSH_TTY=/dev/pts/0, USER=gareth, LS_COLORS (a long string of color codes), LD_LIBRARY_PATH=/opt/lib/root:/opt/lib, SSH_AUTH_SOCK=/tmp/ssh-ZJvjQ41732/agent.41732, LIBGL_ALWAYS_INDIRECT=1, MAIL=/var/mail/gareth, PATH=/opt/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games, PWD=/home/gareth, LANG=en_GB.UTF-8, ROOTSYS=/opt, SHLVL=1, HOME=/home/gareth, LANGUAGE=en_GB:en, MATHEMATICA_HOME=/usr/local/Wolfram/Mathematica/10.0, PYTHONPATH=/opt/lib/root:/opt/lib, LOGNAME=gareth, SSH_CONNECTION=194.36.1.2 59716 130.209.202.212 22, LESSOPEN=| /usr/bin/lesspipe %s, LESSCLOSE=/usr/bin/lesspipe %s %s _=/usr/bin/env, and gareth@brutha:~\$.

Variables

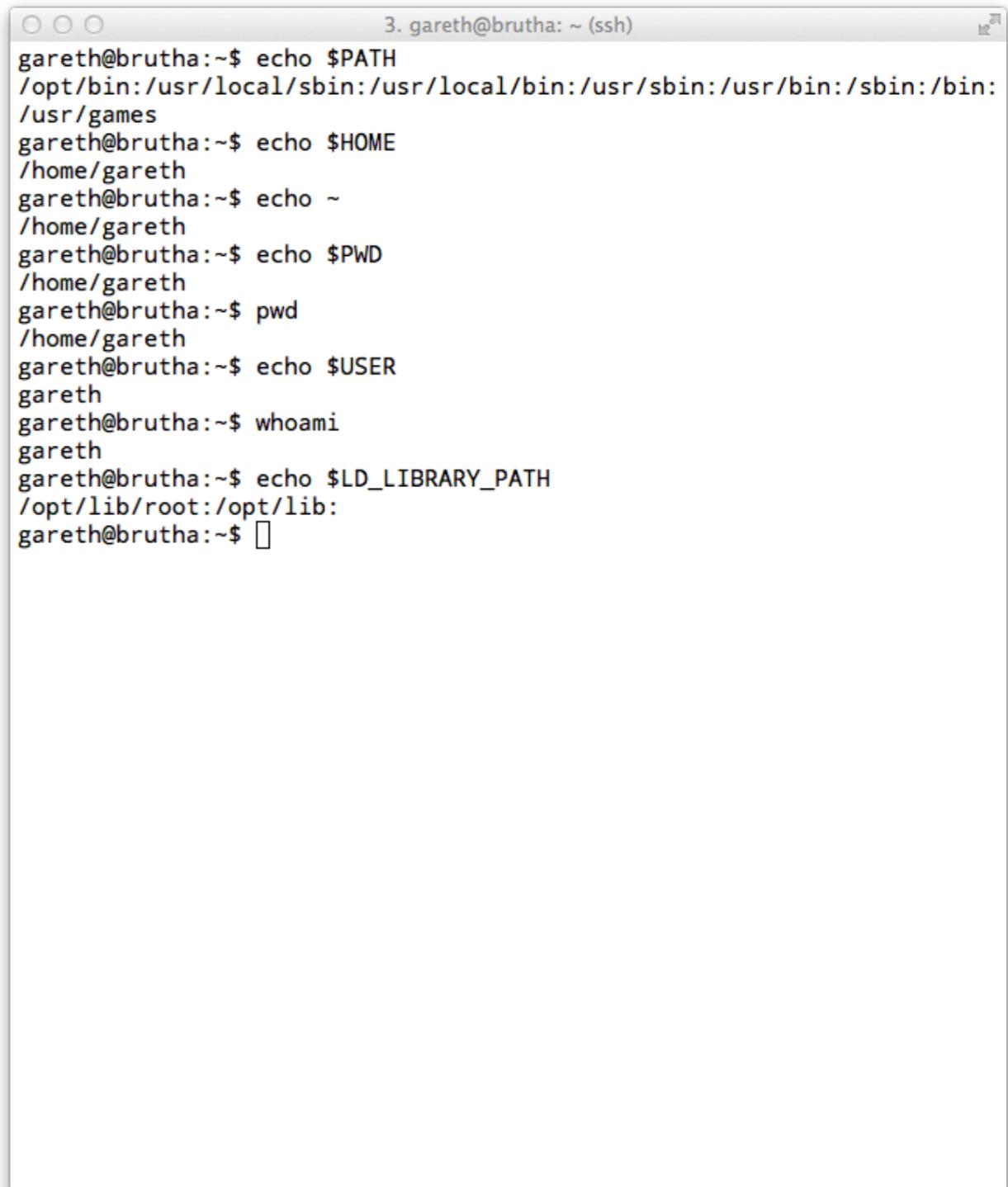
- To access a specific variable we can use the **echo** command.
- **echo** writes to the screen anything passed to it:
 - **echo** “hello”
- To access a specific shell variable we can echo its contents by adding **\$** to its name:
 - **echo** \$SHELL
- It's better practice to reference our variables using \${} notation. The above would then be
 - **echo** \${SHELL}
- This allows us to embed the variable expansion inside another value without worrying about the variable name:
 - **echo** “Hello from \${SHELL}”



```
gareth@brutha:~$ echo "Hello"
Hello
gareth@brutha:~$ echo $SHELL
/bin/bash
gareth@brutha:~$ echo ${SHELL}
/bin/bash
gareth@brutha:~$ echo "Hello from ${SHELL}"
Hello from /bin/bash
gareth@brutha:~$ echo "Hello from $LOGNAMEs terminal"
Hello from terminal
gareth@brutha:~$ echo "Hello from ${LOGNAME}s terminal"
Hello from gareths terminal
gareth@brutha:~$ 
```

Important Variables

- There are some shell variables that it is useful to know about
- **\$PATH** - the places to look for a particular command.
- **\$HOME** - the users home directory (can be accessed using the ~ symbol)
- **\$PWD** - the current working directory (also available via **pwd** command)
- **\$USER** - the username (also available via the **whoami** command)
- **\$LD_LIBRARY_PATH** - the paths to search for library files when building C code.



```
gareth@brutha:~$ echo $PATH  
/opt/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:  
/usr/games  
gareth@brutha:~$ echo $HOME  
/home/gareth  
gareth@brutha:~$ echo ~  
/home/gareth  
gareth@brutha:~$ echo $PWD  
/home/gareth  
gareth@brutha:~$ pwd  
/home/gareth  
gareth@brutha:~$ echo $USER  
gareth  
gareth@brutha:~$ whoami  
gareth  
gareth@brutha:~$ echo $LD_LIBRARY_PATH  
/opt/lib/root:/opt/lib:  
gareth@brutha:~$ 
```

Aliases

- Often typed commands can be “aliased” to much simpler commands.
- For instance if we would like a long directory listing, showing all hidden files and a classifier:

▶ **ls -laF**

- This could be aliased to

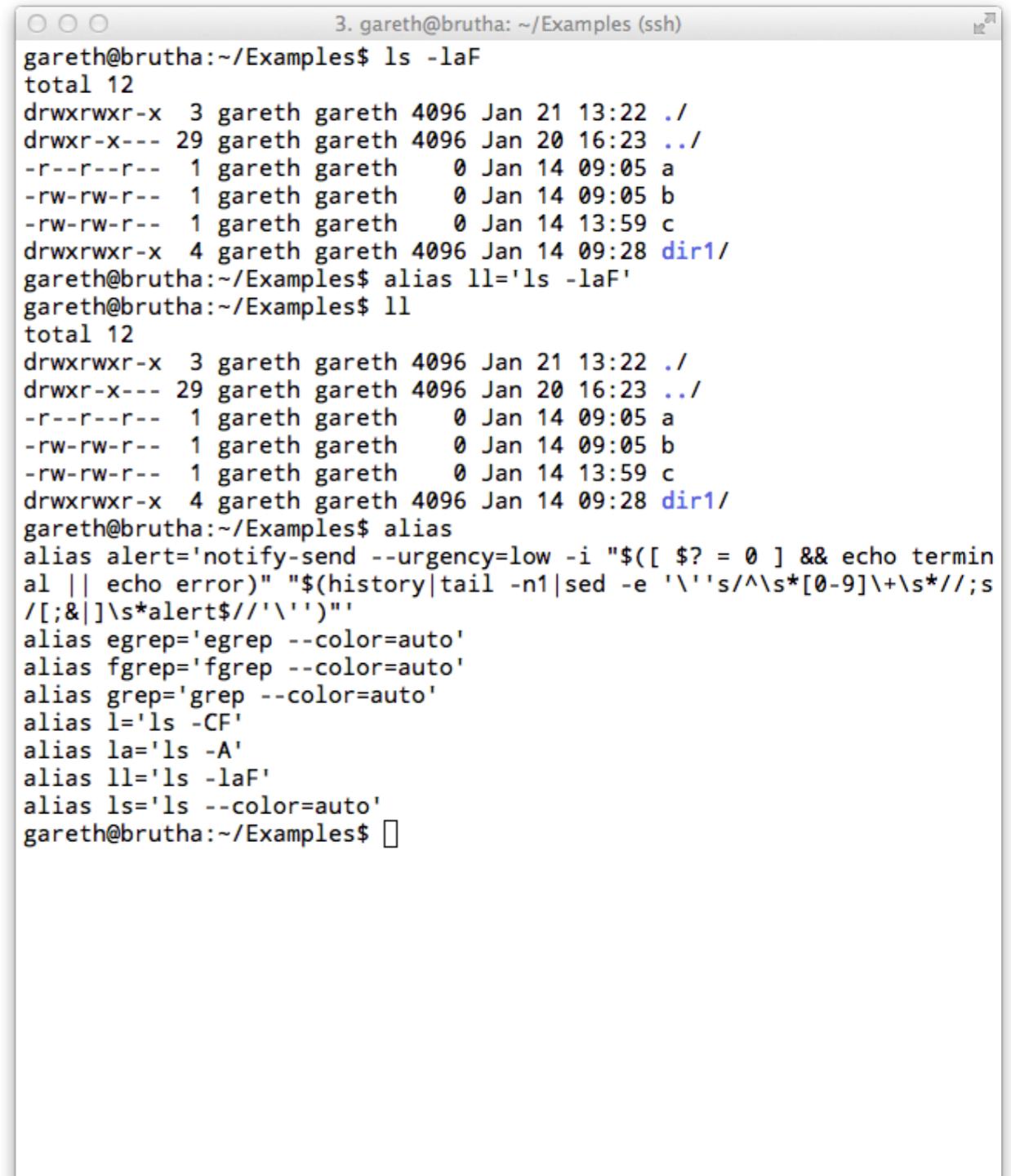
▶ **alias ll='ls -laF'**

- At the command line we would then use:

▶ **ll**

- You can see a list of all aliases set in your shell by running the alias command without any arguments

▶ **alias**



The screenshot shows a terminal window titled "gareth@brutha: ~/Examples (ssh)". The user has run several commands to demonstrate aliasing:

```
gareth@brutha:~/Examples$ ls -laF
total 12
drwxrwxr-x  3 gareth gareth 4096 Jan 21 13:22 .
drwxr-x--- 29 gareth gareth 4096 Jan 20 16:23 ..
-r--r--r--  1 gareth gareth    0 Jan 14 09:05 a
-rw-rw-r--  1 gareth gareth    0 Jan 14 09:05 b
-rw-rw-r--  1 gareth gareth    0 Jan 14 13:59 c
drwxrwxr-x  4 gareth gareth 4096 Jan 14 09:28 dir1/
gareth@brutha:~/Examples$ alias ll='ls -laF'
gareth@brutha:~/Examples$ ll
total 12
drwxrwxr-x  3 gareth gareth 4096 Jan 21 13:22 .
drwxr-x--- 29 gareth gareth 4096 Jan 20 16:23 ..
-r--r--r--  1 gareth gareth    0 Jan 14 09:05 a
-rw-rw-r--  1 gareth gareth    0 Jan 14 09:05 b
-rw-rw-r--  1 gareth gareth    0 Jan 14 13:59 c
drwxrwxr-x  4 gareth gareth 4096 Jan 14 09:28 dir1/
gareth@brutha:~/Examples$ alias
alias alert='notify-send --urgency=low -i "$( [ $? = 0 ] && echo terminal || echo error)" "$(history|tail -n1|sed -e '\''\''s/^\\s*[0-9]\\+\\s*//;s/[\;\\&]\\s*alert$\'\'\''\'\'")'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -laF'
alias ls='ls --color=auto'
gareth@brutha:~/Examples$
```

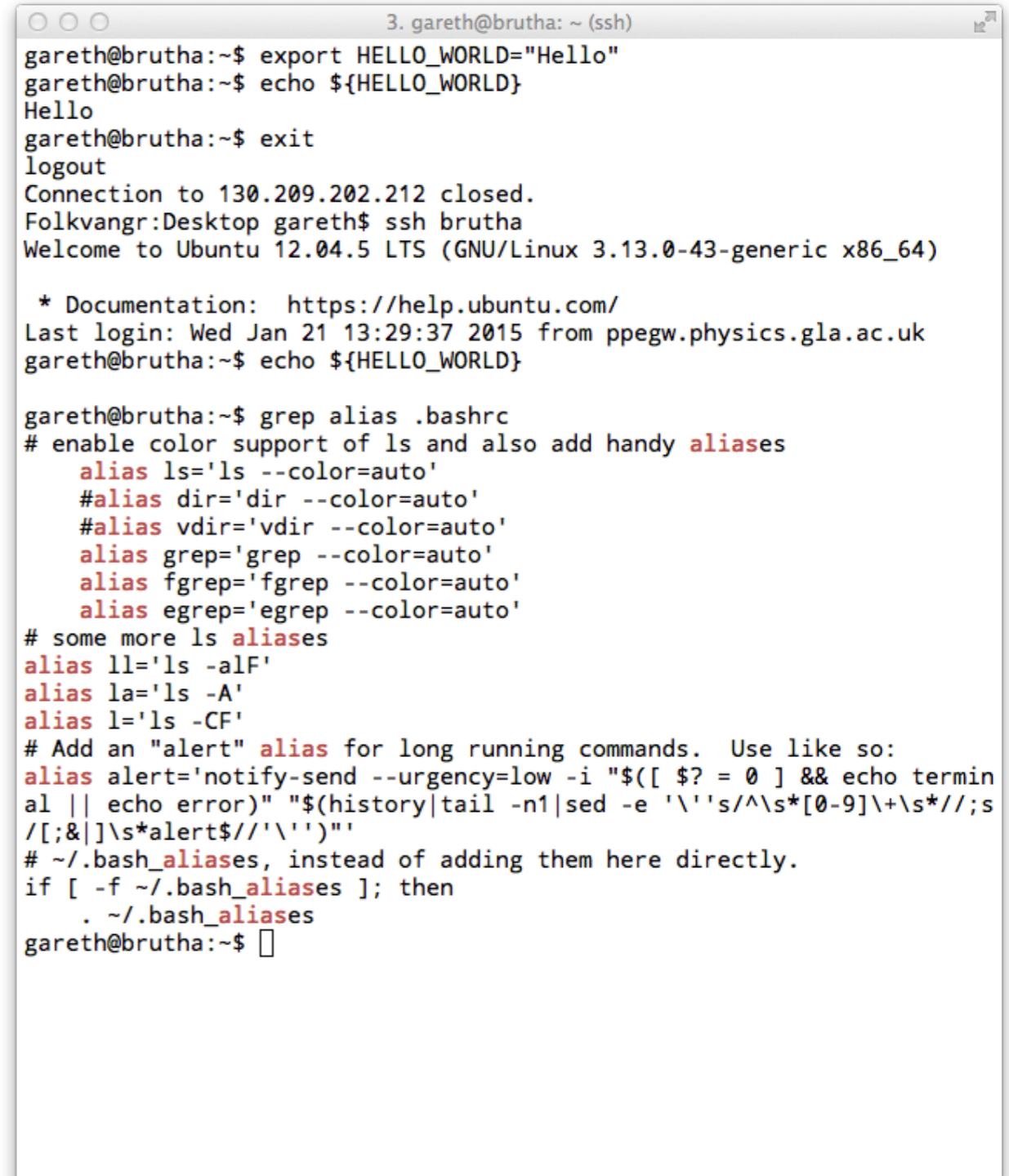
Persistent Variables

- All variables in **bash** shells are transient.
- Variables you set in one shell will not persist when you open a new one.
- When you open a new shell bash reads a number of configuration files to set its state.
- We can make variables persist by adding them to one of these configuration files.
- Bash has two configuration files found in your home directory

- **.bashrc**

- **.bash_profile**

- The difference is subtle and most times you'll want to put settings in **.bashrc** but be aware that **.bash_profile** exists.
- If you edit your .bashrc file you can load the new variables into the current shell by
 - **source .bashrc**



```
3. gareth@brutha: ~ (ssh)
gareth@brutha:~$ export HELLO_WORLD="Hello"
gareth@brutha:~$ echo ${HELLO_WORLD}
Hello
gareth@brutha:~$ exit
logout
Connection to 130.209.202.212 closed.
Folkvangr:Desktop gareth$ ssh brutha
Welcome to Ubuntu 12.04.5 LTS (GNU/Linux 3.13.0-43-generic x86_64)

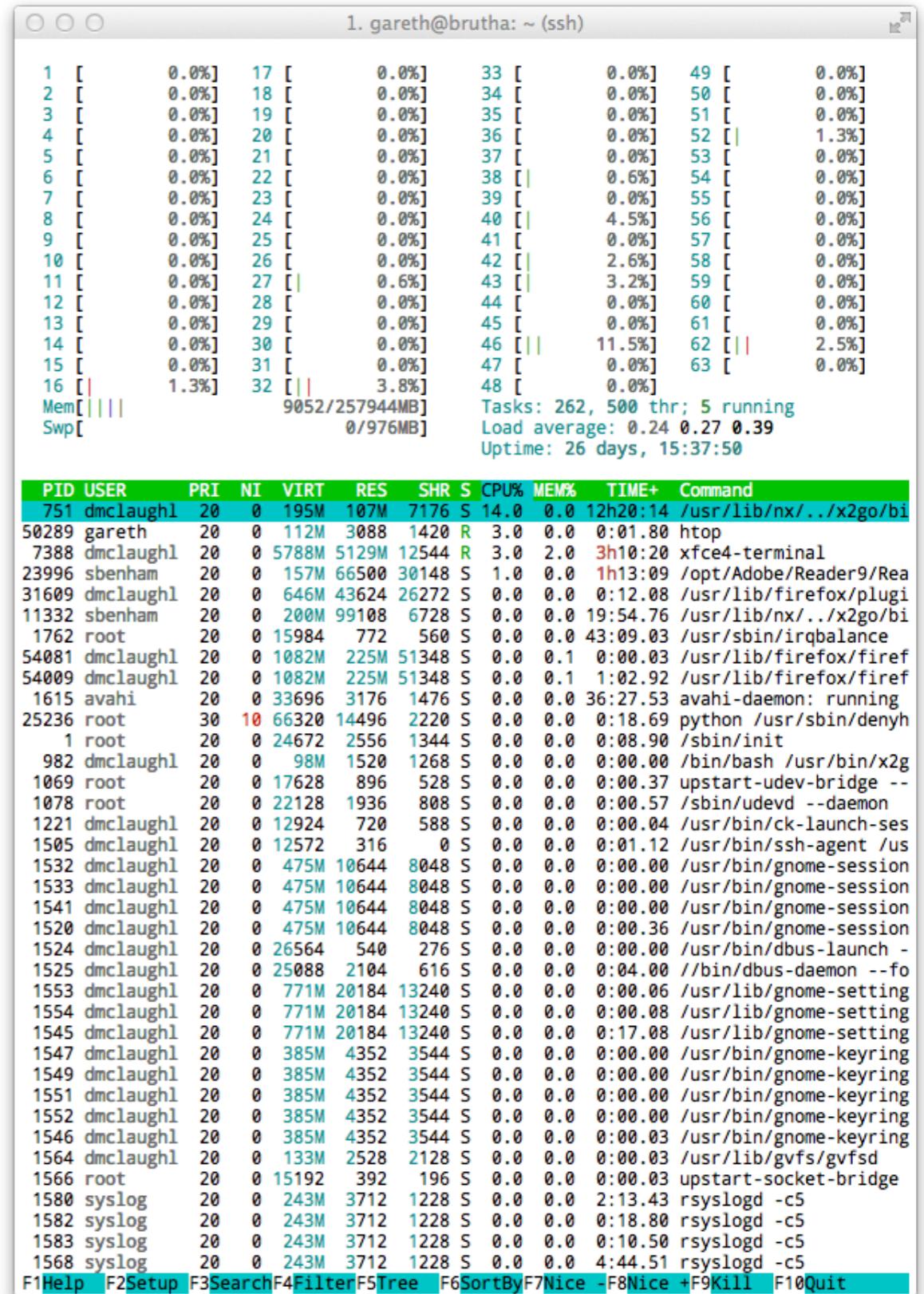
 * Documentation:  https://help.ubuntu.com/
Last login: Wed Jan 21 13:29:37 2015 from ppegw.physics.gla.ac.uk
gareth@brutha:~$ echo ${HELLO_WORLD}

gareth@brutha:~$ grep alias .bashrc
# enable color support of ls and also add handy aliases
alias ls='ls --color=auto'
#alias dir='dir --color=auto'
#alias vdir='vdir --color=auto'
alias grep='grep --color=auto'
alias fgrep='fgrep --color=auto'
alias egrep='egrep --color=auto'
# some more ls aliases
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'
# Add an "alert" alias for long running commands. Use like so:
alias alert='notify-send --urgency=low -i "$( [ $? = 0 ] && echo termin
al || echo error)" "$(history|tail -n1|sed -e '\''s/^\s*[0-9]+\s*//;s
/[\;]\s*alert$/'\''")'
# ~/.bash_aliases, instead of adding them here directly.
if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
gareth@brutha:~$ 
```

- Variables and the Shell
- Processes
- Working with Processes
- IO Streams
- Pipes and Redirections

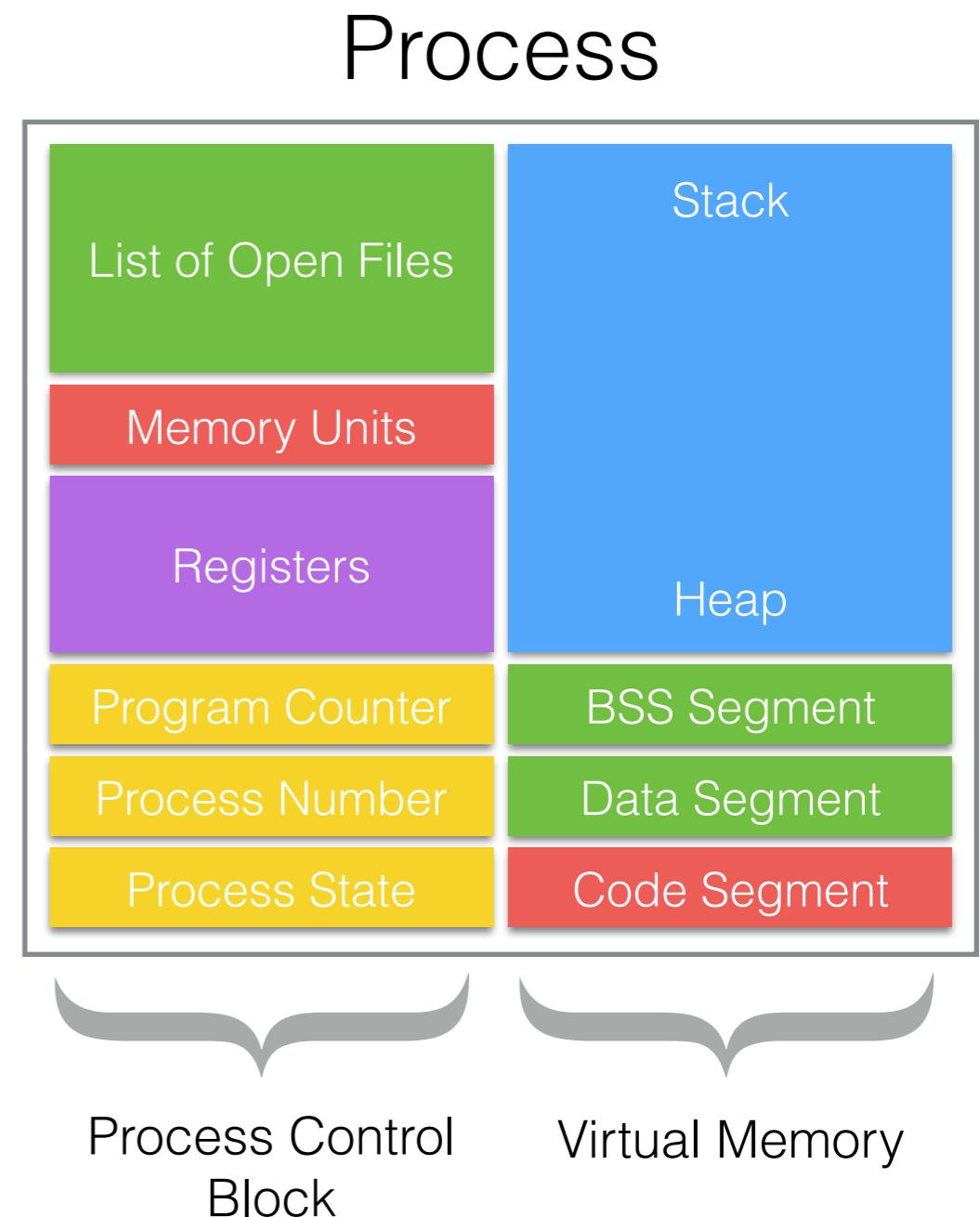
What is a Process?

- Each program that runs on a Linux system runs inside a process.
- Each process is self contained, even when it's for the same program.
- The Linux Kernel schedules CPU time for each process to run.
- Processes are created as children of other processes by a method called “**forking**”.
- **pstree** will show all the processes running on a system, and the relationship they have with each other.



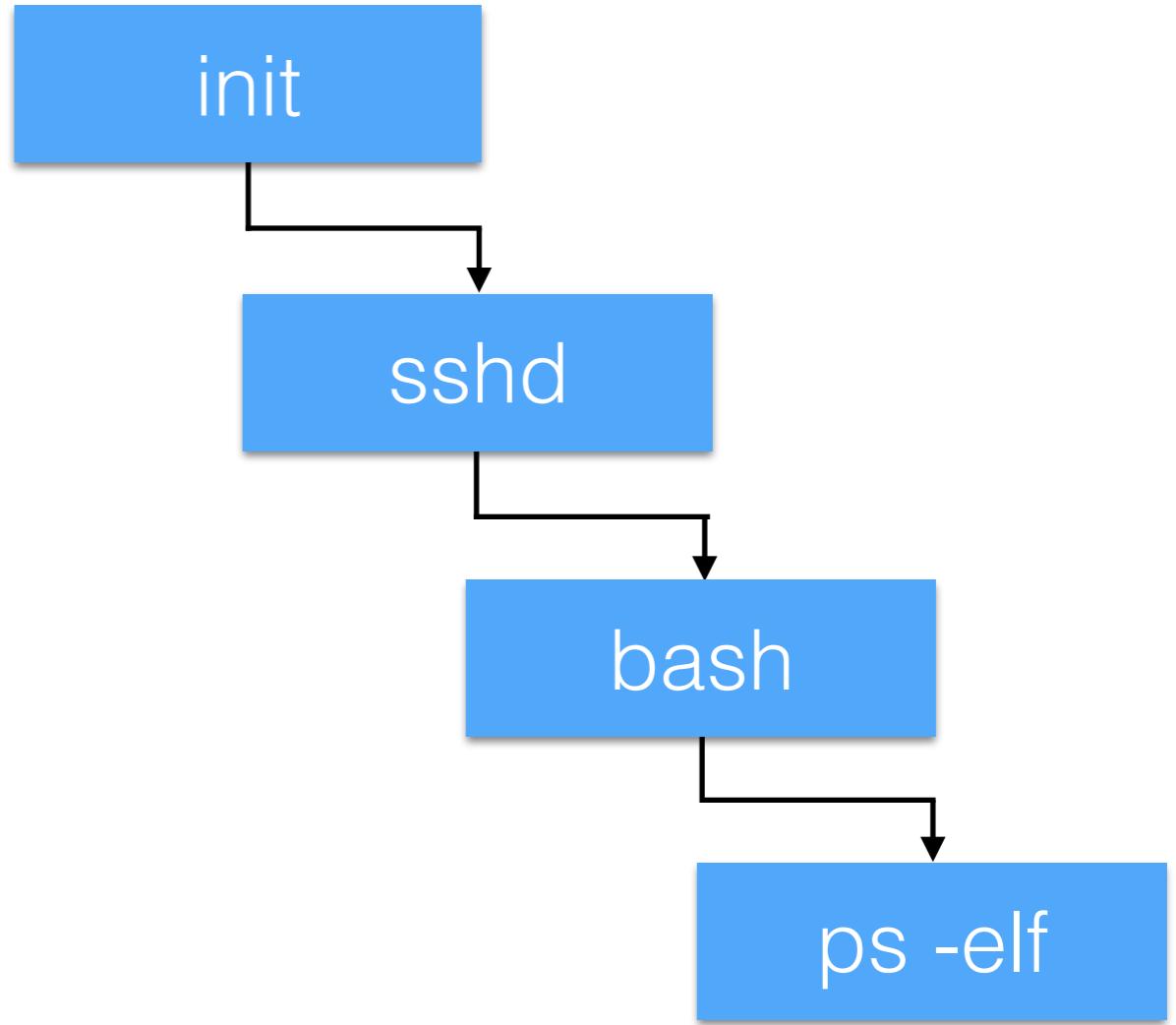
What's inside a Process?

- It contains:
 - a copy of the program.
 - a section of Virtual Memory assigned to the process (containing the Stack and Heap).
 - a link to any open files (file descriptors)
 - a snapshot of the processor state (including register contents)
 - attributes such as owner and permissions.
- Each process is isolated from every other process in the system.



Forking

- Processes are created by “**forking**”.
- When a process **forks** it creates a copy of itself and then starts the program it desires to run.
- This gives a parent child relationship between processes.
- The initial process is usually “**init**”



```
gareth@brutha:~$ ps -elf |grep gareth
4 S root    49740 38657  0  80  0 - 23088 poll_s 12:00 ?          00:00:00 sshd: gareth [priv]
5 S gareth  49755 49740  0  80  0 - 23088 poll_s 12:00 ?          00:00:00 sshd: gareth@pts/0
0 S gareth  49756 49755  3  80  0 - 29351 wait   12:00 pts/0      00:00:00 -bash
0 R gareth  49960 49756  0  80  0 - 26557 -       12:01 pts/0      00:00:00 ps -elf
0 S gareth  49961 49756  0  80  0 - 24365 pipe_w 12:01 pts/0      00:00:00 grep --color=auto gareth
gareth@brutha:~$ pstree 49755
sshd—bash—pstree
gareth@brutha:~$ ps auxwf |grep gareth
root    49740  0.0  0.0 92352 4068 ?          Ss   12:00  0:00  \_ sshd: gareth [priv]
gareth  49755  0.0  0.0 92352 1696 ?          S    12:00  0:00      \_ sshd: gareth@pts/0
gareth  49756  1.1  0.0 117404 10576 pts/0     Ss   12:00  0:00      \_ -bash
gareth  50071  0.0  0.0 107152 2140 pts/0     R+   12:01  0:00          \_ ps auxwf
gareth  50072  0.0  0.0 97460  920 pts/0      S+   12:01  0:00          \_ grep --color=auto gareth
gareth@brutha:~$
```

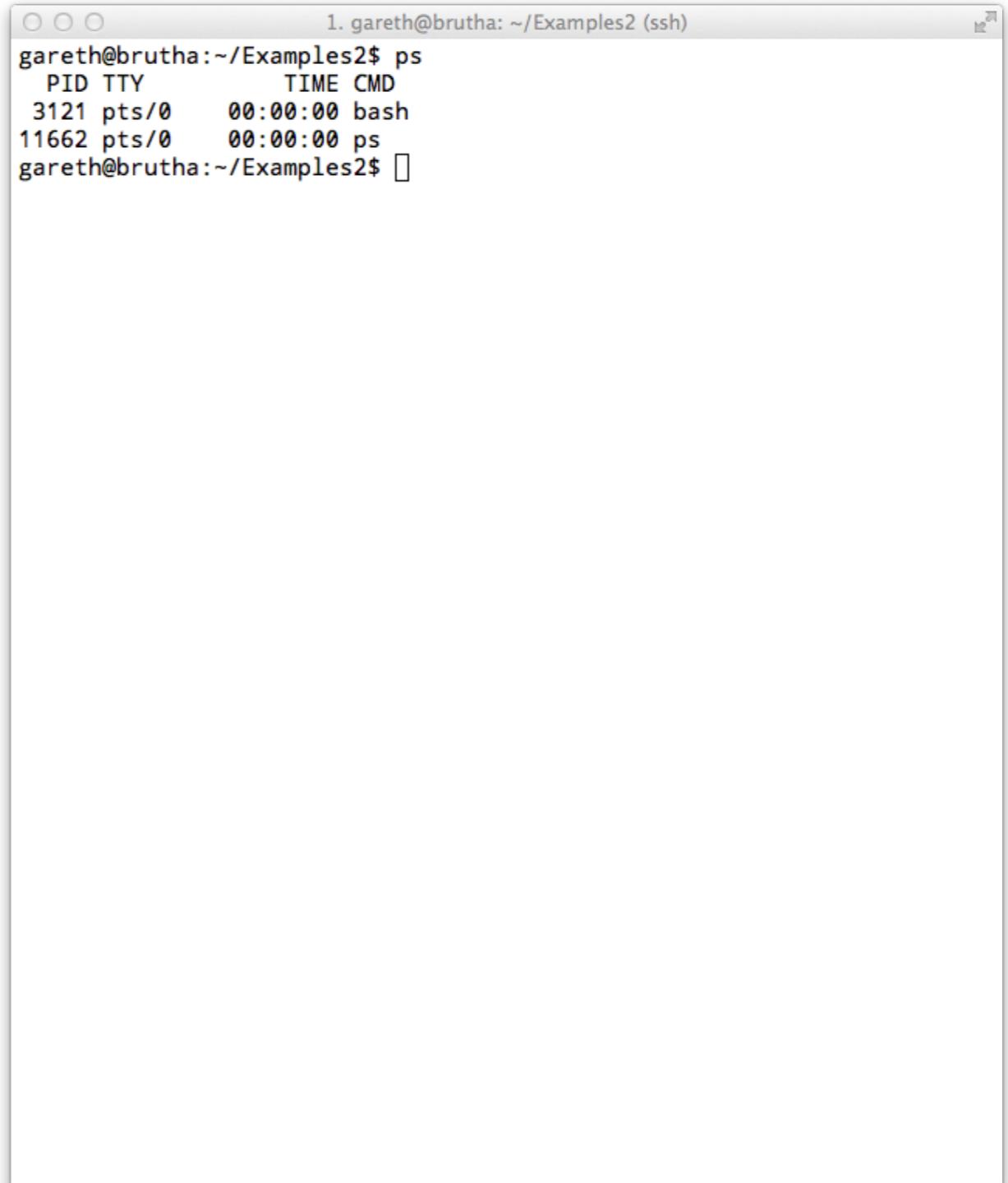
- Variables and the Shell
- Processes
- Working with Processes
- IO Streams
- Pipes and Redirections

Common Tasks

- view the processes running on the system.
 - start a new process in the background.
 - move a process to the background.
 - suspend a running process.
 - kill a process.

Seeing processes

- You'll often need to monitor the status of processes running on your system.
- You can get a list of processes running on your system by using **ps**
- **ps** with no arguments will show you only the processes running in your shell
- **ps** shows the process id (**pid**), how long the command has been running and the command that is run.
- **ps -e** (or **ps -ax**) will show you all processes running on the system.
- other useful filters are:
 - **ps -elf**
 - **ps -auxwf**



```
1. gareth@brutha: ~/Examples2 (ssh)
gareth@brutha:~/Examples2$ ps
 PID TTY      TIME CMD
 3121 pts/0    00:00:00 bash
11662 pts/0    00:00:00 ps
gareth@brutha:~/Examples2$ 
```

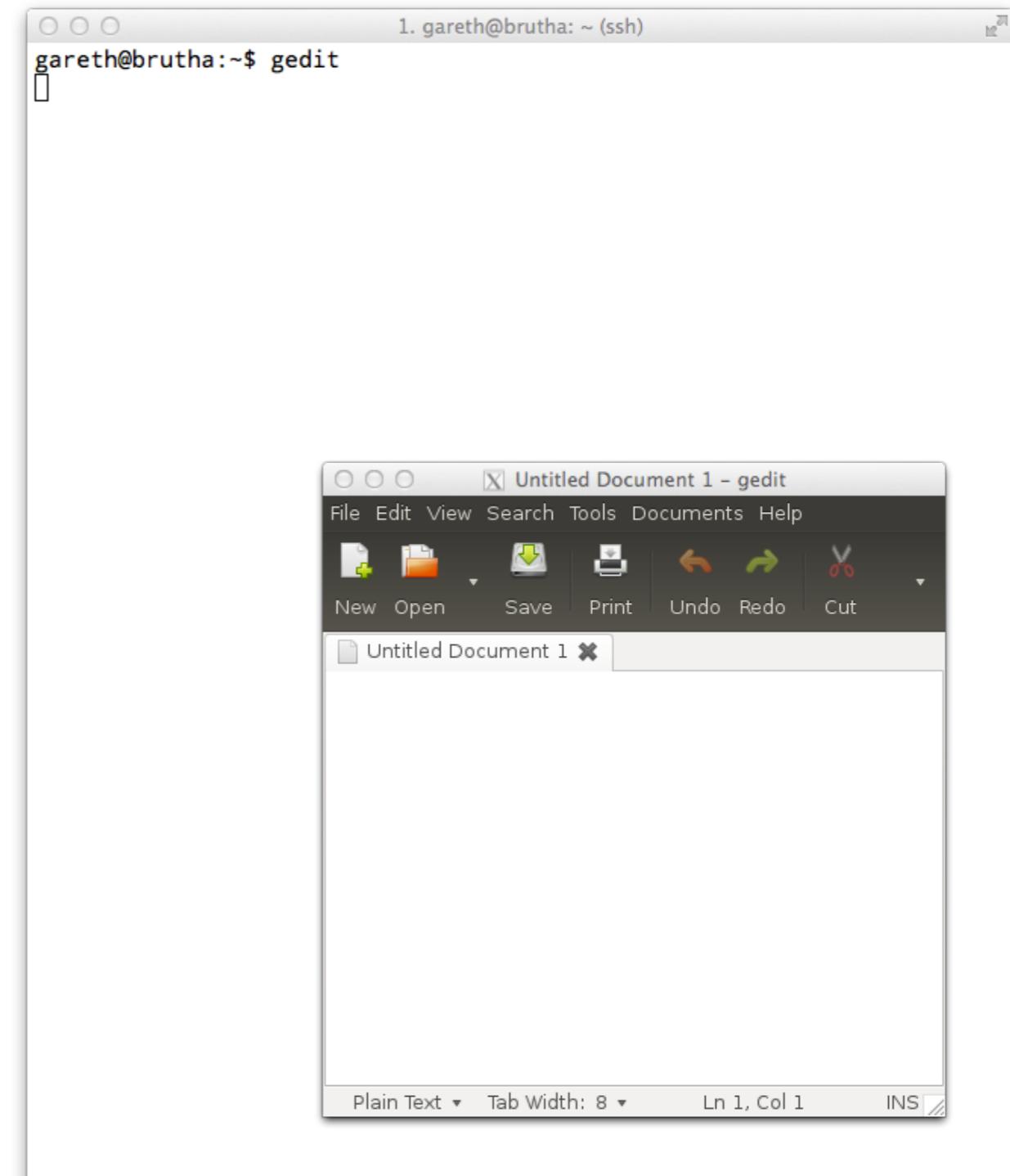
Other useful process tools

- If you want to see what processes are using particular resources you can use **top**.
- By default top sorts by **%CPU**.
- This can be changed by typing **F** and selecting a new sort field.
- To exit **top** type the **q** character.
- To see a hierarchical list of all the process we can use **pstree**.
- **pstree** shows the parent and children of each process in a graph. This lets us easily see the relationship between processes

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
13758	gareth	20	0	18272	2256	972	R	1	0.0	0:00.13	top
55841	dmclaugh	20	0	673m	70m	28m	S	1	0.0	49:17.05	plugin-co
15753	mdeans	20	0	3367m	199m	112m	S	1	0.1	37:36.07	chromium-
27165	mwood	20	0	662m	49m	25m	S	1	0.0	67:07.93	plugin-co
942	root	20	0	0	0	0	S	0	0.0	0:22.73	kworker/2
1576	mdeans	20	0	223m	122m	7164	S	0	0.0	9:17.35	x2goagent
1615	avahi	20	0	34664	4260	1476	S	0	0.0	49:48.17	avahi-dae
1762	root	20	0	15984	772	560	S	0	0.0	54:29.84	irqbalanc
2286	lightdm	20	0	794m	23m	13m	S	0	0.0	131:20.14	unity-gre
50591	dmclaugh	20	0	1172m	311m	53m	S	0	0.1	30:16.79	firefox
61378	martynas	20	0	646m	42m	25m	S	0	0.0	39:17.52	plugin-co
1	root	20	0	24672	2556	1344	S	0	0.0	0:15.47	init
2	root	20	0	0	0	0	S	0	0.0	0:00.84	kthreadd
3	root	20	0	0	0	0	S	0	0.0	0:08.19	ksoftirqd
4	root	20	0	0	0	0	S	0	0.0	0:00.00	kworker/0
5	root	0	-20	0	0	0	S	0	0.0	0:00.00	kworker/0
8	root	20	0	0	0	0	S	0	0.0	72:31.30	rcu_sched
9	root	20	0	0	0	0	S	0	0.0	6:38.74	rcuos/0
10	root	20	0	0	0	0	S	0	0.0	5:49.49	rcuos/1
11	root	20	0	0	0	0	S	0	0.0	2:01.68	rcuos/2
12	root	20	0	0	0	0	S	0	0.0	0:37.26	rcuos/3
13	root	20	0	0	0	0	S	0	0.0	1:50.76	rcuos/4
14	root	20	0	0	0	0	S	0	0.0	5:04.32	rcuos/5
15	root	20	0	0	0	0	S	0	0.0	1:53.96	rcuos/6
16	root	20	0	0	0	0	S	0	0.0	5:32.29	rcuos/7
17	root	20	0	0	0	0	S	0	0.0	0:54.64	rcuos/8
18	root	20	0	0	0	0	S	0	0.0	0:21.13	rcuos/9
19	root	20	0	0	0	0	S	0	0.0	0:38.66	rcuos/10
20	root	20	0	0	0	0	S	0	0.0	1:18.35	rcuos/11
21	root	20	0	0	0	0	S	0	0.0	0:43.65	rcuos/12
22	root	20	0	0	0	0	S	0	0.0	0:11.34	rcuos/13
23	root	20	0	0	0	0	S	0	0.0	0:44.64	rcuos/14
24	root	20	0	0	0	0	S	0	0.0	0:29.06	rcuos/15
25	root	20	0	0	0	0	S	0	0.0	0:16.16	rcuos/16

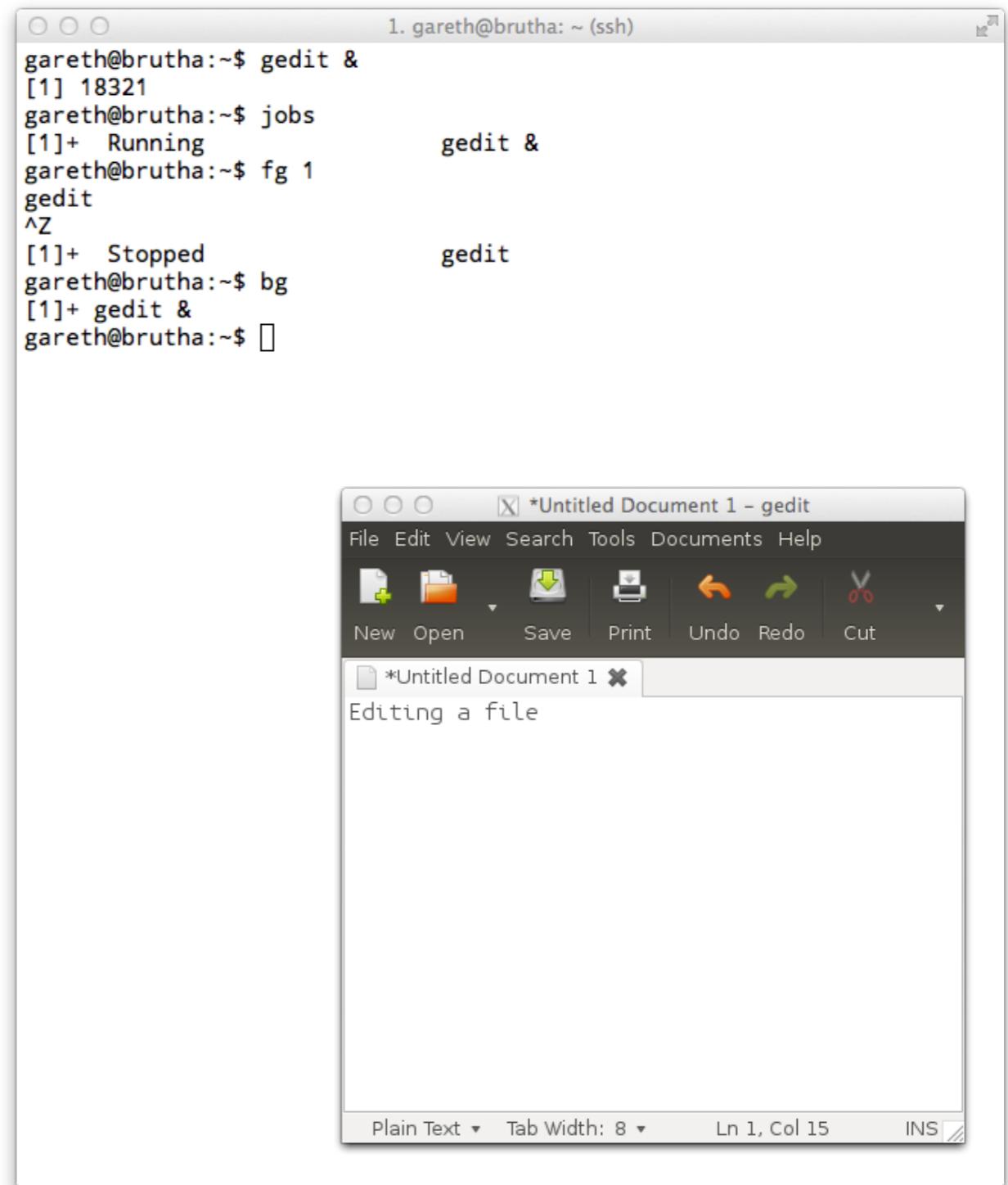
Foreground and Background

- Foreground jobs are processes that have interactive access to the command line.
- Background processes don't have access to the interactive command line but still run.
- Use ‘**&**’ to start a process in the background and return to the command line for the next instruction.
- This can be used to start a text editor but allow you to continue to use the terminal.
- Use ‘jobs’ to check what processes are running in the current terminal.



Job Control

- **ctrl-c** - will kill a job that is currently running in the foreground.
- **ctrl-z** - will suspend a job that's currently running in the foreground.
- A suspended job retains it's state but doesn't run or use CPU time.
- **fg <job id>** - will bring the specific job into the foreground.
- **bg <job id>** - will put a suspended job into the background



The terminal window shows the following session:

```
gareth@brutha:~$ gedit &
[1] 18321
gareth@brutha:~$ jobs
[1]+ Running gedit &
gareth@brutha:~$ fg 1
gedit
^Z
[1]+ Stopped gedit
gareth@brutha:~$ bg
[1]+ gedit &
gareth@brutha:~$ 
```

Below the terminal is a screenshot of the Gedit text editor. The title bar says "*Untitled Document 1 - gedit". The menu bar includes File, Edit, View, Search, Tools, Documents, Help. The toolbar includes New, Open, Save, Print, Undo, Redo, Cut. The main window shows the text "Editing a file". Status bar at the bottom shows Plain Text, Tab Width: 8, Ln 1, Col 15, INS.

Killing jobs

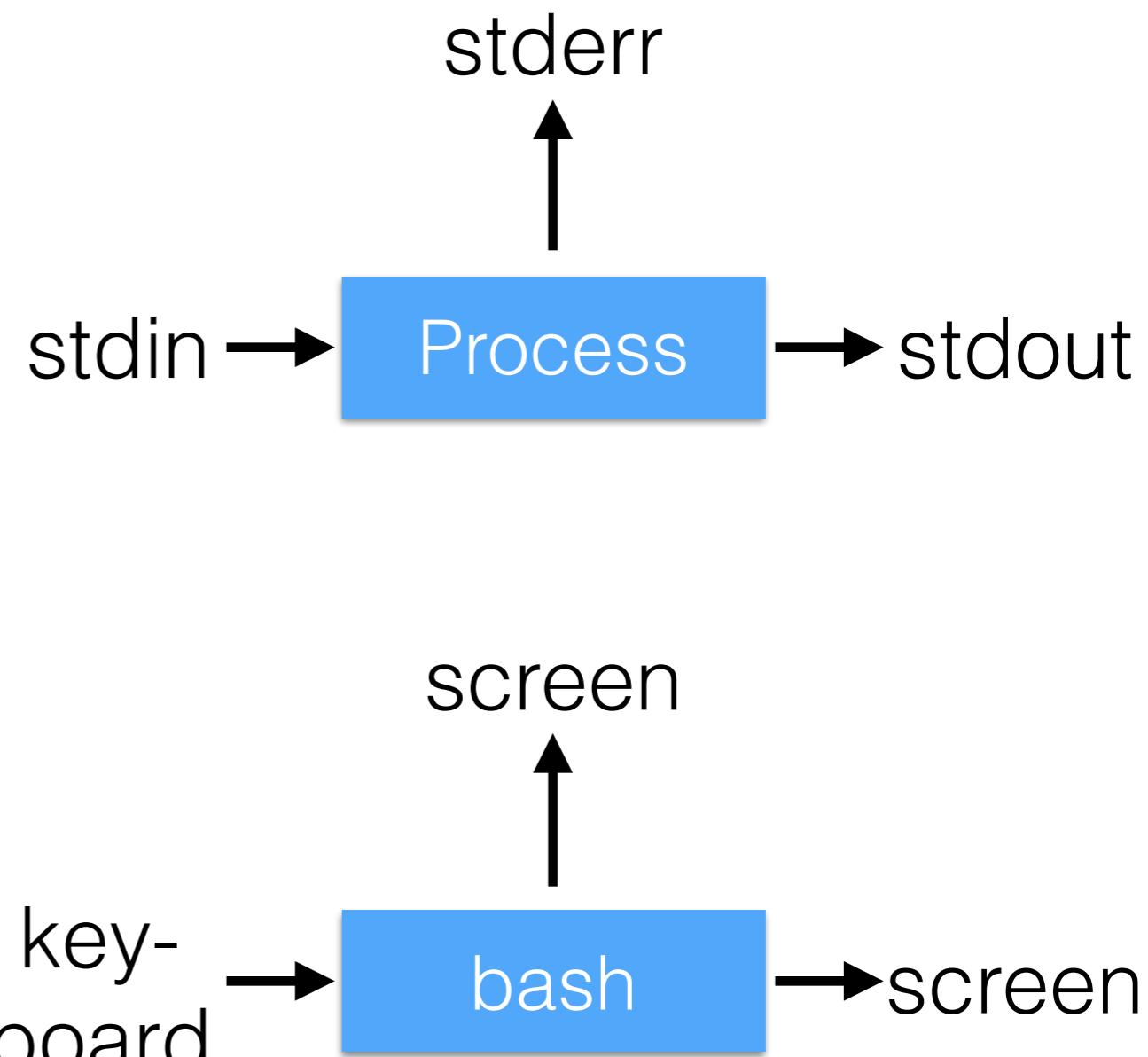
- Sometime we want to get rid of running processes.
- To remove a process from the system we use the **kill** command
 - **kill <pid>**
- You can find the pid either by running ps, or by using the pidof command
 - **pidof <name>**
- Sometimes we want to kill a process and all of it's children. to do this we use:
 - **kill -9 <pid>**

```
1. gareth@brutha: ~/Examples2 (ssh)
gareth@brutha:~/Examples2$ ./infinite > /dev/null &
[1] 23300
gareth@brutha:~/Examples2$ jobs
[1]+  Running                  ./infinite > /dev/null &
gareth@brutha:~/Examples2$ ps
  PID TTY      TIME CMD
17397 pts/0    00:00:00 bash
17690 pts/0    00:00:00 dbus-launch
23300 pts/0    00:00:08 infinite
23337 pts/0    00:00:00 ps
gareth@brutha:~/Examples2$ pidof infinite
23300
gareth@brutha:~/Examples2$ kill 23300
gareth@brutha:~/Examples2$ ps
  PID TTY      TIME CMD
17397 pts/0    00:00:00 bash
17690 pts/0    00:00:00 dbus-launch
23412 pts/0    00:00:00 ps
[1]+  Terminated                 ./infinite > /dev/null
gareth@brutha:~/Examples2$ 
```

- Variables and the Shell
- Processes
- Working with Processes
- IO Streams
- Pipes and Redirections

Standard Streams

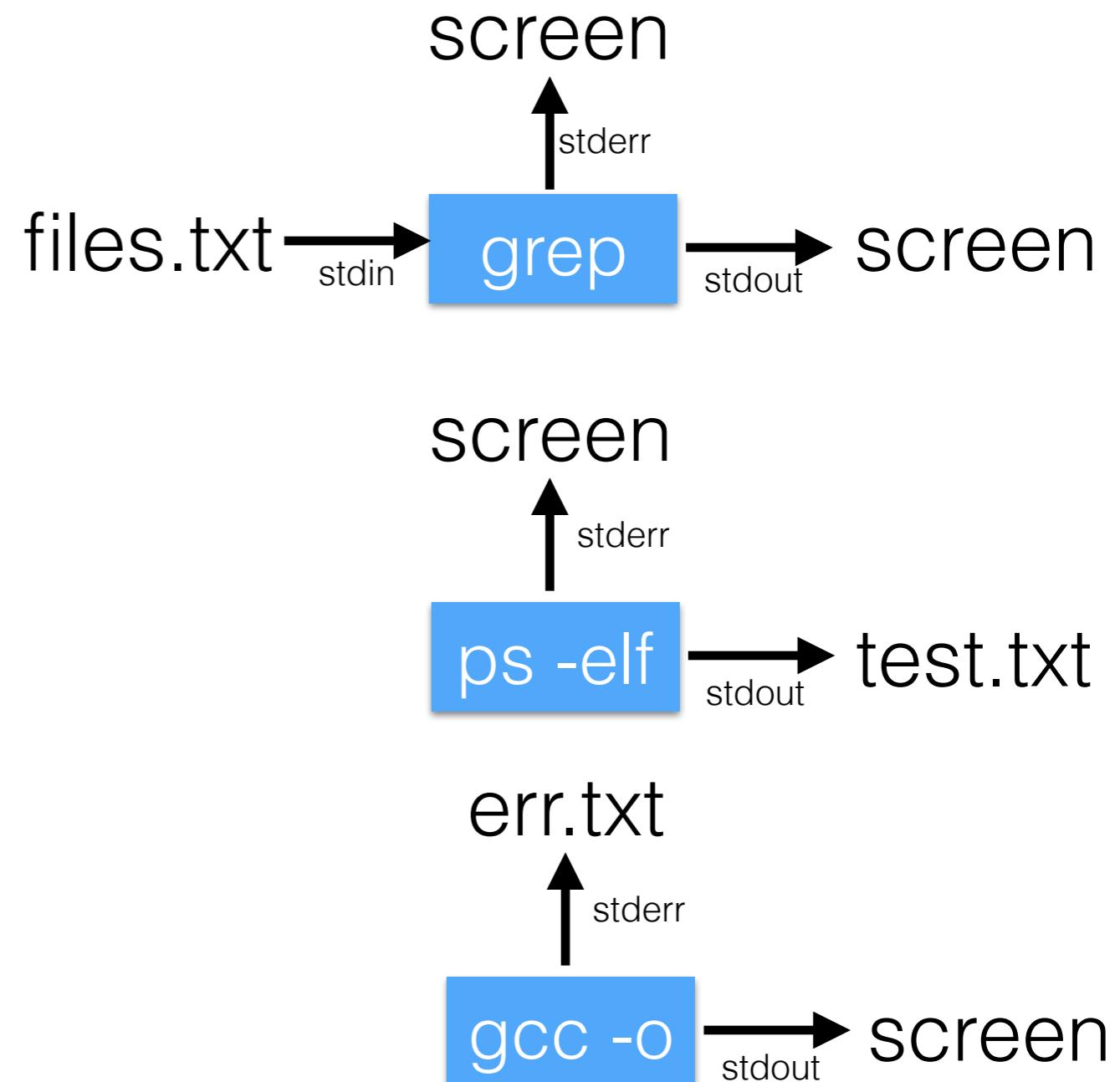
- Each process has three standard input/output streams.
- These come from the C standard and are built-in to every executable that has a “**main**” function.
- **stdin** - is the standard source of input to the program.
- **stdout** - is the standard output of the program, in the cases we have seen this is written to the screen/shell.
- **stderr** - is the standard error of the program, this allows errors to be handled separately from output.



- Variables and the Shell
- Processes
- Working with Processes
- IO Streams
- Pipes and Redirections

Redirection

- The standard streams (**stdin**, **stdout** and **stderr**) can be redirected to point to places.
- You can redirect stdin to put a list of typed commands into a program.
- You can redirect stdout to save the output of a program to a file for later processing.
- You can redirect stderr to save the error messages to look at later, or fix.



Redirection

- On the command line we redirect via the following commands:
- < - redirect stdin from a file or another stream.
- **1>** or **>** - redirect stdout to a file or another stream. If the file didn't exist it will be created, if it did exist it will be overwritten.
- **2>** - redirect stderr to a file or another stream
- **>&1** - redirect to stdout (i.e. **2>&1** to redirect stderr to stdout)
- **>&2** - redirect to stderr
- **>>** - append rather than overwrite.

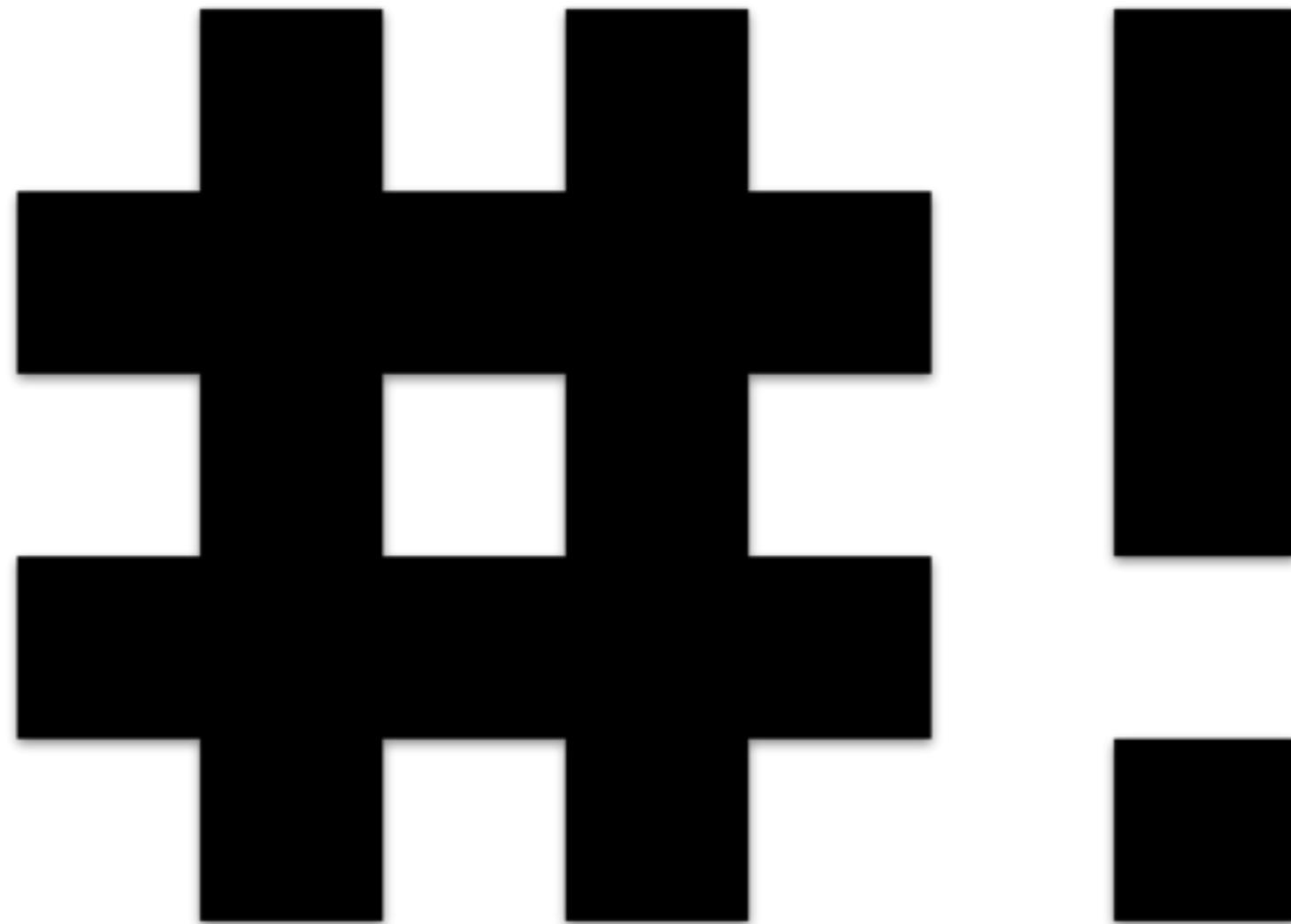
```
1. gareth@brutha: ~/Examples2 (ssh)
gareth@brutha:~/Examples2$ grep for < loop2.c
for (i=0; i < length; i++){
    for (i=1; i <= length; i++) {
gareth@brutha:~/Examples2$ ps > process.txt
gareth@brutha:~/Examples2$ cat process.txt
  PID TTY      TIME CMD
17397 pts/0    00:00:00 bash
17690 pts/0    00:00:00 dbus-launch
43873 pts/0    00:00:00 ps
gareth@brutha:~/Examples2$ gcc
gcc: fatal error: no input files
compilation terminated.
gareth@brutha:~/Examples2$ gcc 2> errors.txt
gareth@brutha:~/Examples2$ cat errors.txt
gcc: fatal error: no input files
compilation terminated.
gareth@brutha:~/Examples2$ 
```

Pipes

- We often want to take the output of one command and use it as the input of another command.
- Using redirection we could do something like:
 - **ps -elf > temp.txt && grep n_tty < temp.txt**
- This is such a common pattern on UNIX systems that the OS provides a system for doing this called pipes.
- The above command could be re-written as:
 - **ps -elf | grep n_tty**
- Where the | character means take the stout of the first command and attach it to the second.
- This allows us to chain together multiple commands.

```
1. gareth@brutha: ~/Examples2 (ssh)
gareth@brutha:~/Examples2$ ps -elf > temp.txt && grep n_tty < temp.txt
4 S root      1705  1  0  80  0 - 25964 n_tty_  2014 tty4  00:0
0:00 /sbin/getty -8 38400  tty4
4 S root      1713  1  0  80  0 - 25964 n_tty_  2014 tty5  00:0
0:00 /sbin/getty -8 38400  tty5
4 S root      1732  1  0  80  0 - 25964 n_tty_  2014 tty2  00:0
0:00 /sbin/getty -8 38400  tty2
4 S root      1733  1  0  80  0 - 25964 n_tty_  2014 tty3  00:0
0:00 /sbin/getty -8 38400  tty3
4 S root      1736  1  0  80  0 - 25964 n_tty_  2014 tty6  00:0
0:00 /sbin/getty -8 38400  tty6
4 S root      2530  1  0  80  0 - 25964 n_tty_  2014 tty1  00:0
0:00 /sbin/getty -8 38400  tty1
0 S 2080089M  7540  7533  0  80  0 - 28189 n_tty_ Jan21 pts/5  00:0
0:00 bash
0 S mdeans    14281 14243  0  80  0 - 28236 n_tty_ Jan20 pts/13  00:0
0:00 bash
0 S mdeans    27672 14243  0  80  0 - 28239 n_tty_ Jan22 pts/12  00:0
0:00 bash
0 S mwood     35105 35082  0  80  0 - 28278 n_tty_ Jan20 pts/11  00:0
0:01 /bin/bash
0 S sams      45877 45866  0  80  0 - 25454 n_tty_ 11:48 pts/15  00:0
0:00 pager -s
0 S 2083412S  47319 47312  0  80  0 - 28200 n_tty_ Jan23 pts/29  00:0
0:00 /bin/bash
0 S 20878010  58152 58145  0  80  0 - 29356 n_tty_ Jan23 pts/14  00:0
0:00 /bin/bash
gareth@brutha:~/Examples2$ ps -elf |grep n_tty |wc
      15      246     1366
gareth@brutha:~/Examples2$
```

- Variables and the Shell
- Processes
- Working with Processes
- IO Streams
- Pipes and Redirections



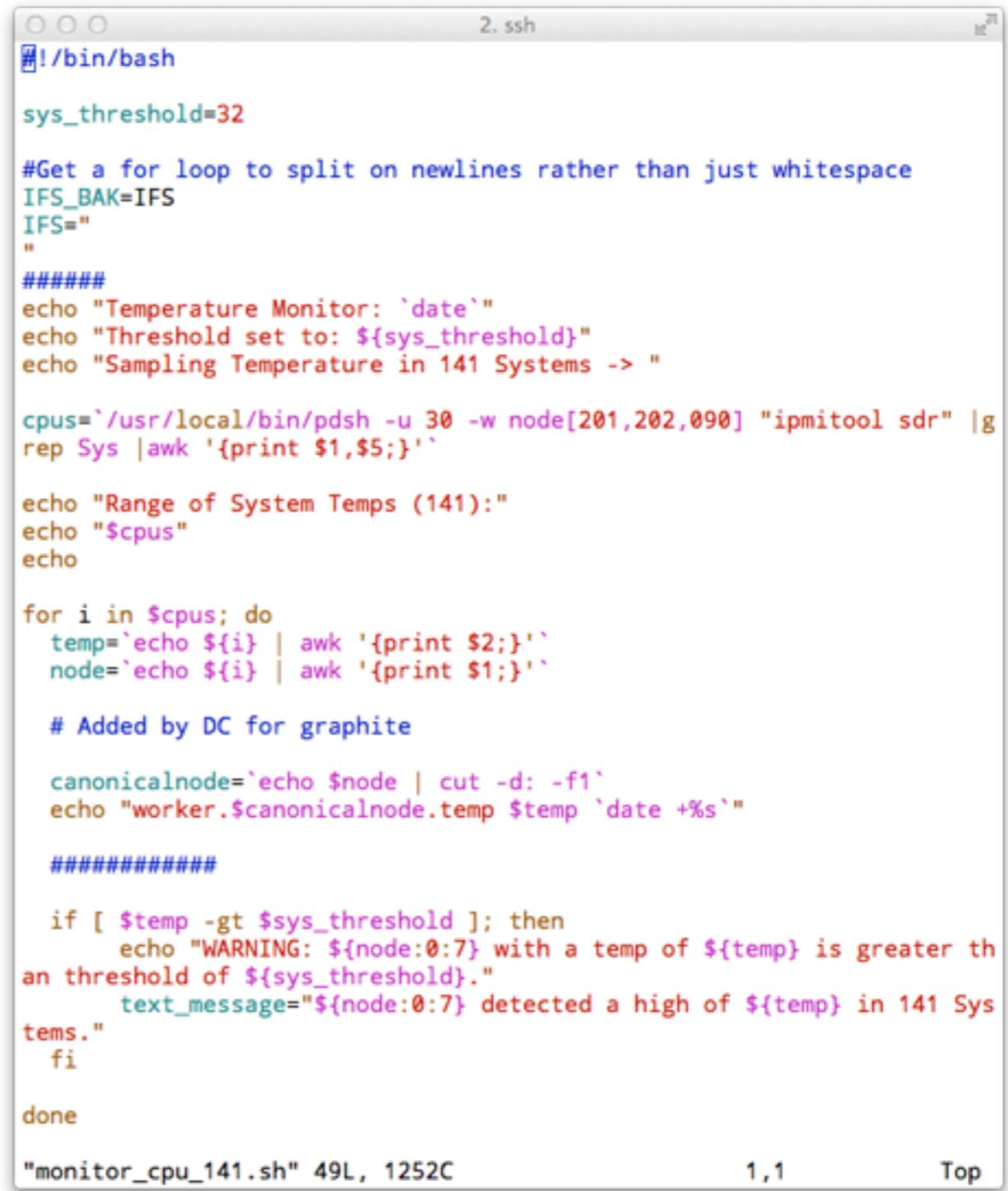
Bash Scripting

Dr. Gareth Roy (x6439)
gareth.roy@glasgow.ac.uk

- A simple script
- Tests (if)
- Ranges & Lists
- Loops (for, while)
- Special Vars

Shell Scripts

- Shell and script programming are useful tools in a physicist's armoury.
- Scripting allows the automation of easy, repetitive tasks.
- Example tasks include: checking webserver logs, backing up data to remote disk, creating directory structures or checking the workings of an automated experiment.
- Shell scripts are deceptively simple, but can be powerful when combined with the many tools found on a Linux platform
- Other scripting languages you may come across include perl, python and ruby. In the rest of this course we will focus on BASH.



```
#!/bin/bash
sys_threshold=32

#Get a for loop to split on newlines rather than just whitespace
IFS_BAK=$IFS
IFS="
"
#####
echo "Temperature Monitor: `date`"
echo "Threshold set to: ${sys_threshold}"
echo "Sampling Temperature in 141 Systems ->"

cpus=`/usr/local/bin/pdsh -u 30 -w node[201,202,090] "ipmitool sdr" | grep Sys | awk '{print $1,$5}'` 

echo "Range of System Temps (141):"
echo "$cpus"
echo

for i in $cpus; do
    temp=`echo ${i} | awk '{print $2;}'` 
    node=`echo ${i} | awk '{print $1;}'` 

    # Added by DC for graphite
    canonicalnode=`echo $node | cut -d: -f1` 
    echo "worker.$canonicalnode.temp $temp `date +%s`" 

#####

    if [ $temp -gt $sys_threshold ]; then
        echo "WARNING: ${node}:0:7 with a temp of ${temp} is greater than threshold of ${sys_threshold}." 
        text_message="${node}:0:7 detected a high of ${temp} in 141 Systems."
    fi
done

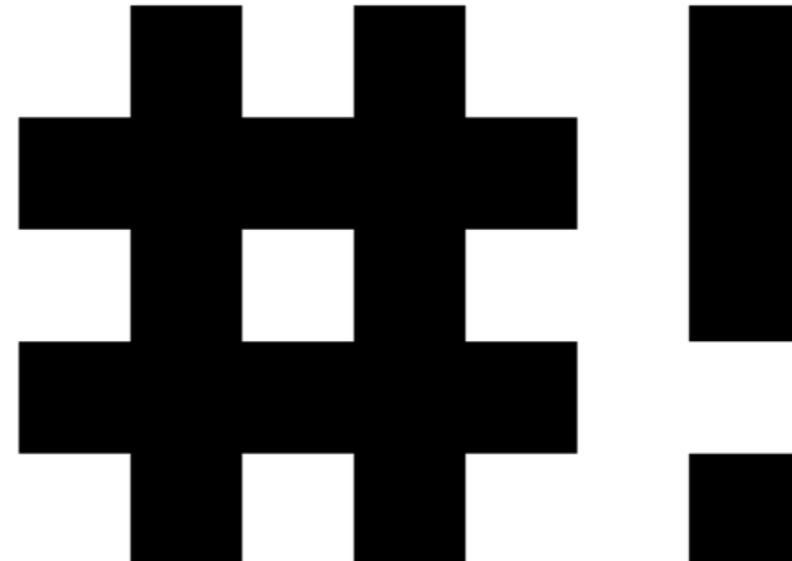
"monitor_cpu_141.sh" 49L, 1252C
```

1,1

Top

Shell Scripts

- In this course we're only going to cover the basics.
- If you want to know more there are some great resources on the web.
- There are also a lot of gotchas so things like stackoverflow.com can also be helpful.
- **Just remember, always cite sources if using things directly from online guides.**



Scripts in this Lecture: /students/p2t-16/share/LinuxLab04/scripts

BEGINNERS GUIDE

<http://tldp.org/LDP/Bash-Beginners-Guide/html/index.html>

ADVANCED SCRIPTING

<http://tldp.org/LDP/abs/html/index.html>

- A simple script
- Tests (if)
- Ranges & Lists
- Loops (for, while)
- Special Vars

A Simple Script

```
3. gareth@brutha: ~/Lec04 (ssh)
1 #!/bin/bash
2
3 # This is a comment. The computer ignores anything after the '#'
4
5 # Check contents, hostname and username
6 echo "You have the following files in this directory"
7 ls --color=auto
8
9 echo "You are working on: ${HOSTNAME}."
10 echo "You are called ${USER} on this machine."
11
12 # Sleep a little bit
13 echo "Sleeping for 2 seconds..."
14 sleep 1
15 echo "1"
16 sleep 1
17 echo "2"
18
19 # Exiting, returning success (0)
20 echo "Exiting now, will return (0)"
21 exit 0
22 
```

simple.sh [+] 22,1 All
-- INSERT --

```
3. gareth@brutha: ~/Lec04 (ssh)
gareth@brutha:~/Lec04$ ./simple.sh
You have the following files in this directory
simple.sh
You are working on: brutha.
You are called gareth on this machine.
Sleeping for 2 seconds...
1
2
Exiting now, will return (0)
gareth@brutha:~/Lec04$ 
```

A Simple Script

- A simple script is just a text file with a series of commands.
- The usual file extension used to identify a shell script is ***.sh**
- **#!** is an interpreter directive. It instructs the shell to run the command specified with the current file.
- It's often referred to as the “**shebang**”, “**hash-bang**” or “**hash-pling**”.
- **#!/bin/bash** translates to:
 - **/bin/bash \$PWD/simple.sh**
- All bash scripts should start with this line.

The screenshot shows a terminal window titled "3. gareth@brutha: ~/Lec04 (ssh)". The window displays a bash script with the following content:

```
1 #!/bin/bash
2
3 # This is a comment. The computer ignores anything after the '#'
4
5 # Check contents, hostname and username
6 echo "You have the following files in this directory"
7 ls --color=auto
8
9 echo "You are working on: ${HOSTNAME}."
10 echo "You are called ${USER} on this machine."
11
12 # Sleep a little bit
13 echo "Sleeping for 2 seconds..."
14 sleep 1
15 echo "1"
16 sleep 1
17 echo "2"
18
19 # Exiting, returning success (0)
20 echo "Exiting now, will return (0)"
21 exit 0
22
```

The terminal window has a vertical scroll bar on the left. At the bottom, it shows the file name "simple.sh [+]". On the right side, there are status indicators: "22,1" and "All".

A Simple Script

- We can make our scripts more understandable by inserting comments
- The '#' character at the beginning of the line denotes it is a comment and these are ignored by the interpreter.
- Our simple script:
 - uses **echo** to output to the screen
 - executes the **ls** command
 - displays the contents of some environment variables (**HOSTNAME** and **USER**)
 - uses **sleep** to pause for 2 seconds
 - uses **exit** to return the value 0 (denoting successful completion).
 - **exit** terminates the program and returns a numeric value to the calling process. By default 0 is success and any numeric value is failure.

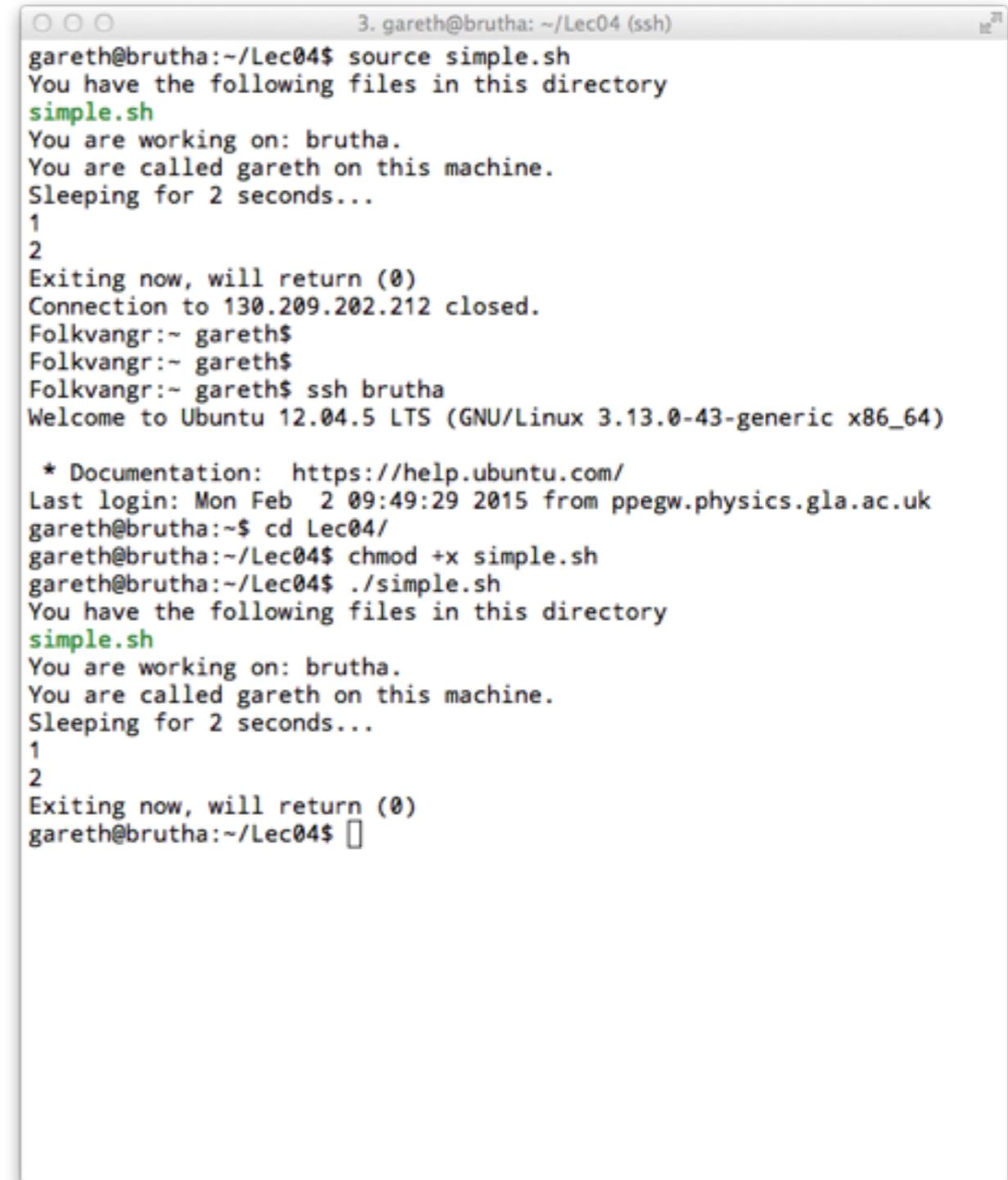
The screenshot shows a terminal window titled "3. gareth@brutha: ~/Lec04 (ssh)". The window contains the following code:

```
1 #!/bin/bash
2
3 # This is a comment. The computer ignores anything after the '#'
4
5 # Check contents, hostname and username
6 echo "You have the following files in this directory"
7 ls --color=auto
8
9 echo "You are working on: ${HOSTNAME}."
10 echo "You are called ${USER} on this machine."
11
12 # Sleep a little bit
13 echo "Sleeping for 2 seconds..."
14 sleep 1
15 echo "1"
16 sleep 1
17 echo "2"
18
19 # Exiting, returning success (0)
20 echo "Exiting now, will return (0)"
21 exit 0
22
```

The terminal window has a vertical scroll bar on the left. At the bottom, the status bar shows "simple.sh [+] 22,1 All" and "-- INSERT --".

Running the script

- We can run the script in two ways:
 - **source simple.sh**
 - **chmod +x simple.sh; ./simple.sh**
- **source** runs the commands in simple.sh one after another in the current shell.
- Marking the script as executable and running it via **./simple.sh** forks a new process.
- Running a script using **source** can be dangerous as it can overwrite variables in your current session.
- It will also exit your current session if it encounters an **exit** statement.



The screenshot shows a terminal window with the following session:

```
gareth@brutha:~/Lec04$ source simple.sh
You have the following files in this directory
simple.sh
You are working on: brutha.
You are called gareth on this machine.
Sleeping for 2 seconds...
1
2
Exiting now, will return (0)
Connection to 130.209.202.212 closed.
Folkvangr:~ gareth$
Folkvangr:~ gareth$
Folkvangr:~ gareth$ ssh brutha
Welcome to Ubuntu 12.04.5 LTS (GNU/Linux 3.13.0-43-generic x86_64)

 * Documentation: https://help.ubuntu.com/
Last login: Mon Feb  2 09:49:29 2015 from ppegw.physics.gla.ac.uk
gareth@brutha:~$ cd Lec04/
gareth@brutha:~/Lec04$ chmod +x simple.sh
gareth@brutha:~/Lec04$ ./simple.sh
You have the following files in this directory
simple.sh
You are working on: brutha.
You are called gareth on this machine.
Sleeping for 2 seconds...
1
2
Exiting now, will return (0)
gareth@brutha:~/Lec04$
```

An aside on Variables

- We can assign the output of a command to a variable.
- There are two ways to do this:
 - **MYUSER=\$(whoami)**
 - **MYUSER=`whoami`**
- There are two types of strings in bash
 - “” - double quoted
 - ‘’ - single quoted
- In double quotes strings variable names are replaced with their value.
- In single quoted strings variable substitution does not take place.

The screenshot shows two terminal windows. The top window, titled '3. gareth@brutha: ~/Lec04 (ssh)', contains the following script code:

```
1 #!/bin/bash
2
3 # Store the results of executing whoami
4 MYUSER=$(whoami)
5 echo "You are called ${MYUSER} on this machine."
6
7 # Another way to store the results
8 MYUSER2=`whoami`
9 echo "You are called ${MYUSER2} on this machine."
10
11 # Variables are not substituted in single quotes strings
12 MYUSER3=$(whoami)
13 echo 'You are called ${MYUSER3} on this machine.'
14
15 # Exiting, returning success (0)
16 echo "Exiting now, will return (0)"
17 exit 0
18
```

The bottom window, titled '4. gareth@brutha: ~/Lec04 (ssh)', shows the execution of the script and its output:

```
gareth@brutha:~/Lec04$ ./simple2.sh
You are called gareth on this machine.
You are called gareth on this machine.
You are called ${MYUSER3} on this machine.
Exiting now, will return (0)
gareth@brutha:~/Lec04$
```

A small 'All' button is visible in the bottom right corner of the terminal window.

- A simple script
- Tests (if)
- Ranges & Lists
- Loops (for, while)
- Special Vars

If, then, else

- **if** statements are similar to those found in C.
- They allow conditional execution of code, allowing decisions to be made about what to execute
- For instance **if** a file exists, **then** remove that file, **else** log an error.
- There are three general forms for an **in** **if** statement in Bash:
 - **if** **x then** **y**
 - **if** **x then** **y else** **z**
 - **if** **x then** **y elif** **a then** **z**

```
if [ conditional ]; then  
    some command  
fi
```

```
if [ conditional ]  
then  
    some command  
else  
    some other command  
fi
```

```
if [ conditional ]  
then  
    some command  
elif [ conditional ]  
    some other command  
else  
    yet another command  
fi
```

Bash Conditionals

[-f tmp.txt]

String Comparision	Result
string 1 == string 2	True if the strings are equal
string 1 != string 2	True if the strings are different
-n string	True if the string is not null
-z string	True if the string is null

File Conditionals	Result
-d file	True if file is a directory
-e file	True if file exists
-f file	True if the file is regular
-r file	True if file is readable
-s file	True if file has nonzero size
-w file	True if file is writeable
-x file	True if file is executable

Arithmetic Comparision	Result
exp1 -eq exp2	True if both are equal
exp1 -ne exp2	True if both are different
exp1 -gt exp2	True if exp1 is greater than exp2
exp1 -ge exp2	True if exp1 is greater than or equal to exp2
exp1 -lt exp2	True if exp1 is less than exp2
exp1 -le exp2	True if exp1 is less than or equal to exp2
! exp	Invertes exp, true if exp is false. False if exp is true

If, then, else

- Here is an example script using if statements and conditionals.
- The script runs **whoami**, and stores the value in a variable. It then creates an empty file called **temp_file**.
- If the file exists it writes a message and lists the directory.
- The script checks to see if a **lock** file is present, if it is it writes a message and exits with an error.
- If the lock file is not present it checks to make sure we are not the **root** user, and if not it removes **temp_file**.
- Finally it lists the directory contents and exits with a success

The terminal window shows the execution of a shell script named `simple3.sh`. The script contains the following code:

```
#!/bin/bash
# Get the user running this script
MYID=$(whoami)
# Create an empty file
touch temp_file
# Test to see if the file exists
if [ -e temp_file ]; then
    echo "My temp_file exists!! see?"
    ls
fi
#If I haven't locked the directory rm the file
if [ -e lock ]; then
    echo "Sorry it appears you've locked the directory!"
    exit 1
elif [ $MYID == "root" ]; then
    echo "Please don't run this as root!"
    exit 2
else
    echo "Deleting the temp_file"
    rm temp_file
fi
echo "Dir contents:"
ls
# Exiting, returning success (0)
echo "Exiting now, will return (0)"
exit 0
```

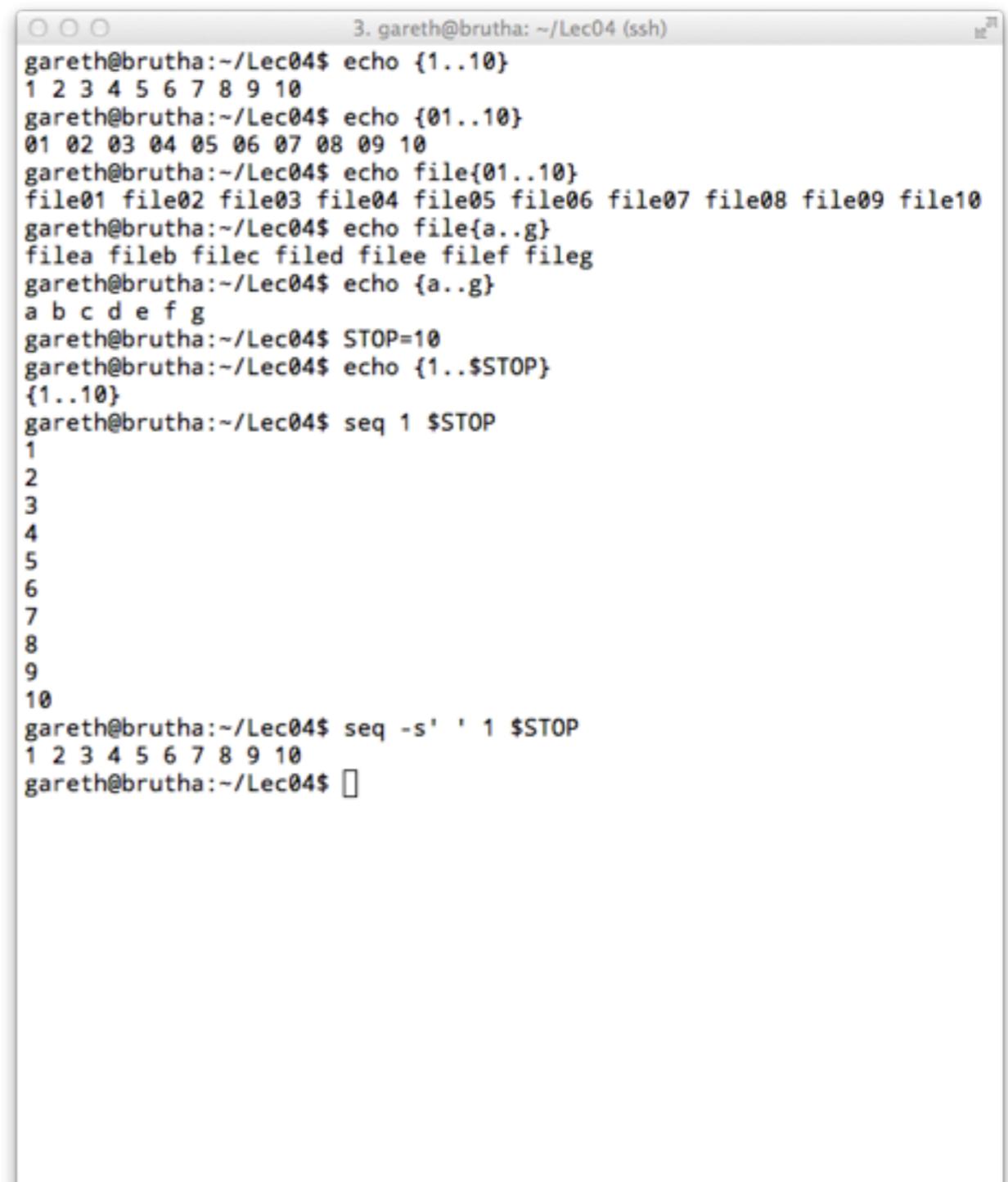
The terminal session shows the script being run and its output:

```
gareth@brutha:~/Lec04$ ./simple3.sh
My temp_file exists!! see?
simple2.sh simple3.sh simple.sh temp_file
Deleting the temp_file
Dir contents:
simple2.sh simple3.sh simple.sh
Exiting now, will return (0)
gareth@brutha:~/Lec04$ touch lock
gareth@brutha:~/Lec04$ ./simple3.sh
My temp_file exists!! see?
lock simple2.sh simple3.sh simple.sh temp_file
Sorry it appears you've locked the directory!
gareth@brutha:~/Lec04$ ls
lock simple2.sh simple3.sh simple.sh temp_file
gareth@brutha:~/Lec04$
```

- A simple script
- Tests (if)
- Ranges & Lists
- Loops (for, while)
- Special Vars

Ranges

- We can specify numeric ranges in Bash using the **{start..stop}** notation.
- A range of 1 to 3 would be written as:
 - **{1..3}** => 1 2 3
 - **{01..03}** => 01 02 03
- Range can be characters as well as numbers:
 - **{a..c}** => a b c
 - **{A..C}** => A B C
- Unfortunately we cannot use variables in the above definition, which means we cannot change the range while running a script.
- For numerical ranges we can use a command called **seq**.
 - **STOP=20; seq 1 \${STOP}**
- We'll use ranges frequently while writing loops in bash.



```
gareth@brutha:~/Lec04$ echo {1..10}
1 2 3 4 5 6 7 8 9 10
gareth@brutha:~/Lec04$ echo {01..10}
01 02 03 04 05 06 07 08 09 10
gareth@brutha:~/Lec04$ echo file{01..10}
file01 file02 file03 file04 file05 file06 file07 file08 file09 file10
gareth@brutha:~/Lec04$ echo file{a..g}
filea fileb filec filed filee filef fileg
gareth@brutha:~/Lec04$ echo {a..g}
a b c d e f g
gareth@brutha:~/Lec04$ STOP=10
gareth@brutha:~/Lec04$ echo {1..$STOP}
{1..10}
gareth@brutha:~/Lec04$ seq 1 $STOP
1
2
3
4
5
6
7
8
9
10
gareth@brutha:~/Lec04$ seq -s' ' 1 $STOP
1 2 3 4 5 6 7 8 9 10
gareth@brutha:~/Lec04$ []
```

Lists

- Bash treats a group of space separated strings as a list.
- You can specify a list using brace expansion similar to ranges (be careful not to include spaces):

- **{a,b,c,d}**

- Bash treats a double or single quoted string as a list if it contains spaces so:

- **MYVAR="a b c d"**

- **MYVAR='a b c d'**

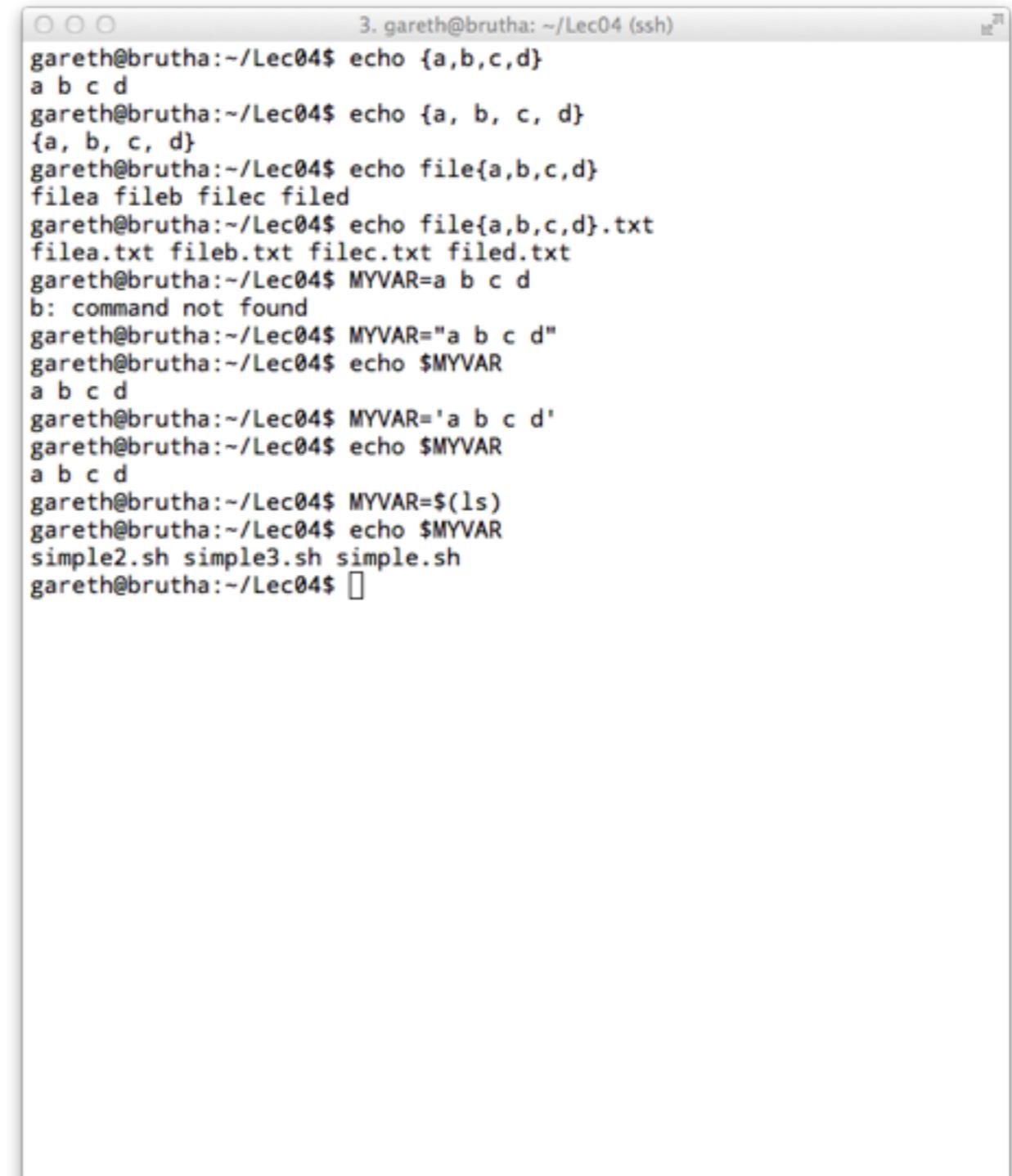
- This useful when using filesystem commands such as **ls**:

- **MYVAR=\$(ls)**

- When we come to loops we can also specify a list as below although it's usually better to wrap values in a variable:

- **a b c d**

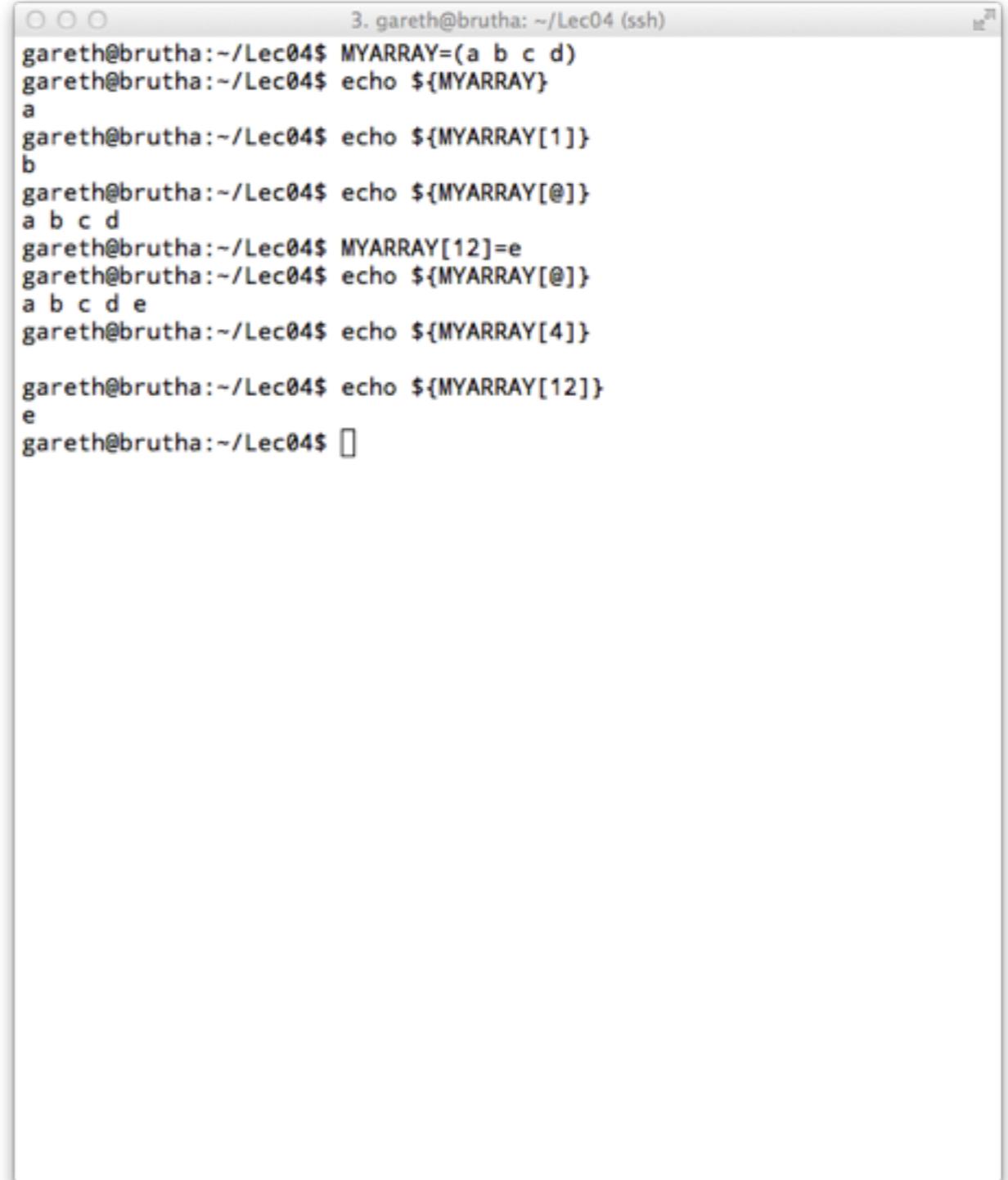
- **"a" "b" "c" "d"**



```
3. gareth@brutha: ~/Lec04 (ssh)
gareth@brutha:~/Lec04$ echo {a,b,c,d}
a b c d
gareth@brutha:~/Lec04$ echo {a, b, c, d}
{a, b, c, d}
gareth@brutha:~/Lec04$ echo file{a,b,c,d}
filea fileb filec filed
gareth@brutha:~/Lec04$ echo file{a,b,c,d}.txt
filea.txt fileb.txt filec.txt filed.txt
gareth@brutha:~/Lec04$ MYVAR=a b c d
b: command not found
gareth@brutha:~/Lec04$ MYVAR="a b c d"
gareth@brutha:~/Lec04$ echo $MYVAR
a b c d
gareth@brutha:~/Lec04$ MYVAR='a b c d'
gareth@brutha:~/Lec04$ echo $MYVAR
a b c d
gareth@brutha:~/Lec04$ MYVAR=$(ls)
gareth@brutha:~/Lec04$ echo $MYVAR
simple2.sh simple3.sh simple.sh
gareth@brutha:~/Lec04$ 
```

Arrays

- Newer versions of Bash also have arrays, which can be accessed via indexes.
- To create an array you would do the following:
 - **MYARRAY=(a b c d)**
- Arrays are 0 indexed so below would return the second element of the array:
 - **echo \${MYARRAY[1]}**
- You can access all the elements of an array by passing the special '@' character as an index.
 - **echo \${MYARRY[@]}**
- Array indexes don't need to be consecutive, so we could add an index at 12 by doing:
 - **MYARRAY[12]=e**



```
3. gareth@brutha: ~/Lec04 (ssh)
gareth@brutha:~/Lec04$ MYARRAY=(a b c d)
gareth@brutha:~/Lec04$ echo ${MYARRAY}
a
gareth@brutha:~/Lec04$ echo ${MYARRAY[1]}
b
gareth@brutha:~/Lec04$ echo ${MYARRAY[@]}
a b c d
gareth@brutha:~/Lec04$ MYARRAY[12]=e
gareth@brutha:~/Lec04$ echo ${MYARRAY[@]}
a b c d e
gareth@brutha:~/Lec04$ echo ${MYARRAY[4]}
gareth@brutha:~/Lec04$ echo ${MYARRAY[12]}
e
gareth@brutha:~/Lec04$ 
```

- A simple script
- Tests (if)
- Ranges & Lists
- Loops (for, while)
- Special Vars

For loops

- There are two types of for loops used in Bash scripts.
- A C-style three expression for loop, with an initialiser, condition and increment
- An iterator-style for loop, which take each item in a list and places that value in a loop variable.
- **Note:** It is rare to find the C-style for loop used in Bash (if ever).
- The iterator style is more prevalent as the list can come from Ranges, Lists, Arrays or running commands.
- This style is similar to the foreach construct found in other scripting/programming languages (java, C#, etc).

```
for (( exp1;exp2;exp3 ))  
do  
    some command  
done
```

```
for i in {1..5}  
do  
    some command  
done
```

```
for file in $(ls)  
do  
    some command  
done
```

```
for file in /etc/*  
do  
    some command  
done
```

For loops

- In our example, we can see C and Range style for loops.
- Notice that the ranges don't need to be numerical.
- The more complicated examples iterate over all the files in our current directory.
- The first loop prints the output of **file** on each item in the directory.
- The second, takes the output of **file**, grabs the second element from the output and stores that in a variable.
- It then tests to see if that is a shell script, and if so prints a message to screen.

```
3. gareth@brutha: ~/Lec04 (ssh)
1#!/bin/bash
2
3 # A C style for loop
4 for (( c=1; c<=3; c++ ))
5 do
6         echo "I am a C style iteration ${c}!"
7 done
8
9 # A Bash style range for loop
10 for c in {a..d}
11 do
12         echo "I am Range style iteration ${c}!"
13 done
14
15 # A more useful for loop, return the type of each file in a dir.
16 # Instead of * we could use 'ls' or $(ls)
17 for FILE in *
18 do
19         file -i ${FILE}
20 done
21
22 # A more complicated example, lets loop over the files
23 # check their types and if it is a shellscript write a message
24 for FILE in *
25 do
26         TYPE=`file -i ${FILE} | awk '{print $2}'`
27         if [ ${TYPE} == "text/x-shellscript;" ]
28         then
29                 echo "${FILE} is a shellscript"
30         fi
31 done
```

```
4. gareth@brutha: ~/Lec04 (ssh)
gareth@brutha:~/Lec04$ ./simple4.sh
I am a C style iteration 1!
I am a C style iteration 2!
I am a C style iteration 3!
I am Range style iteration a!
I am Range style iteration b!
I am Range style iteration c!
I am Range style iteration d!
simple2.sh: text/x-shellscript; charset=us-ascii
simple3.sh: text/x-shellscript; charset=us-ascii
simple4.sh: text/x-shellscript; charset=us-ascii
simple.sh: text/x-shellscript; charset=us-ascii
simple2.sh is a shellscript
simple3.sh is a shellscript
simple4.sh is a shellscript
simple.sh is a shellscript
gareth@brutha:~/Lec04$
```

While loops

- While loops will continue to loop while some condition holds true.
- You can use any of the conditionals we've already discussed (file, string or arithmetic).
- While loops are often used to create infinite loops that will exit on external conditions:
 - waiting for a file to be created then carry out an action.
 - carry out a task at a specific time interval (sampling data).
 - waiting for a long running process to complete.

```
while [ conditional ]
do
    some command
done
```

```
while [ conditional ]; do
    some command
done
```

```
while true
do
    some command
    break
done
```

While loops

- In this example we use two while loops.
- The first loops until a variable **SECONDS** is equal to **12**
- **SECONDS** is set using the **date** command to contain the seconds of the current time.
- When this loop exits it **echo's** the current time.
- The second while loop continues to loop unless a **lock** file exists at which point we **break** out of the loop and exit.

```
3. gareth@brutha: ~/Lec04 (ssh)
1 #!/bin/bash
2
3 SECONDS=''
4
5 # Wait for seconds to reach 12s
6 while [ ! ${SECONDS} -eq 12 ]
7 do
8     SECONDS=`date +%S`
9     echo ${SECONDS}
10    sleep 1
11 done
12 echo "Done waiting, its $(date)"
13
14 # Wait until a lock file exists then exit
15 while true
16 do
17     if [ -e lock ]; then
18         break
19     fi
20 done
21
22 echo "Exiting..."
23 exit 0
24 
```

```
4. gareth@brutha: ~/Lec04 (ssh)
gareth@brutha:~/Lec04$ ./simple5.sh
5
6
7
8
9
10
11
Done waiting, its Tue Feb  3 10:48:12 GMT 2015
Exiting...
gareth@brutha:~/Lec04$ 
```

- A simple script
- Tests (if)
- Ranges & Lists
- Loops (for, while)
- Special Vars

Special Variables

- Bash has some special variables that provide useful information.
- \$1, \$2, \$@ and \$# are used to work with arguments given to the shell script.
- \$0 give the name of the running script
- \$\$ provides the script with it's PID when running.
- \$? is used to check the return code of a command run as part of the script.
- This is useful to check for any error that occurred.

Variable	Usage
\$\$	Process ID of the running Bash script.
\$0	The name of the Bash script
\$1, \$2, ...	The first, second, ... argument passed to the Bash script
#	The number of arguments passed to the Bash script
\$?	The return code of the previous command

Special Variables

- In this example we:
 - echo our PID
 - echo our scripts name
 - echo the number of arguments
 - echo the argument list
 - echo the first argument
 - tests the exit code of running ls and gcc (note ls succeeds and gcc fails).

The image shows two terminal windows. The top window, titled '3. gareth@brutha: ~/Lec04 (ssh)', contains the source code of a shell script named simple6.sh. The bottom window, titled '4. gareth@brutha: ~/Lec04 (ssh)', shows the output of running the script with four arguments: a, b, c, d.

```
1 #!/bin/bash
2
3 # $$ gives the Process ID (PID) of the process running me
4 echo "My PID is: $$"
5
6 # $0 returns the name of the script
7 echo "My name is: $0"
8
9 # $# gives the number of arguments passed to the script
10 echo "I was started with $# arguments"
11
12 # $@ is a list of all the arguments passed
13 echo "My arguments are: $@"
14
15 # $1 returns the first argument passed to the script
16 echo "My first argument was: $1"
17
18 # $? returns the exit code of the previous command
19 # Good for finding things that failed. 0 - success, > 0 - fail
20 echo "Running (ls)"
21 ls > /dev/null 2>&1
22 echo "Return code is $?"
23
24 echo "Running (gcc)"
25 gcc > /dev/null 2>&1
26 echo "Return code is $?"
27
28 # Exiting .... Success!
29 echo "Exiting, (0)"
30 exit 0
31
```

```
gareth@brutha:~/Lec04$ ./simple6.sh a b c d
My PID is: 17847
My name is: ./simple6.sh
I was started with 4 arguments
My arguments are: a b c d
My first argument was: a
Running (ls)
Return code is 0
Running (gcc)
Return code is 4
Exiting, (0)
gareth@brutha:~/Lec04$
```

- A simple script
- Tests (if)
- Ranges & Lists
- Loops (for, while)
- Special Vars



More Bash

Dr. Gareth Roy (x6439)
gareth.roy@glasgow.ac.uk

- Input (User & File)
- Functions
- Good Practice
- An Aside - Compilation

- Input (User & File)
- Functions
- Good Practice
- An Aside - Compilation

User Input

- Instead of getting input via passed arguments we can also get it interactively.
- **read** can be used to interactively get input from the User.
- **read** can take a variable to store or if no variable is specified it defaults to **\$REPLY**
- Useful flags that can be passed are:
 - **-p** - specifies a prompt.
 - **-s** - specifies that input is not printed to screen
 - **-n** - specifies the number of characters to read.
- **Note:** It's usually a good idea to check input is present and well formed before using it.

```
1#!/bin/bash
2
3 # read can be used to get input from the user and store
4 # it in a variable
5 echo "Type a word: "
6 read WORD
7
8 # test to see if we actually got a word.
9 if [ -z ${WORD} ]; then
10     echo "No really, you need to type in a word!"
11     exit 1
12 else
13     echo "Your word was - ${WORD}."
14 fi
15
16 # specifying -s means any characters typed won't be echoed
17 # to the screen.
18 echo "Type a secret word!!!!: "
19 read -s SECRET
20 echo "Your secret was - ${SECRET}."
21
22
23 # If we don't specify a variable read defaults to $REPLY
24 # -p allows us to pass a string as a prompt for input.
25 while true
26 do
27     read -p "Do you want to exit? [y/n]: "
28     if [ ${REPLY} == "y" ]; then
29         break
30     elif [ ${REPLY} == "n" ]; then
31         echo "Continuing on..."
32     else
33         echo "${REPLY} not an understood answer."
34     fi
35
36 done
```

```
2. gareth@brutha:~/Lec05 (ssh)
gareth@brutha:~/Lec05$ ./simple.sh
Type a word:
Hello
Your word was - Hello.
Type a secret word!!!!:
Your secret was - Sssssh.
Do you want to exit? [y/n]: n
Continuing on...
Do you want to exit? [y/n]: a
a not an understood answer.
Do you want to exit? [y/n]: y
gareth@brutha:~/Lec05$
```

File Input

- **read** can also be used to read files.
- Often you want to be able to read a particular file line by line and act based on the contents of each line.
- Combining **read** which a **while** loop allows us to do this.
- We create a while loop, whose conditional is to read some data.
- While there is data to read the loop will continue. When all the data has been read, the loop will terminate.
- The data can either be **piped (|)**, or **redirected (<)** into the while loop.

The terminal window shows two parts. Part 1 displays the script content:

```
1. gareth@brutha: ~/Lec05 (ssh)
1 #!/bin/bash
2
3 # Check to see if we've been given a input file
4 # We could also check $# > 0
5 if [ -z $1 ]; then
6     echo "Please specify an input file!"
7     exit 1
8 fi
9
10 # Assume that a file is passed as the first argument to the script
11 # The data can be sent to the while loop either by using a pipe
12 cat $1 | while read LINE
13 do
14     echo "1: ${LINE}"
15 done
16
17 # Or by redirecting the the contents of the file to STDIN
18 while read LINE
19 do
20     echo "2: ${LINE}"
21 done < $1
22
23 exit 0
~
```

Part 2 shows the execution of the script:

```
2. gareth@brutha: ~/Lec05 (ssh)
gareth@brutha:~/Lec05$ cat test_file
10 rations of food
2 blue potions
1 scrol entitles "exyc ahks"
gareth@brutha:~/Lec05$ ./simple2.sh test_file
1: 10 rations of food
1: 2 blue potions
1: 1 scrol entitles "exyc ahks"
2: 10 rations of food
2: 2 blue potions
2: 1 scrol entitles "exyc ahks"
gareth@brutha:~/Lec05$
```

- Input (User & File)
- Functions
- Good Practice
- An Aside - Compilation

Declaring Functions

- A function in Bash consists of a label and a block of code (denoted by {} brackets).
- Functions must be declared before they are used.
- Functions can be declared in two ways:
 - using the **function** keyword and supplying a name, and code block
 - supplying a name filled by (), along with a code block.
- Functions are called in the same ways as command, by simply writing their name
- As with C functions are a good way to be able to reuse pieces of code without having to copy and paste.

```
function name {  
    commands  
    commands  
}  
  
name
```

```
name() {  
    commands  
    commands  
}  
  
name
```

Declaring Functions

- In this example we declare two functions:
 - **hello**
 - **world**
- The hello function echo's a string without appending a newline (**-n**)
- The world function simply echo's a string.
- In the main body of the script we use a for loop, to loop 5 times.
- During each loop we call the hello and world functions.

```
1 #!/bin/bash
2
3 # Function definitions happen before they are used
4 # They have a label, and a body denoted by {}
5 # They can be declared using the function keyword
6 function hello {
7     echo -n "Hello, "
8 }
9
10 # Or by appending () to the label of the function
11 world() {
12     echo "World! (well P2T)"
13 }
14
15 # The main body of the script
16 # Will loop 5 times, and call the helloworld five times
17 for i in {1..5}
18 do
19     hello
20     world
21 done
22
23 exit 0
24
```

```
gareth@brutha:~/Lec05$ ./simple3.sh
Hello, World! (well P2T)
gareth@brutha:~/Lec05$ █
```

Passing Arguments

- Passing parameters to functions in Bash is similar to passing arguments to the script.
- When the function is called Bash passes a list of arguments to it.
- **\$@** - can be used to see all of the arguments passed to the function.
- **\$#** - is the number of arguments passed to the function.
- **\$1** - is the first argument passed to the function.
- **\$2** - is the second argument passed to the function, and so on.
- If there are a large number of parameters passed to the function, **shift** can be used to cycle through them.

```
function name {  
    commands  
    commands  
}  
  
name arg1 arg2 arg3
```

```
name() {  
    commands  
    commands  
}  
  
name arg1 arg2 arg3
```

Function Arguments

- In the example shown we test the use of passing arguments.
- In the first example we see the use of **\$#**, **\$@** and **\$1**.
- In the second example we use **shift** to cycle through the arguments.
- **Note:** each time we call **shift** we discard the first argument from the list one space to the left.
- Using shift is destructive so if you use this method you must store each value you are interested.

The image shows two terminal windows. The top window, titled '1. gareth@brutha: ~/Lec05 (ssh)', contains a Bash script named 'simple4.sh' with the following code:

```
1#!/bin/bash
2
3 # Arguments can be accessed using the same parameters as
4 # the Bash script itself, i.e. $1 represents the first
5 # argument passed to the function
6 function test_args {
7     echo "I was called with $# arguments"
8     echo "The arguments were $@"
9     echo "The first argument was $1"
10    echo
11 }
12
13 # Functions are called in the same way as commands
14 # Arguments are listed after the function name
15 test_args a b c d
16
17 # We can access all the arguments by using a shift function
18 # shift effectively removes the first element and "shifts"
19 # the other elements left.
20 function shift_args {
21     until [ -z $1 ]
22     do
23         echo "There are $# args - $@"
24         echo $1
25         shift
26     done
27 }
28
29 shift_args a b c d
```

The bottom window, titled '2. gareth@brutha: ~/Lec05 (ssh)', shows the output of running the script:

```
gareth@brutha:~/Lec05$ ./simple4.sh
I was called with 4 arguments
The arguments were a b c d
The first argument was a

There are 4 args - a b c d
a
There are 3 args - b c d
b
There are 2 args - c d
c
There are 1 args - d
d
gareth@brutha:~/Lec05$
```

A small number '11' is visible in the bottom right corner of the terminal window.

Global & Local Variables

- In Bash all variables are global within the running script.
- This means variables declared in functions are available outside of the functions calls.
- If this is not desirable behaviour Bash provides the **local** keyword that allows us to restrict the scope of a variable.
- **Note:** Variables declared outside of a function are also available within a Bash function.

The terminal window shows two parts: a script editor and a terminal session.

Script Editor (Top):

```
1#!/bin/bash
2
3 # By default all variables declared in a Bash script have file
4 # scope, or in other words are declared a GLOBAL Variables
5 GLOBALVAR="This is a GLOBALVAR"
6
7 # We can specify a variable should be local to a function by
8 # using the local keyword
9 function variables {
10
11     MYGLOBALVAR="This is also a GLOBALVAR"
12     local MYVAR="This is a local VAR"
13
14     echo "Function: ${GLOBALVAR}"
15     echo "Function: ${MYGLOBALVAR}"
16     echo "Function: ${MYVAR}"
17 }
18
19
20 echo "Main before: ${GLOBALVAR}"
21 echo "Main before: ${MYGLOBALVAR}"
22 echo "Main before: ${MYVAR}"
23
24 variables
25
26 echo "Main after: ${GLOBALVAR}"
27 echo "Main after: ${MYGLOBALVAR}"
28 echo "Main after: ${MYVAR}"
29
30 exit 0
```

Terminal Session (Bottom):

```
gareth@brutha:~/Lec05$ ./simple5.sh
Main before: This is a GLOBALVAR
Main before:
Main before:
Function: This is a GLOBALVAR
Function: This is also a GLOBALVAR
Function: This is a local VAR
Main after: This is a GLOBALVAR
Main after: This is also a GLOBALVAR
Main after:
gareth@brutha:~/Lec05$
```

- Input (User & File)
- Functions
- Good Practice
- An Aside - Compilation

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Brian Kernighan

Good Practice

- ALWAYS use comments, you should include at least a single line comment telling what the script does.
- Use white space to separate ideas, it will make it easier to understand each section of code.
- Use Quoted Variables, i.e. “**\$MYVAR**”, this will get around problems with Bash “word splitting”.
- Always use exit statements, and test sub-commands return codes. Bash has a tendency to fail quietly.

The terminal window shows two tabs. Tab 2 displays the contents of the script simple6.sh, which includes a usage function, argument checking, and a loop to print file lines. Tab 3 shows the execution of the script with arguments, resulting in the expected output of file contents.

```
2. gareth@brutha: ~/Lec05 (ssh)
1 #!/bin/bash
2
3 # A single or multi-line comment telling the person
4 # reading the code what the script is supposed to do.
5
6 function usage {
7     echo "This script prints out a given file line by line"
8     echo "It requires a single argument whcih is the file"
9     echo "to read."
10    echo
11    echo "simple6.sh <filename>"
12 }
13
14 # Test to make sure that an argument has been supplied.
15 if [ -z "$1" ]; then
16     usage
17     exit 1
18 fi
19
20 # Loop over the file and print each line
21 # Prepend each line with the filename.
22 while read LINE; do
23     echo "$1: ${LINE}"
24 done < $1
25
26 exit 0
27 
```

```
3. gareth@brutha: ~/Lec05 (ssh)
gareth@brutha:~/Lec05$ ./simple6.sh
This script prints out a given file line by line
It requires a single argument whcih is the file
to read.

simple6.sh <filename>
gareth@brutha:~/Lec05$ ./simple6.sh test_file
test_file: 10 rations of food
test_file: 2 blue potions
test_file: 1 scrol entitiles "exyc ahks"
gareth@brutha:~/Lec05$ 
```

Debugging

- A Bash script can be just as complicated as a piece of C code.
- Luckily there are some tools we can use to help debug scripts:
 - **set -x** - shows a simple trace of the script executing.
 - **set -n** - stops any commands from running, good for looking at syntax errors.
 - **set -v** - prints out the shell input line as the interpreter runs.
Even more verbose than **set -x**

https://www.gnu.org/software/bash/manual_html_node/The-Set-Builtin.html

The terminal window shows two parts. Part 1 (top) contains the source code of simple6.sh:

```
1 #!/bin/bash
2
3 # Debugging support:
4 export PS4='+$BASH_SOURCE:$LINENO:$FUNCNAME: '
5 set -x
6
7 # A single or multi-line comment telling the person
8 # reading the code what the script is supposed to do.
9
10 function usage {
11     echo "This script prints out a given file line by line"
12     echo "It requires a single argument which is the file"
13     echo "to read."
14     echo
15     echo "simple6.sh <filename>"
16 }
17
18 # Test to make sure that an argument has been supplied.
19 if [ -z "$1" ]; then
20     usage
21     exit 1
22 fi
23
24 # Loop over the file and print each line
25 # Prepend each line with the filename.
26 while read LINE; do
27     echo "$1: ${LINE}"
28 done < $1
29
30 exit 0
```

Part 2 (bottom) shows the execution of the script:

```
gareth@brutha:~/Lec05$ ./simple6.sh test_file
./simple6.sh:19:: '[' -z test_file ']'
./simple6.sh:26:: read LINE
./simple6.sh:27:: echo 'test_file: 10 rations of food'
test_file: 10 rations of food
./simple6.sh:26:: read LINE
./simple6.sh:27:: echo 'test_file: 2 blue potions'
test_file: 2 blue potions
./simple6.sh:26:: read LINE
./simple6.sh:27:: echo 'test_file: 1 scrol entitiles "exyc ahks"'
test_file: 1 scrol entitiles "exyc ahks"
./simple6.sh:26:: read LINE
./simple6.sh:30:: exit 0
gareth@brutha:~/Lec05$
```

- Input (User & File)
- Functions
- Good Practice
- An Aside - Compilation

Compiling

- Unlike a shell script, a C program needs to be compiled.
- Compilation means taking the human-readable source code and translating it into a set of instructions for a machine and operating system.
- Unlike a shell script, a compiled program runs independently of other programs. It does not require a shell to interpret it (c.f. python, ruby & perl).
- So far in lectures and labs you've seen something like:

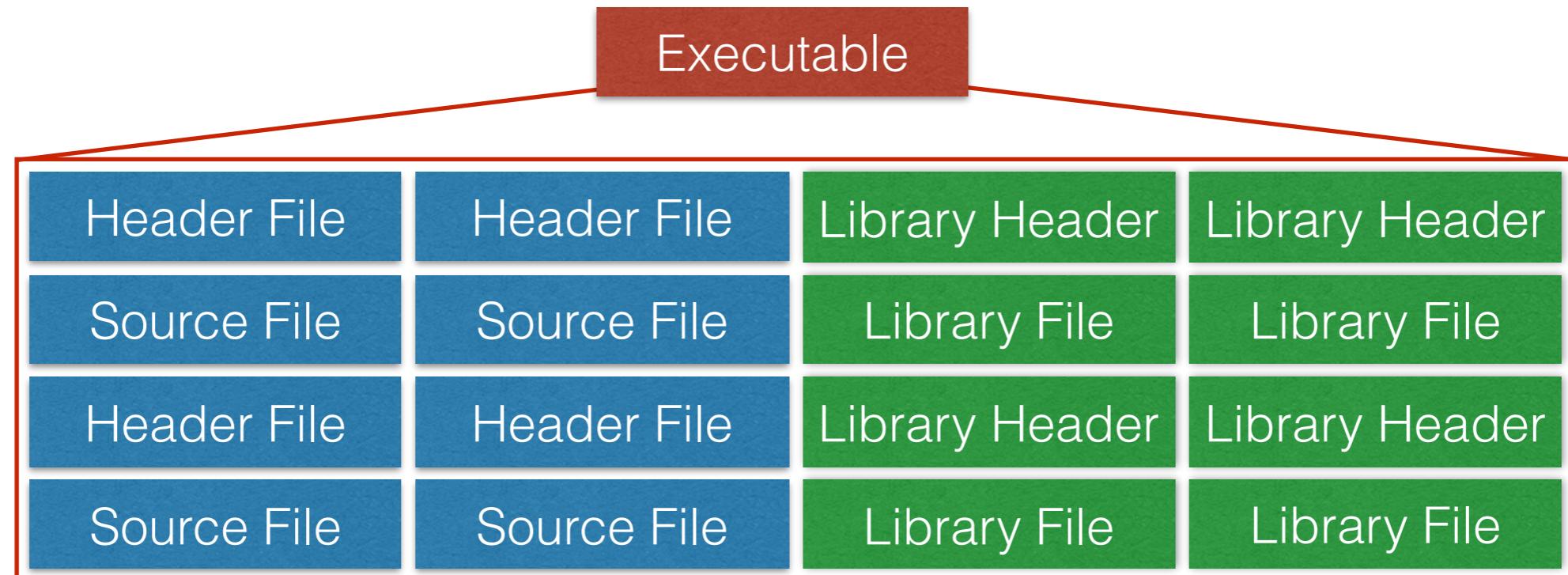
‣ **gcc -o loop loop.c**

The image shows two terminal windows. The top window, titled '2. gareth@brutha: ~/Lec05/C (ssh)', displays a C program named 'loop.c'. The code reads arguments from standard input and prints them one by one. The bottom window, titled '3. gareth@brutha: ~/Lec05/C (ssh)', shows the execution of the program. It first compiles 'loop.c' to an executable 'loop' using 'gcc -o loop loop.c'. Then it runs the executable with four arguments: 'a', 'b', 'c', and 'd'. The output shows the program printing each argument on a new line.

```
2. gareth@brutha: ~/Lec05/C (ssh)
1 #include <stdio.h>
2
3 /*
4  * A small program that reads arguments from stdin,
5  * loops over the arguments and prints them out one
6  * by one.
7  *
8  */
9
10 /* Main */
11 int main(int argc, char *argv[]) {
12     int i,sum,average;
13
14     /* Ensure we actually have some arguments */
15     if (1 == argc) {
16         printf("Arghhhh\n");
17         return 1;
18     }
19
20     /* Loop over each Argument and print to screen */
21     for (i=1; i < argc; i++) {
22         printf("Arg %d - %s\n",i,argv[i]);
23     }
24     return 0;
25 }
26
27
3. gareth@brutha: ~/Lec05/C (ssh)
gareth@brutha:~/Lec05/C$ gcc -o loop loop.c
gareth@brutha:~/Lec05/C$ ./loop
Arghhhh
gareth@brutha:~/Lec05/C$ ./loop a b c d
Arg 1 - a
Arg 2 - b
Arg 3 - c
Arg 4 - d
gareth@brutha:~/Lec05/C$
```

Components of a C program

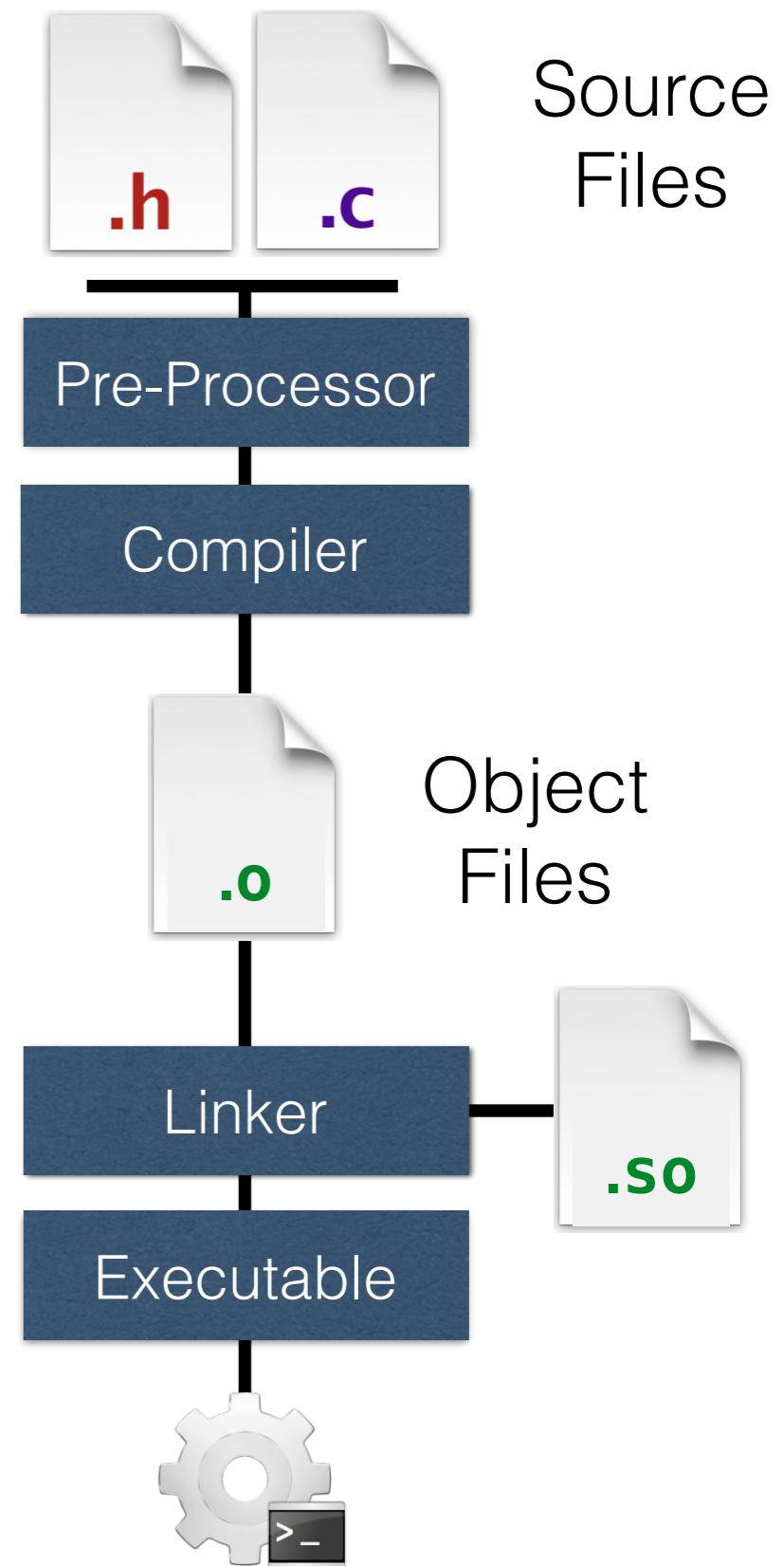
Header File	Definitions & Prototypes (*.h)
Source File	Our functions, etc (*.c)
Library Header	Library Definitions & Prototypes (*.h)
Library File	Pre-compiled Library Functions (*.a, *.so)



An executable is usually created from a number of source and header files (created by us and from libraries).

Compilation

- Compilation is actually three separate stages:
- **Pre-processor** - takes source and header files and combines the two, while expanding all #defines etc (see C lectures).
- **Compilation** - takes the preprocessed C files and convert them to machine code, label each function with a name (referred to as a symbol) and create an object file.
- **Linking** - takes all the object files, and match the symbols to actual machine code fragments. Place all fragments in the executable and replace the names with addresses.



GNU Compiler Collection

- So far you've been using `gcc` to compile your code.
- A more complete example of compiling a binary would use the following flags:
 - `-l` :: Library Name
 - `-L` :: Path to search for library code
 - `-I` :: Path to search for header files.
 - `-o` :: Executable name
- In this course you'll likely only be using standard system libraries which means `-L` and `-I` can be ignored as `gcc` will look in all the standard locations (including `.`) by default.
- `-c` can be used to only do pre-processing and compilation, and generates an object file.

```
gcc -l libraries  
-L path  
-I path  
-o name  
source.c
```

```
gcc -l mylib  
-L ./obj/  
-I ./include/  
-o myexe  
source1.c  
source2.c
```

Example

- We've created a small program that creates some random data and writes it to screen.
- Three source files:
 - **main.c** - creates an array of integers, calls a function to generate random data and prints it to the screen.
 - **util.h** - includes **time.h** and **stdlib.h** and has a function prototype.
 - **util.c** - has a function that takes an array and fills it with a series of random numbers.

```
2. gareth@brutha: ~/Lec05/C/multi (ssh)
1 #include <time.h>
2 #include <stdlib.h>
3
4 void generateRandomData(int data[], int len);
~ ~ ~
util.h 1,1 All
1 #include "util.h"
2
3 void generateRandomData(int data[], int len) {
4
5     int i;
6
7     srand(time(NULL));
8     for(i=0; i<len; i++) {
9         data[i] = rand() % 100;
10    }
11 }

util.c 1,1 All
1 #include <stdio.h>
2 #include "util.h"
3
4 #define SIZE_OF_DATA 10
5
6 int main(void) {
7     int data[SIZE_OF_DATA];
8
9     generateRandomData(data, SIZE_OF_DATA);
10
11    int i;
12    for(i=0; i<SIZE_OF_DATA; i++){
13        printf("%d: %d\n", i, data[i]);
14    }
15
16    return 0;
17 }

main.c [+] 17,1 All
-- INSERT --
```

Example

- Compile **util.c** into and object file:
 - **gcc -c util.c**
- Look at the contents of the object file with **nm**:
 - **nm util.o**
 - U means a symbol is unknown, while T means a symbol exist (in the Text/Code Segment)
- Now with **main.c**:
 - **gcc -c main.c**
 - **nm main.o**
- Create the executable **myprog**:
 - **gcc -o myprog main.o util.o**

```
2. gareth@brutha: ~/Lec05/C/multi (ssh)
gareth@brutha:~/Lec05/C/multi$ ls
main.c util.c util.h
gareth@brutha:~/Lec05/C/multi$ gcc -c util.c
gareth@brutha:~/Lec05/C/multi$ ls
main.c util.c util.h util.o
gareth@brutha:~/Lec05/C/multi$ nm util.o
0000000000000000 T generateRandomData
                      U rand
                      U srand
                      U time
gareth@brutha:~/Lec05/C/multi$ gcc -c main.c
gareth@brutha:~/Lec05/C/multi$ ls
main.c main.o util.c util.h util.o
gareth@brutha:~/Lec05/C/multi$ nm main.o
                      U generateRandomData
0000000000000000 T main
                      U printf
gareth@brutha:~/Lec05/C/multi$ gcc -o myprog main.o util.o
gareth@brutha:~/Lec05/C/multi$ ls
main.c main.o myprog util.c util.h util.o
gareth@brutha:~/Lec05/C/multi$ ./myprog
0: 8
1: 58
2: 76
3: 99
4: 80
5: 47
6: 82
7: 73
8: 0
9: 76
gareth@brutha:~/Lec05/C/multi$ 
```

Libraries (Dynamic and Static)

- We need to specify external libraries using the `-l` flag. This will search your **LD_LIBRARY_PATH** env variable or path specified by `-L`
- Linking to libraries can be done in two ways.
- By default **gcc** will link to a library dynamically. This means the library functions are not added in at compile time.
- Instead library functions are loaded when the program starts/runs.
- You can see what library's a program will load dynamically by using **ldd**.
- We can force all libraries functions to be included at compile time by using the **-static** keyword.

The screenshot shows a terminal window with the following session:

```
2. gareth@brutha: ~/Lec05/C/multi (ssh)
gareth@brutha:~/Lec05/C/multi$ clear
gareth@brutha:~/Lec05/C/multi$ ldd myprog
    linux-vdso.so.1 => (0x00007fff3780000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fcbbd3c0000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fcbbd7b0000)
gareth@brutha:~/Lec05/C/multi$ gcc -o myprogstatic -static *.o
gareth@brutha:~/Lec05/C/multi$ ls -ltrh
total 900K
-rw-rw-r-- 1 gareth gareth 86 Feb  9 13:29 util.h
-rw-rw-r-- 1 gareth gareth 238 Feb  9 14:07 main.c
-rw-rw-r-- 1 gareth gareth 152 Feb  9 14:07 util.c
-rw-rw-r-- 1 gareth gareth 1.6K Feb  9 14:07 util.o
-rw-rw-r-- 1 gareth gareth 1.6K Feb  9 14:07 main.o
-rwxrwxr-x 1 gareth gareth 8.5K Feb  9 14:49 myprog
-rwxrwxr-x 1 gareth gareth 865K Feb 10 08:42 myprogstatic
gareth@brutha:~/Lec05/C/multi$ ldd myprogstatic
    not a dynamic executable
gareth@brutha:~/Lec05/C/multi$ 
```

- Input (User & File)
- Functions
- Good Practice
- An Aside - Compilation



KEEP
CALM
AND
COMPILE
CODE

Makefiles & Git

Dr. Gareth Roy (x6439)
gareth.roy@glasgow.ac.uk

- Compilation
- Makefiles
- Revision Control
 - Git

- Compilation
- Makefiles
- Revision Control
 - Git

THE #1 PROGRAMMER EXCUSE
FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILING."

HEY! GET BACK
TO WORK!

COMPILING!

OH. CARRY ON.



Simple Example

- Last lecture we created a simple example project. It generated an array of random integers and wrote the contents of the array to screen.
- Three source files:
 - **main.c** - creates an array of integers, calls a function to generate random data and prints it to the screen.
 - **util.h** - includes **time.h** and **stdlib.h** and has a function prototype.
 - **util.c** - has a function that takes an array and fills it with a series of random numbers.

```
2. gareth@brutha: ~/Lec05/C/multi (ssh)
1 #include <time.h>
2 #include <stdlib.h>
3
4 void generateRandomData(int data[], int len);
~ ~ ~
util.h
1 #include "util.h"
2
3 void generateRandomData(int data[], int len) {
4
5     int i;
6
7     srand(time(NULL));
8     for(i=0; i<len; i++) {
9         data[i] = rand() % 100;
10    }
11 }

util.c
1 #include <stdio.h>
2 #include "util.h"
3
4 #define SIZE_OF_DATA 10
5
6 int main(void) {
7     int data[SIZE_OF_DATA];
8
9     generateRandomData(data, SIZE_OF_DATA);
10
11    int i;
12    for(i=0; i<SIZE_OF_DATA; i++){
13        printf("%d: %d\n", i, data[i]);
14    }
15
16    return 0;
17 }

main.c [+]
-- INSERT --
```

Compiling the Example

- We can compile all the code at once by doing:
 - **gcc -o myprog main.c util.c**
- Compiling the code this way is simple but it means we need to recompile **all** the code each time we build our program (on large projects this can be slow).
- Instead we can build in stages. To compile **util.c** into an object file:
 - **gcc -c util.c**
- Compile **main.c** into an object file:
 - **gcc -c main.c**
- Create the executable **myprog** by linking the object files together:
 - **gcc -o myprog main.o util.o**
- This has the benefit that when one file is changed only that file needs rebuilt into an object file, and the executable need remade.

```
1. gareth@brutha: ~/Lec06/multi (ssh)
gareth@brutha:~/Lec06/multi$ ls
main.c util.c util.h
gareth@brutha:~/Lec06/multi$ gcc -c util.c
gareth@brutha:~/Lec06/multi$ ls
main.c util.c util.h util.o
gareth@brutha:~/Lec06/multi$ gcc -c main.c
gareth@brutha:~/Lec06/multi$ ls
main.c main.o util.c util.h util.o
gareth@brutha:~/Lec06/multi$ gcc -o myprog main.o util.o
gareth@brutha:~/Lec06/multi$ ls
main.c main.o myprog util.c util.h util.o
gareth@brutha:~/Lec06/multi$ ./myprog
0: 92
1: 69
2: 62
3: 96
4: 47
5: 16
6: 93
7: 11
8: 32
9: 57
gareth@brutha:~/Lec06/multi$ 
```

Compiling the Example

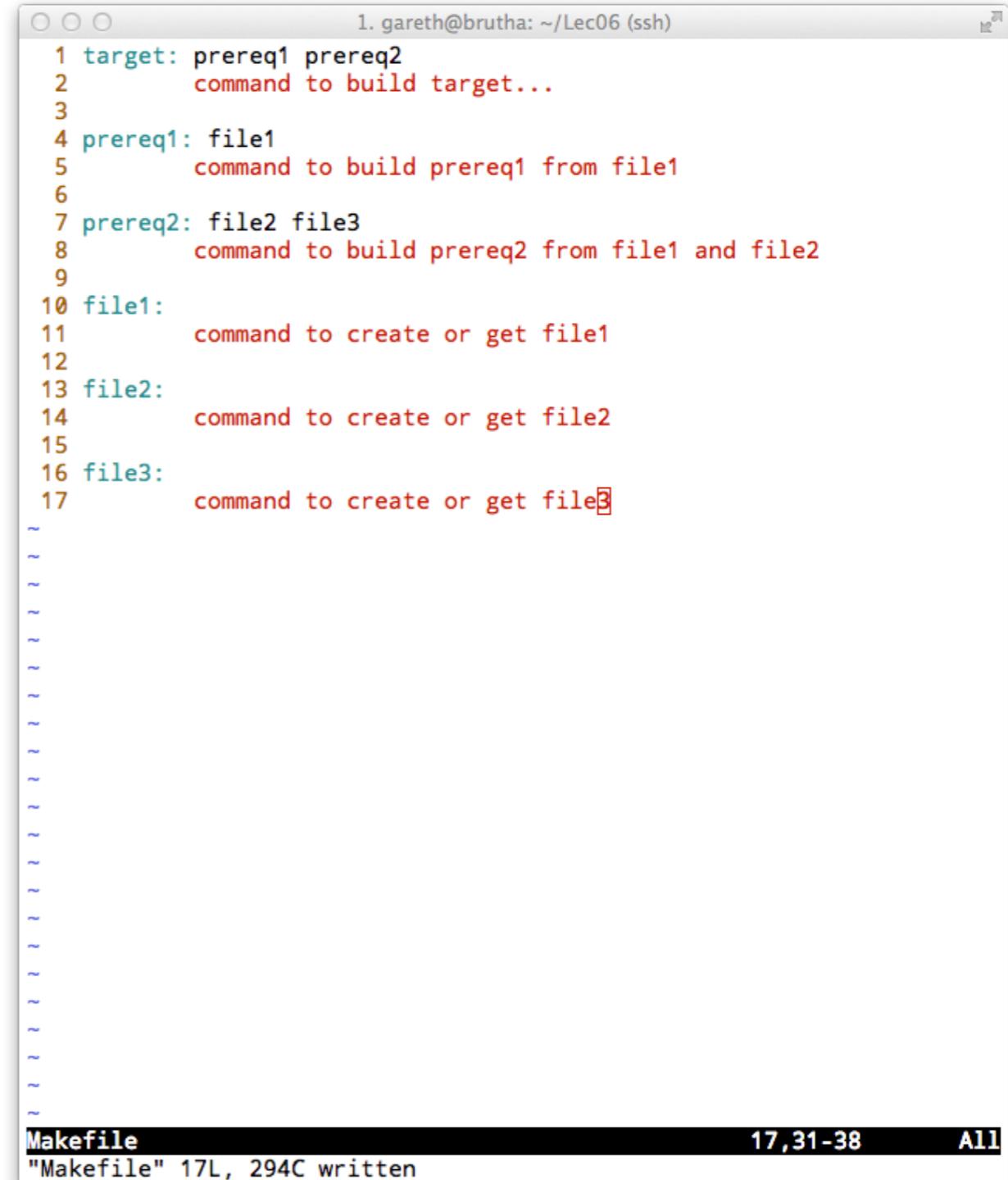
- The downside of incremental builds is the programmer is required to keep track of all of the files that change. When code bases grow to 1000's of files this can be a problem
- Importantly we need to ensure no stale object files are linked into the main program.
- Manually writing compile lines can be a pain, and it's often easy to forget to link libraries, use the same compilation flags etc.
- Using a script doesn't help us here as we want to ensure dependancies and make sure only the code that needs to be built is built - it's possible in bash but the script will get complicated quickly.
- We need a solution that does all of these things for us.

```
1. gareth@brutha: ~/Lec06/multi (ssh)
gareth@brutha:~/Lec06/multi$ ls
main.c util.c util.h
gareth@brutha:~/Lec06/multi$ gcc -c util.c
gareth@brutha:~/Lec06/multi$ ls
main.c util.c util.h util.o
gareth@brutha:~/Lec06/multi$ gcc -c main.c
gareth@brutha:~/Lec06/multi$ ls
main.c main.o util.c util.h util.o
gareth@brutha:~/Lec06/multi$ gcc -o myprog main.o util.o
gareth@brutha:~/Lec06/multi$ ls
main.c main.o myprog util.c util.h util.o
gareth@brutha:~/Lec06/multi$ ./myprog
0: 92
1: 69
2: 62
3: 96
4: 47
5: 16
6: 93
7: 11
8: 32
9: 57
gareth@brutha:~/Lec06/multi$ []
```

- Compilation
- Makefiles
- Revision Control
 - Git

Makefiles

- Makefiles are a way of automating the build process without losing the flexibility of only building object files.
- A **Makefile** can be thought of as a set of rules that explain how a piece of code is to be built.
- **make**, the command, then runs through each rule required to build a target ensuring they are met.
- By default make will look for a file called **makefile** or **Makefile**. You can specify a different file by:
 - **make -f myMakefile**
- Additionally you can run a specific rule by specifying a target i.e. **make target**
- Using Makefiles takes care off:
 - Incremental builds
 - Dependency tracking
 - Cleaning build files
 - Ensuring clean and consistent builds
 - Release vs. Production etc.

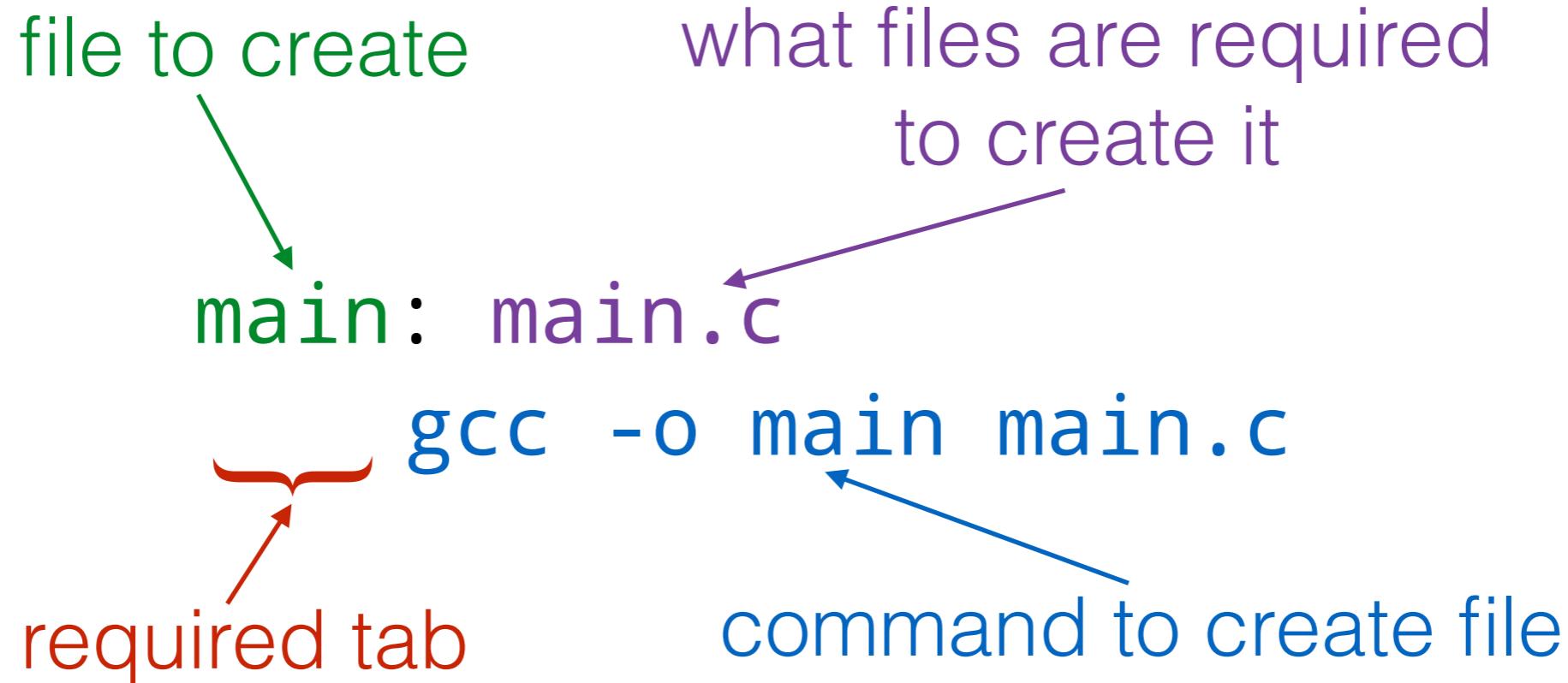


A screenshot of a terminal window titled "1. gareth@brutha: ~/Lec06 (ssh)". The window displays a Makefile script with the following content:

```
1 target: prereq1 prereq2
2         command to build target...
3
4 prereq1: file1
5         command to build prereq1 from file1
6
7 prereq2: file2 file3
8         command to build prereq2 from file1 and file2
9
10 file1:
11     command to create or get file1
12
13 file2:
14     command to create or get file2
15
16 file3:
17     command to create or get file3
```

The terminal window has a dark theme. The status bar at the bottom shows "Makefile" and "Makefile" 17L, 294C written. To the right of the status bar are the numbers "17,31-38" and "All".

Make Rules



- Makefile rules require a **target**, its **dependencies** and the **command** needed to produce the target from its dependencies. In this instance `main` is produced from `main.c` by running `gcc`.
- Makefiles can be used for a number of things, not just compiling. For instance the following rule downloads the xkcd comic you saw before:

`compiling.png:`

`wget http://imgs.xkcd.com/comics/compiling.png`

Simple Example

- Our simple program consisted of **main.c**, **util.c** and **util.h**
- Our simple Makefile consists of three rules:
 - A rule to make **util.o** from **util.c**
 - A rule to make **main.o** from **main.c**
 - A rule to make **myprog** from **main.o** and **util.o**
- We can compile our code by running **make** with no parameters. Make will try and ensure the first target in it's list exists (in this case **myprog**).
- If we run it a second time make tells us there is nothing to be done, and **myprog** is up to date.
- If we delete **main.o** and rerun **make**, **main.o** is rebuilt along with **myprog** but **util.o** is left untouched.

The screenshot shows two terminal windows. The top window displays a Makefile with the following content:

```
1 myprog: main.o util.o
2         gcc -o myprog main.o util.o
3
4 main.o: main.c
5         gcc -c main.c
6
7 util.o: util.c
8         gcc -c util.c
9 
```

The bottom window shows the execution of the Makefile:

```
gareth@brutha:~/Lec06/multi$ make
gcc -c main.c
gcc -c util.c
gcc -o myprog main.o util.o
gareth@brutha:~/Lec06/multi$ make
make: `myprog' is up to date.
gareth@brutha:~/Lec06/multi$ rm main.o
gareth@brutha:~/Lec06/multi$ make
gcc -c main.c
gcc -o myprog main.o util.o
gareth@brutha:~/Lec06/multi$ 
```

A small 'All' button is visible in the bottom right corner of the terminal window.

Variables

- Just like with a shell script you can assign variables which are expanded when executing commands
- Variable assignment works in the same way as Bash:
 - **VAR=value**
 - **\${VAR}**
 - **\$(VAR)**
- Referencing the variable again is similar to Bash:
- Variables are useful so that compile options can be consistent throughout the build. In our example we use **CC** to represent the compiler, **CFLAGS** to represent gcc compiler flags and **LDFLAGS** to represent linker flags (like loading math libraries **-lm**)

The image shows two terminal windows side-by-side. The top window, titled '1. gareth@brutha: ~/Lec06/multi/vars (ssh)', displays a Makefile with the following content:

```
1 # Comments in Make can look like this
2 CC=gcc
3 CFLAGS=-O2
4 LDFLAGS=-lm
5
6 # Debug flags, don't use for release
7 #CFLAGS=-g
8
9 myprog: main.o util.o
10      $(CC) $(LDFLAGS) -o myprog main.o util.o
11
12 main.o: main.c
13      $(CC) $(CFLAGS) -c main.c
14
15 util.o: util.c
16      $(CC) $(CFLAGS) -c util.c
17
```

The bottom window, titled '2. gareth@brutha: ~/Lec06/multi/vars (ssh)', shows the output of running 'make':

```
gareth@brutha:~/Lec06/multi/vars$ make
gcc -O2 -c main.c
gcc -O2 -c util.c
gcc -lm -o myprog main.o util.o
gareth@brutha:~/Lec06/multi/vars$
```

Targets and Phony

- Targets in Makefiles are usually files, for instance **myprog** or **main.c**
- By default **make** attempts to create the first target in a Makefile. For each dependency required **make** checks that these files exist and if not attempts to carry out the rule needed to create it.
- We can build a specific a different target by:
 - **make <target>**
 - **make myprog**
- Sometimes we want targets that don't create files. For instance a “**clean**” target that removes all of the build files so we can rebuild the whole project.
- Problems occur if there was a file present named the same as one of these targets as the rule would never run. For instance “**make clean**” would never remove the build files.
- For these instances we can use a **.PHONY** target which means the target will be run whenever specified.

The screenshot shows two terminal windows. The top window, titled "1. gareth@brutha: ~/Lec06/multi/vars (ssh)", displays a Makefile with the following content:

```
1 # Comments in Make can look like this
2 CC=gcc
3 CFLAGS=-O2
4 LDFLAGS=-lm
5 EXE=myprog
6
7 # Debug flags, don't use for release
8 #CFLAGS=-g
9
10 $(EXE): main.o util.o
11         $(CC) $(LDFLAGS) -o $(EXE) main.o util.o
12
13 main.o: main.c
14         $(CC) $(CFLAGS) -c main.c
15
16 util.o: util.c
17         $(CC) $(CFLAGS) -c util.c
18
19
20 .PHONY: clean
21 clean:
22         rm *.o
23         rm $(EXE)
```

The bottom window, titled "2. gareth@brutha: ~/Lec06/multi/vars (ssh)", shows the execution of the Makefile:

```
gareth@brutha:~/Lec06/multi/vars$ ls
main.c  main.o  Makefile  myprog  util.c  util.h  util.o
gareth@brutha:~/Lec06/multi/vars$ make clean
rm *.o
rm myprog
gareth@brutha:~/Lec06/multi/vars$ ls
main.c  Makefile  util.c  util.h
gareth@brutha:~/Lec06/multi/vars$ touch clean
gareth@brutha:~/Lec06/multi/vars$ make
gcc -O2 -c main.c
gcc -O2 -c util.c
gcc -lm -o myprog main.o util.o
gareth@brutha:~/Lec06/multi/vars$ make clean
rm *.o
rm myprog
gareth@brutha:~/Lec06/multi/vars$ ls
clean  main.c  Makefile  util.c  util.h
gareth@brutha:~/Lec06/multi/vars$ rm clean
gareth@brutha:~/Lec06/multi/vars$
```

Automatic Variables

- Make has a number of builtin automatic variables that help to make our Makefiles simpler.
 - **%** is analogous to ***** in a shell script. It matches anything but with a twist. It will not match NULL and it temporarily knows what it matched.
Hence: **%.o:** **%.c** means “any file ending in ‘.o’ depends upon a similarly-named ‘.c’ file.
 - **\$@** is the name of the target
 - **\$<** is the name of the first prerequisite
 - **\$^** is the list of prerequisites for the target you are compiling...more-or-less.
 - **\$?** is the list of prerequisites that are newer than the target and therefore must be compiled first.
- See: <http://www.gnu.org/software/automake/manual/make/> for a full manual...

The image shows two terminal windows. The top window, titled "1. gareth@brutha: ~/Lec06/multi/autovars (ssh)", displays a Makefile with syntax highlighting for comments, variables, and rules. The bottom window, titled "2. gareth@brutha: ~/Lec06/multi/autovars (ssh)", shows the output of running 'make' followed by the execution of the generated executable.

```
1 # Comments in Make can look like this
2 CC=gcc
3 EXE=myprog
4
5 CFLAGS=-O2
6 LDFLAGS=-lm
7 OBJ=main.o util.o
8
9 # Build the EXE specified by $(EXE)
10 $(EXE): $(OBJ)
11     $(CC) $(LDFLAGS) -o $@ $^
12
13 #Build all source files into object files
14 %.o:%.c
15     $(CC) $(CFLAGS) -c $<
16
17
18 .PHONY: clean test
19 # Clean all build files
20 clean:
21     rm *.o
22     rm $(EXE)
23
24 #Run the executable
25 test:
26     ./$(EXE)
```

```
gareth@brutha:~/Lec06/multi/autovars$ make
gcc -O2 -c main.c
gcc -O2 -c util.c
gcc -lm -o myprog main.o util.o
gareth@brutha:~/Lec06/multi/autovars$ make test
./myprog
0: 49
1: 85
2: 52
3: 51
4: 25
5: 80
6: 47
7: 19
8: 14
9: 78
gareth@brutha:~/Lec06/multi/autovars$
```

More Advanced Make

- Make allows targets to reference one another as dependencies.
- This can be used to build up more complicated interactions.
- For instance say you want to be able to use the same makefile to build an optimised build, or a debug build (or a release build).
- You could create targets that modified the flags passed to the build step so that different compiler options were set.
- The example Makefile add the contents of **OPTCFLAGS** to **CFLAGS** if the build is optimised and **DEBUGCFLAGS** to **CFLAGS** if the build is a debug.

The image shows two terminal windows. The top window displays a Makefile with numbered lines. The bottom window shows the execution of the Makefile with various build commands.

Makefile Content (Top Window):

```
1 # Comments in Make can look like this
2 CC=gcc
3 EXE=myprog
4
5 CFLAGS=-Wall
6 LDFLAGS=-lm
7
8 OPTCFLAGS=-O2
9 DEBUGCFLAGS=-g
10
11 OBJ=main.o util.o
12
13 # Default Target, dependency on $(EXE) target
14 all: $(EXE)
15
16 # Optimised target, add OPTCFLAGS
17 opt: CFLAGS+=${OPTCFLAGS}
18 opt: $(EXE)
19
20 # Debug target, add DEBUG Flags
21 debug: CFLAGS+=${DEBUGCFLAGS}
22 debug: $(EXE)
23
24 $(EXE): $(OBJ)
25     $(CC) $(LDFLAGS) -o $@ $^
26
27 %.o:%.c
28     $(CC) $(CFLAGS) -c $<
29
30 .PHONY: clean test
31 clean:
32     rm *.o
33     rm $(EXE)
```

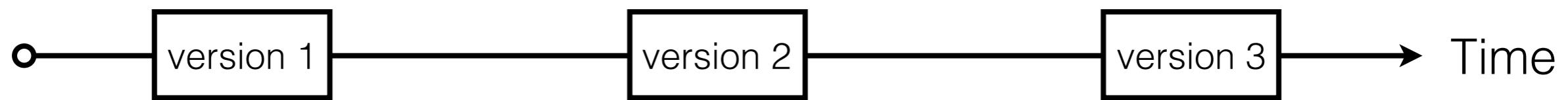
Terminal Execution (Bottom Window):

```
gareth@brutha:~/Lec06/multi/advanced$ make debug
gcc -Wall -g -c main.c
gcc -Wall -g -c util.c
gcc -lm -o myprog main.o util.o
gareth@brutha:~/Lec06/multi/advanced$ make clean
rm *.o
rm myprog
gareth@brutha:~/Lec06/multi/advanced$ make opt
gcc -Wall -O2 -c main.c
gcc -Wall -O2 -c util.c
gcc -lm -o myprog main.o util.o
gareth@brutha:~/Lec06/multi/advanced$ make clean
rm *.o
rm myprog
gareth@brutha:~/Lec06/multi/advanced$ make
gcc -Wall -c main.c
gcc -Wall -c util.c
gcc -lm -o myprog main.o util.o
gareth@brutha:~/Lec06/multi/advanced$
```

- Compilation
- Makefiles
- Revision Control
 - Git

What is Revision Control?

- Files, such as C source files, change over time as modifications are made.
- Often, over a long period of time, the reason for the changes that have been made is lost. This is particularly difficult when a file is worked on by multiple people.
- Revision control (also known as Version control, Change control and Source Code management) is a method for managing the changes to a file, and maintaining a history of the changes that have taken place.



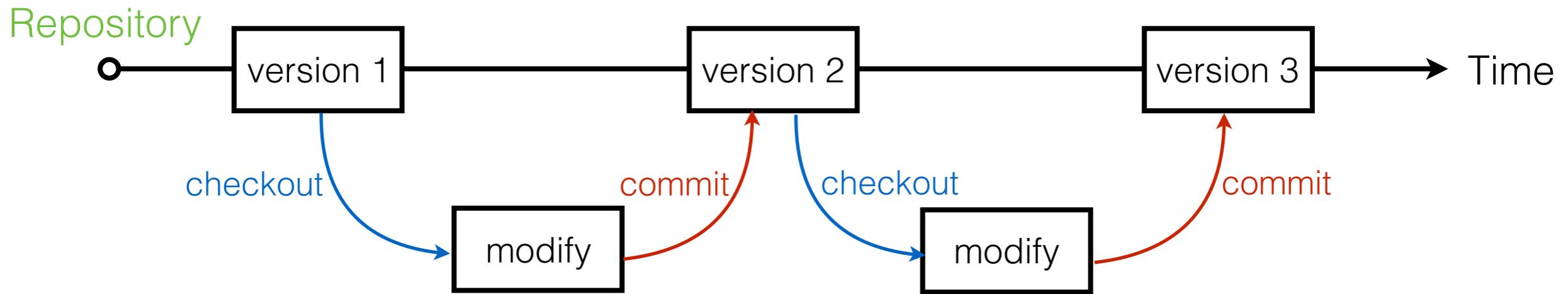
```
1. gareth@brutha: ~/Lec06/revision (ssh)
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     if (1 == argc) {
5         printf("Arghhhh\n");
6         return 1;
7     }
8
9     return 0;
10}
11
12
13
14

1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     if (1 == argc) {
5         printf("Arghhhh\n");
6         return 1;
7     }
8     for (i=1; i < argc; i++) {
9         printf("Arg %d - %s\n", i, argv[i]);
10    }
11
12 }
13
14

1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     if (1 == argc) {
5         printf("Arghhhh\n");
6         return 1;
7     }
8     for (i=1; i < argc; i++) {
9         printf("Arg %d - %s\n", i, argv[i]);
10    }
11
12 }
13
14

loop1.c      5,0-1   All  loop2.c      11,0-1  All  loop3.c      14,0-1  All
```

The Basics of Revision Control



Repository: Location of stored files, and file changes

Checkout: Get a complete version of the code from the repository

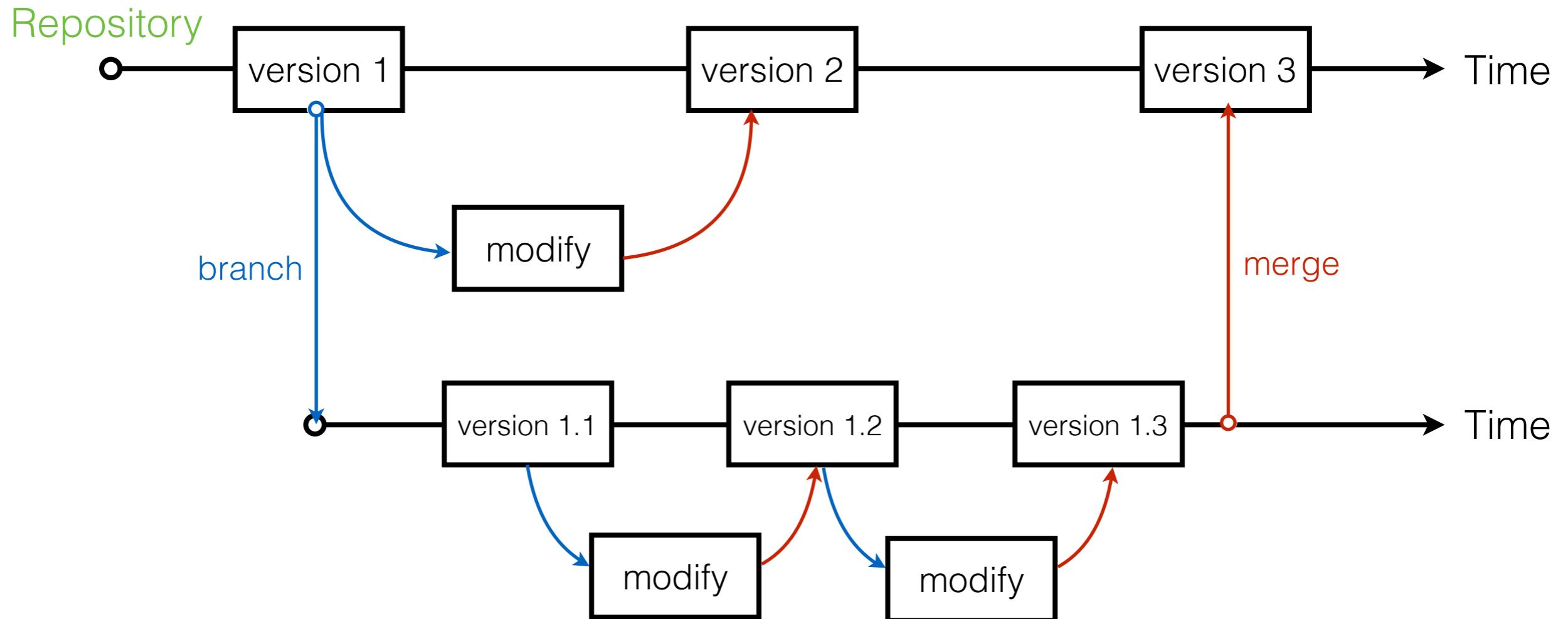
Commit: Send an updated version of the code to the repository
(Along with some message/log)

```
1. gareth@brutha: ~/Lec06/revision (ssh)
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     if (1 == argc) {
5         printf("Arghhhh\n");
6         return 1;
7     }
8
9     return 0;
10}
11
12}

1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     if (1 == argc) {
5         printf("Arghhhh\n");
6         return 1;
7     }
8     for (i=1; i < argc; i++) {
9         printf("Arg %d - %s\n", i, argv[i]);
10    }
11
12}
13
14

loop1.c      5,0-1   All loop2.c      11,0-1  All loop3.c      14,0-1  All
```

Branch & Merge

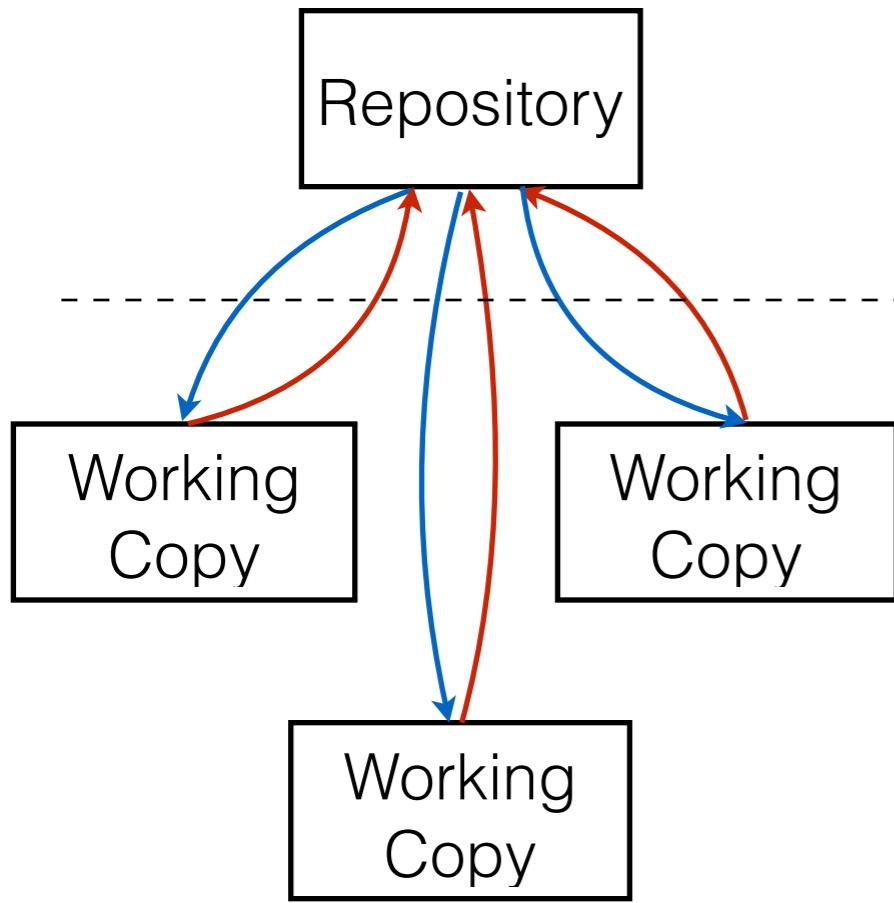


Another key concept is branching and merging.

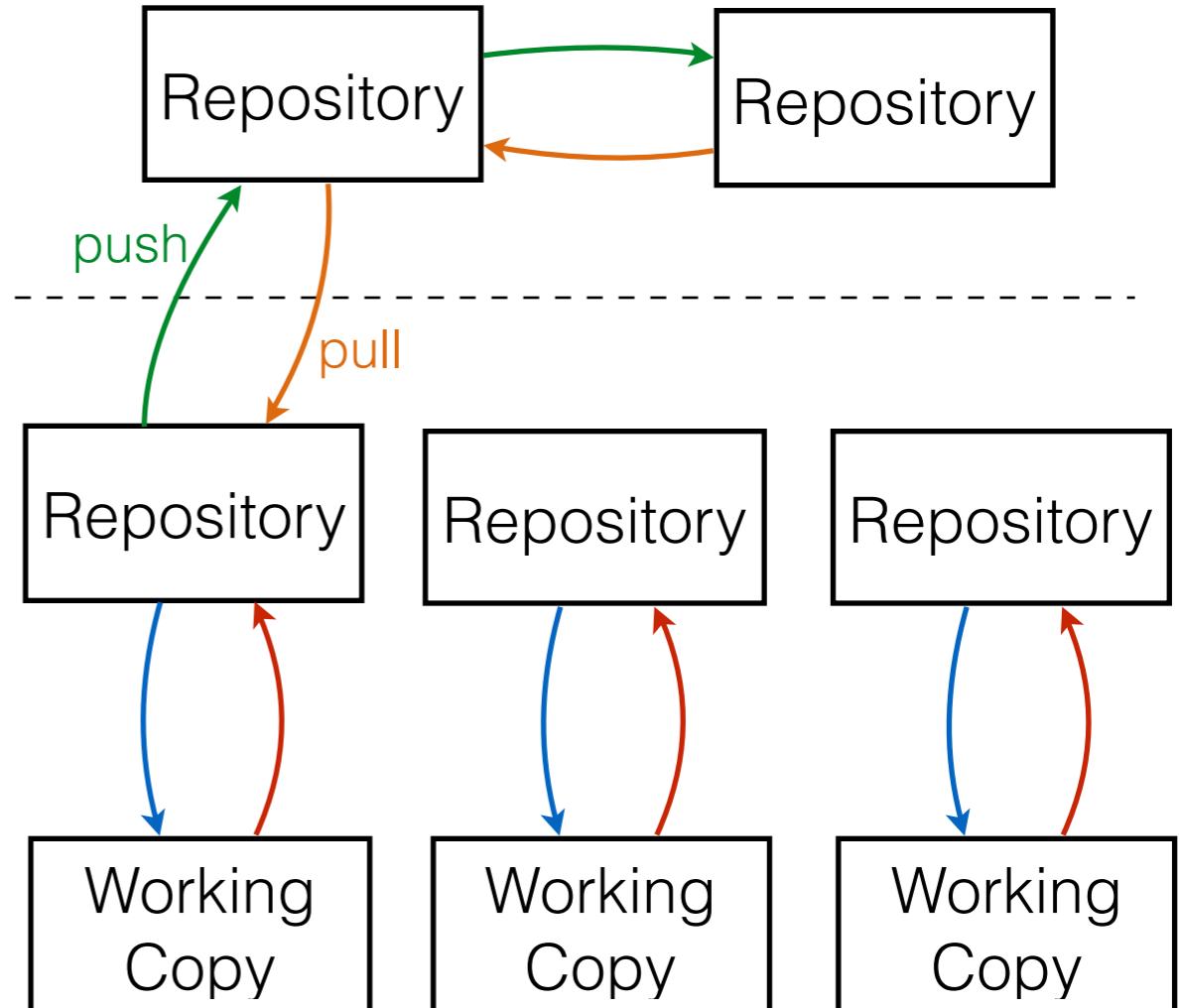
Branch: Create a complete copy of the code with its own revision history.

Merge: Take all the changes that have occurred within a branch and merge them (including the history).

Distributed vs. Centralised



Server
Client



Centralised

- All history, tags, branches are kept on the server.
- Only one canonical source for all code.
- Simple access controls, and user management.

Distributed

- Each client has a complete copy of the history.
- Easy for offline work, simple syncing mechanisms.
- No single canonical source for all changes.

- Compilation
- Makefiles
- Revision Control
 - Git

Git

- **git** is a distributed revision control system developed by Linus Torvalds in 2005 to manage the development of the Linux Kernel.
- Due to it's distributed nature it became popular amongst the open source community.
- Lots of online resources are available as well as free online hosting of repositories.
- Two of the biggest sites are:
 - github.com - unlimited public repositories, need to pay for private, “social coding”.
 - bitbucket.org - unlimited private repositories, tools not quite so good as github.
- Both of these sites have lots of links to tutorials and training material. if your interested have a look!

The image displays two side-by-side screenshots of Git hosting websites. On the left is the Bitbucket interface, showing a team named 'Gla-Phys-P2T' (glaphysp2t) created in April 2014. It lists four repositories: 'C Labs' (updated 2015-01-12), 'lab5-example' (updated 2015-01-09), 'C Lectures' (updated 2014-07-11), and 'C Examples' (updated 2014-06-02). A sidebar on the right shows a 'Teams connect with HipChat!' integration and recent activity. On the right is the GitHub interface, specifically the repository 'torvalds / linux'. This page shows the Linux kernel source tree with 503,468 commits, 1 branch, 406 releases, and 4,681 contributors. The main area lists commit history for various kernel submodules like Documentation, arch, block, crypto, drivers, firmware, fs, include, init, ipc, kernel, lib, mm, net, samples, scripts, security, sound, tools, usr, virt/kvm, and .gitignore. GitHub features like 'Code', 'Pull Requests', 'Pulse', 'Graphs', and download options are visible on the right.

Git - Getting Started

- A git repository can be created in two ways, we can:
 - 1) create an new, empty repository
 - 2) copy/clone an existing repository (i.e. pull down a complete copy)
- To create a new, empty repository we can do:
 - **git init <directory>**
 - **git init .**
- All of the files that git uses to manage your repository are located in a special hidden directory call **.git** in your project folder.
- To copy an exiting project, we need a link to it's location (for instance in the lab you'll be asked to):
 - **git clone https://bitbucket.org/glaphysp2t/lab5-example.git**
- This will copy down a complete version of the code including it's entire history.

```
1. gareth@brutha: ~/Lec06/git (ssh)
gareth@brutha:~/Lec06/git$ git init proj
Initialized empty Git repository in /home/gareth/Lec06/git/proj/.git/
gareth@brutha:~/Lec06/git$ tree
.
└── proj

1 directory, 0 files
gareth@brutha:~/Lec06/git$ tree -a
.
└── proj
    └── .git
        ├── branches
        ├── config
        ├── description
        ├── HEAD
        ├── hooks
        │   ├── applypatch-msg.sample
        │   ├── commit-msg.sample
        │   ├── post-update.sample
        │   ├── pre-applypatch.sample
        │   ├── pre-commit.sample
        │   ├── prepare-commit-msg.sample
        │   ├── pre-rebase.sample
        │   └── update.sample
        ├── info
        │   └── exclude
        ├── objects
        │   ├── info
        │   └── pack
        └── refs
            ├── heads
            └── tags

3. gareth@brutha: ~/Lec06/git/lab5-example (ssh)
gareth@brutha:~/Lec06/git$ git clone https://bitbucket.org/glaphysp2t/
lab5-example.git
Cloning into 'lab5-example'...
remote: Counting objects: 33, done.
remote: Compressing objects: 100% (31/31), done.
remote: Total 33 (delta 11), reused 0 (delta 0)
Unpacking objects: 100% (33/33), done.
gareth@brutha:~/Lec06/git$ ls
lab5-example  notes  proj
gareth@brutha:~/Lec06/git$ cd lab5-example/
gareth@brutha:~/Lec06/git/lab5-example$ ls
draw.c  draw.h  histogram.c  makefile  README.md  util.c  util.h
gareth@brutha:~/Lec06/git/lab5-example$
```

Git - Adding & Committing

- Now that we have a repository we'd like to add some files to our repository.
- To do this we use the “**git add**” command.

- **git add <filename>**

- It's important to note that this only stages the file to be added, but hasn't actually added it yet.
- To make any change to the repository we must save the action, or commit it.**

- To do this we use the “**git commit**” command, and supply it a message (usually a reason for what has added or has been changed)

- **git commit -m “Adding some files”**

- If the “**-m**” parameter is not specified, the commit log will be opened in whatever editor is specified in \$EDITOR or \$VISUAL.
- You can remove a file from the repository using the “**git rm**” command.

- **git rm <filename>**

- As with **git add** changes don't take effect until you commit them.

```
1. gareth@brutha: ~/Lec06/git/proj (ssh)
gareth@brutha:~/Lec06/git/proj$ touch test.c
gareth@brutha:~/Lec06/git/proj$ git add -v test.c
add 'test.c'
gareth@brutha:~/Lec06/git/proj$ git commit -m "Added a file"
[master (root-commit) 7ea05b6] Added a file
 0 files changed
  create mode 100644 test.c
gareth@brutha:~/Lec06/git/proj$ git rm test.c
rm 'test.c'
gareth@brutha:~/Lec06/git/proj$ git commit -m "Removed a file"
[master 48c8921] Removed a file
 0 files changed
  delete mode 100644 test.c
gareth@brutha:~/Lec06/git/proj$ ls
gareth@brutha:~/Lec06/git/proj$ █
```

Git - Status

- It's often difficult to see the status of all the files we're working on.
- Often we've created new files, modified some and deleted some.
- We can check the status of all the files in the repository using:

- **git status**

- “**git status**” will tell us about files that are present but not part of the repository, files that have been added and not committed, files that have been modified or files that have been deleted.
- It will omit any files that are unchanged, which are included in the repository.
- If the output is overly verbose we can get a short description by using:

- **git status -s**

```
1. gareth@brutha: ~/Lec06/git/proj (ssh)
gareth@brutha:~/Lec06/git/proj$ touch test.c
gareth@brutha:~/Lec06/git/proj$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       test.c
nothing added to commit but untracked files present (use "git add" to track)
gareth@brutha:~/Lec06/git/proj$ git add test.c
gareth@brutha:~/Lec06/git/proj$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   test.c
#
gareth@brutha:~/Lec06/git/proj$ git commit -m "Added a file"
[master f029e5a] Added a file
 0 files changed
  create mode 100644 test.c
gareth@brutha:~/Lec06/git/proj$ git status
# On branch master
nothing to commit (working directory clean)
gareth@brutha:~/Lec06/git/proj$ rm test.c
gareth@brutha:~/Lec06/git/proj$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    test.c
#
no changes added to commit (use "git add" and/or "git commit -a")
gareth@brutha:~/Lec06/git/proj$ git status -s
 D test.c
gareth@brutha:~/Lec06/git/proj$ 
```

Git - Adding & Committing (2)

- If we modify a file and want to commit the changes we need to use the “**git add**” command again:

- **git add <filename>**

- The file is now tagged as modified in the output of “**git status**”
- Once again the changes aren’t saved in the repository until we commit them:

- **git commit -m “Modified a file”**

- With modified files we can use a short cut and add, commit them at the same time using:

- **git commit -a -m “Commit and Add”**

- **Note:** This only works for modified files, not files that are not in the repository already.

```
1. gareth@brutha: ~/Lec06/git/proj (ssh)
gareth@brutha:~/Lec06/git/proj$ git status -s
gareth@brutha:~/Lec06/git/proj$ echo Hello > test.c
gareth@brutha:~/Lec06/git/proj$ git status -s
M test.c
gareth@brutha:~/Lec06/git/proj$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   test.c
#
no changes added to commit (use "git add" and/or "git commit -a")
gareth@brutha:~/Lec06/git/proj$ git add test.c
gareth@brutha:~/Lec06/git/proj$ git status -s
M test.c
gareth@brutha:~/Lec06/git/proj$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   test.c
#
gareth@brutha:~/Lec06/git/proj$ git commit -m "Modified test.c"
[master 6cb577d] Modified test.c
 1 file changed, 1 insertion(+)
gareth@brutha:~/Lec06/git/proj$ echo Hello2 >> test.c
gareth@brutha:~/Lec06/git/proj$ git status -s
M test.c
gareth@brutha:~/Lec06/git/proj$ git commit -a -m "Committed and added"
[master c10555f] Committed and added
 1 file changed, 1 insertion(+)
gareth@brutha:~/Lec06/git/proj$ git status -s
gareth@brutha:~/Lec06/git/proj$ █
```

Git - Log

- To see a log of all the commits, and the messages associated with them we can use the “**git log**” command.

- › **git log**

- By default “**git log**” will output:

- the full hash for the commit (it's ID).
 - the full commit message.
 - who performed the committed
 - when the commit took place.
- This can be quite verbose so we can simplify this by doing:

- › **git log --oneline**

```
1. gareth@brutha: ~/Lec06/git/proj (ssh)
gareth@brutha:~/Lec06/git/proj$ git log
commit c805352a3edca2214321b5ce00aa8f870ed70a06
Author: Gareth Roy <gareth.roy@glasgow.ac.uk>
Date: Tue Feb 17 11:19:59 2015 +0000

    Deleted another file

commit f029e5aacf05b1e070d5a924ce61dc3ab084bfa7
Author: Gareth Roy <gareth.roy@glasgow.ac.uk>
Date: Tue Feb 17 11:10:47 2015 +0000

    Added a file

commit 48c89211b8bc858a0727ff98a18587f234348003
Author: Gareth Roy <gareth.roy@glasgow.ac.uk>
Date: Tue Feb 17 11:06:56 2015 +0000

    Removed a file

commit 7ea05b6cbb2934604fc0a9ef00b6366c7fe0b36e
Author: Gareth Roy <gareth.roy@glasgow.ac.uk>
Date: Tue Feb 17 11:05:59 2015 +0000

    Added a file
gareth@brutha:~/Lec06/git/proj$ git log --oneline
c805352 Deleted another file
f029e5a Added a file
48c8921 Removed a file
7ea05b6 Added a file
gareth@brutha:~/Lec06/git/proj$ 
```

```
2. gareth@brutha: ~/Lec06/git/lab5-example (ssh)
gareth@brutha:~/Lec06/git/lab5-example$ git log --oneline
8fbdb35f Refactored to move utility code out of the main file
e532f4f Added randomly generated data
6dcdf5 Refactored code so that the drawing routines are in their own file, added to the makefile
6b4875c Added a simple makefile to build histogram, included a make test which runs the program
8b2c746 Added a function that calculates the maximum of a list of data and returns its value
66ee505 Added in a simple axis to the histogram
c496941 Refactored to create a printBar function
2a65f62 A simple program that prints out an ASCII histogram based on a test set of integer data
d71c39b Added an initial README to the project
gareth@brutha:~/Lec06/git/lab5-example$ 
```

Git - Log

```
Folkvangr:CLabs-Tex gareth$ git log --oneline --graph --decorate
* dfb76d6 (HEAD, origin/master, master) Removed more questions from labs
* 3e037f8 Removed some questions
* 14b61b3 Removed some questions
* d9768f0 Draft changes to labs SCS
* 1170b7a First fixed revision, lab01 SCS
* e96d985 Merge branch 'master' of bitbucket.org:glaphysp2t/c-labs
  \
  * 03fbab5 Merge branch 'master' of bitbucket.org:glaphysp2t/c-labs
    \
    * aeaac3a Merge branch 'master' of bitbucket.org:glaphysp2t/c-labs
      \
      * b404d78 Tidied up some of the text - GDR
      * b0210c1 lab6: challenging question added
        \
        * b27b03f lab5: minor changes
          \
          * a69e7a0 lab6: minor corrections
          * 840609e lab6: minor corrections
          * b58571d lab6: background theory added
            \
            * 9b5392e minor changes
            * d8e4004 lab5: background theory/information added
            * 476a117 Added loop examples
            * 1e3da6c Merge branch 'master' of bitbucket.org:glaphysp2t/c-labs
              \
              * 6ca1227 lab3: minor corrections
              * 3e88faa Updated and re-worked lab02 - GDR
                \
                * 0c46ce9 lab4: description of functions
                * 268c124 lab1: expanded on precedence
                * 307388f lab2: minor correction
                * 32d36ce lab3: background theory/information
                * 9246f34 lab1: char type decription modified
                * fbb77a5 lab1: modified puts() and printf()
                * 07d3346 lab1: question 2 modified
                * 1378206 lab1: question 1 merged with question 4
                * f217d69 Merge branch 'master' of bitbucket.org:glaphysp2t/c-labs
      \

```

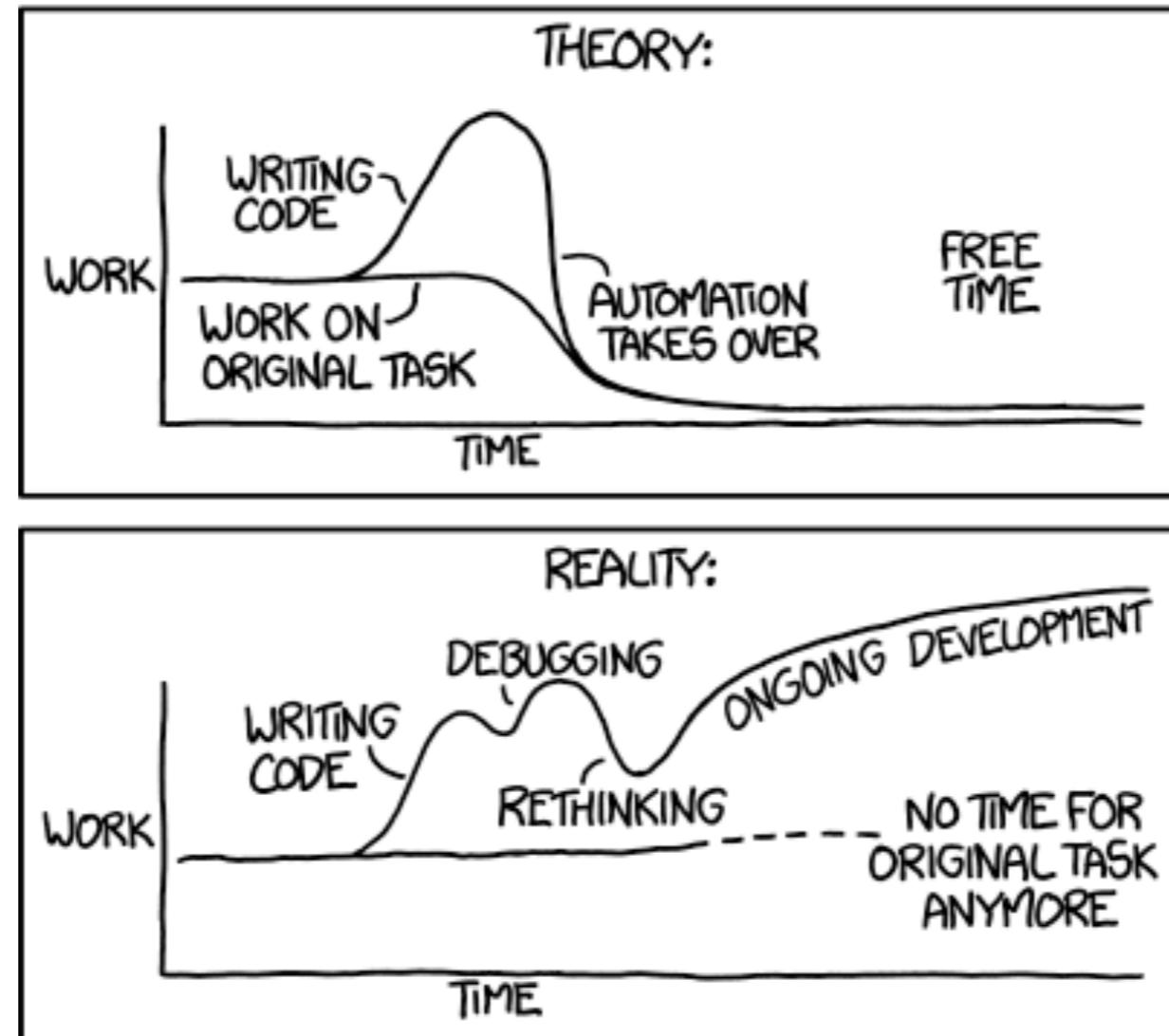
Git - Branch and Merge

- To create a branch we do the following:
 - **git branch <branch>**
- We can get a list of all the branches in our repository by doing:
 - **git branch --list**
- We can delete an unneeded branch by doing:
 - **git branch -d <branch>**
- To change into a branch we have to “checkout” that branch into our working directory:
 - **git checkout <branch>**
- We can create a new branch and change into it all at once doing:
 - **git checkout -b <branch>**
- Finally we can merge a branch back into the current branch by doing:
 - **git merge <branch>**

```
1. gareth@brutha: ~/Lec06/git/proj (ssh)
gareth@brutha:~/Lec06/git/proj$ git branch myfeature
gareth@brutha:~/Lec06/git/proj$ git branch --list
* master
  myfeature
gareth@brutha:~/Lec06/git/proj$ git checkout myfeature
Switched to branch 'myfeature'
gareth@brutha:~/Lec06/git/proj$ git branch --list
  master
* myfeature
gareth@brutha:~/Lec06/git/proj$ cat test.c
Hello
Hello2
gareth@brutha:~/Lec06/git/proj$ echo Hello3 >> test.c
gareth@brutha:~/Lec06/git/proj$ git commit -a -m "Test"
[myfeature c985651] Test
  1 file changed, 1 insertion(+)
gareth@brutha:~/Lec06/git/proj$ cat test.c
Hello
Hello2
Hello3
gareth@brutha:~/Lec06/git/proj$ git checkout master
Switched to branch 'master'
gareth@brutha:~/Lec06/git/proj$ cat test.c
Hello
Hello2
gareth@brutha:~/Lec06/git/proj$ git merge myfeature
Updating 2ae35bd..c985651
Fast-forward
  test.c |  1 +
  1 file changed, 1 insertion(+)
gareth@brutha:~/Lec06/git/proj$ cat test.c
Hello
Hello2
Hello3
gareth@brutha:~/Lec06/git/proj$ git branch --list
* master
  myfeature
gareth@brutha:~/Lec06/git/proj$ git branch -d myfeature
Deleted branch myfeature (was c985651).
gareth@brutha:~/Lec06/git/proj$ 
```

- Compilation
- Makefiles
- Revision Control
 - Git

"I SPEND A LOT OF TIME ON THIS TASK.
I SHOULD WRITE A PROGRAM AUTOMATING IT!"



GDB & Debugging

Dr. Gareth Roy (x6439)
gareth.roy@glasgow.ac.uk

- Getting started
- Running GDB
- Interacting with our Code

- Getting started
- Running GDB
- Interacting with our Code

Simple Example

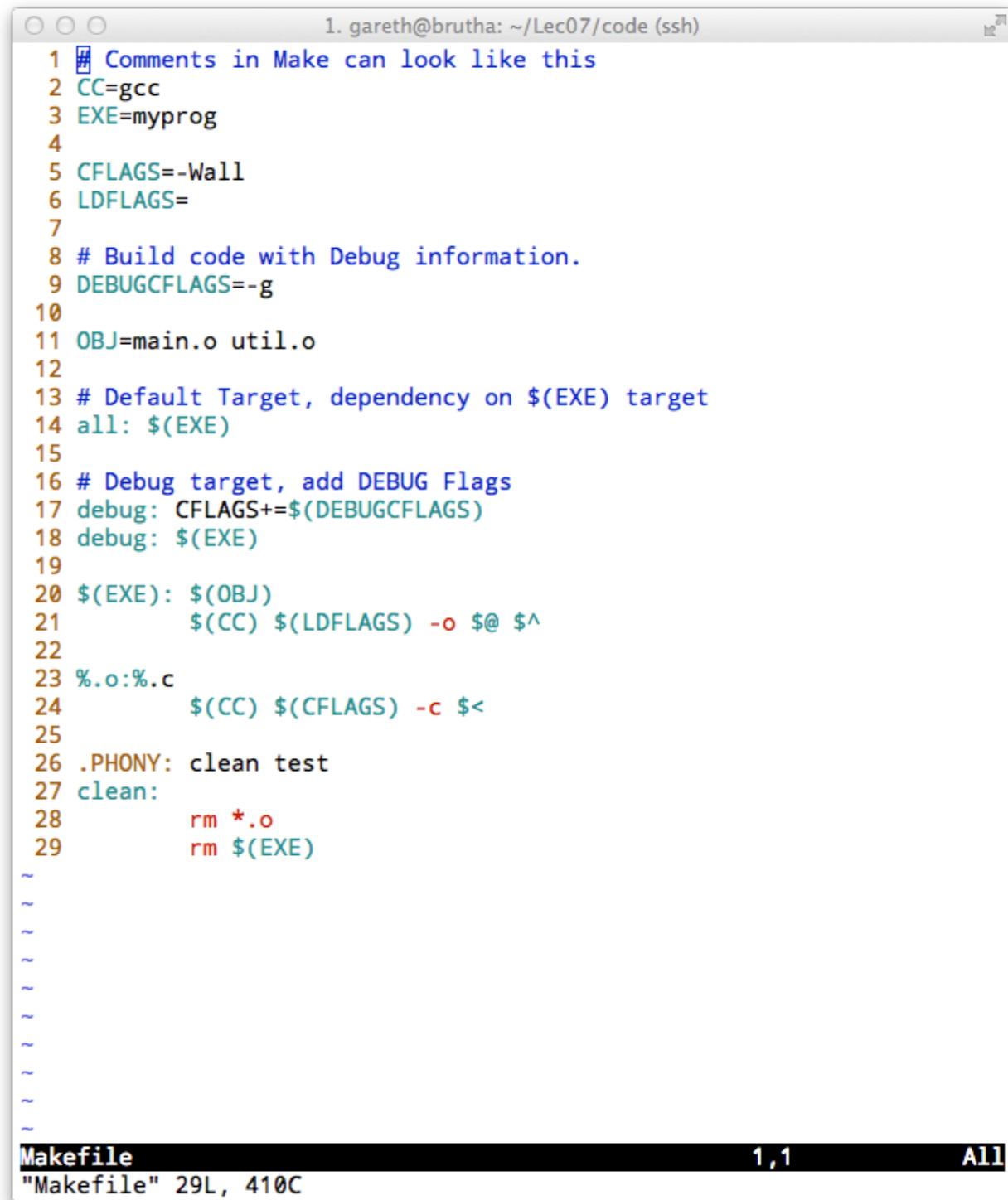
- We'll use the same code example to experiment with GDB.
- For completeness it consists of three source files:
 - **main.c** - creates an array of integers, calls a function to generate random data and prints it to the screen.
 - **util.h** - includes **time.h** and **stdlib.h** and has a function prototype.
 - **util.c** - has a function that takes an array and fills it with a series of random numbers.

The screenshot shows a terminal window with three tabs open, each displaying a source code file:

- util.h**: Includes `<time.h>` and `<stdlib.h>`. It contains a function prototype `void generateRandomData(int data[], int len);`.
- util.c**: Includes `<stdio.h>` and `"util.h"`. It defines a constant `SIZE_OF_DATA` and a `main` function that generates random data and prints it to the console.
- main.c**: Includes `"util.h"`. It calls the `generateRandomData` function and prints the generated data.

Adding Debug Information

- To our small project we've added a makefile.
- As part of our makefile we've added a target called **debug**.
- This target allows us create a version of our code with debug information.
- In order to use a debugger we have to insert special information into the object files so it can trace the execution of the program.
- In **gcc** to compile code which adds support for debugging we must specify the **-g** flag.
- A one-line compilation would look like:
 - **gcc -g -o myprog *.c**



```
1 # Comments in Make can look like this
2 CC=gcc
3 EXE=myprog
4
5 CFLAGS=-Wall
6 LDFLAGS=
7
8 # Build code with Debug information.
9 DEBUGCFLAGS=-g
10
11 OBJ=main.o util.o
12
13 # Default Target, dependency on $(EXE) target
14 all: $(EXE)
15
16 # Debug target, add DEBUG Flags
17 debug: CFLAGS+=${DEBUGCFLAGS}
18 debug: $(EXE)
19
20 $(EXE): $(OBJ)
21         $(CC) $(LDFLAGS) -o $@ $^
22
23 %.o:%.c
24         $(CC) $(CFLAGS) -c $<
25
26 .PHONY: clean test
27 clean:
28     rm *.o
29     rm $(EXE)

Makefile
"Makefile" 29L, 410C
```

Adding Debug Information

Without -g

```
1. gareth@brutha: ~/Lec07/code (ssh)
main.c
6     int main(void) {
7         int data[SIZE_OF_DATA];
8
9         generateRandomData(data, SIZE_OF_DATA);
10
11        int i;
12        for(i=0; i<SIZE_OF_DATA; i++){
13            printf("%d: %d\n", i, data[i]);
14        }
15        return 0;
16    }
17
18
19
20
21
22
23
24
25
26
27
28
29
30

exec No process In:                                     Line: ??   PC: ???
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/gareth/Lec07/code/myprog...done.
(gdb)
```

With -g

- Getting started
- Running GDB
- Interacting with our Code

Running GDB

- You can start the gdb and load a program for debugging by:
 - **gdb <executable>**
- gdb comes with a simpler terminal UI, which can be used via:
 - **gdb --tui <executable>**
 - **gdbtui <executable>**

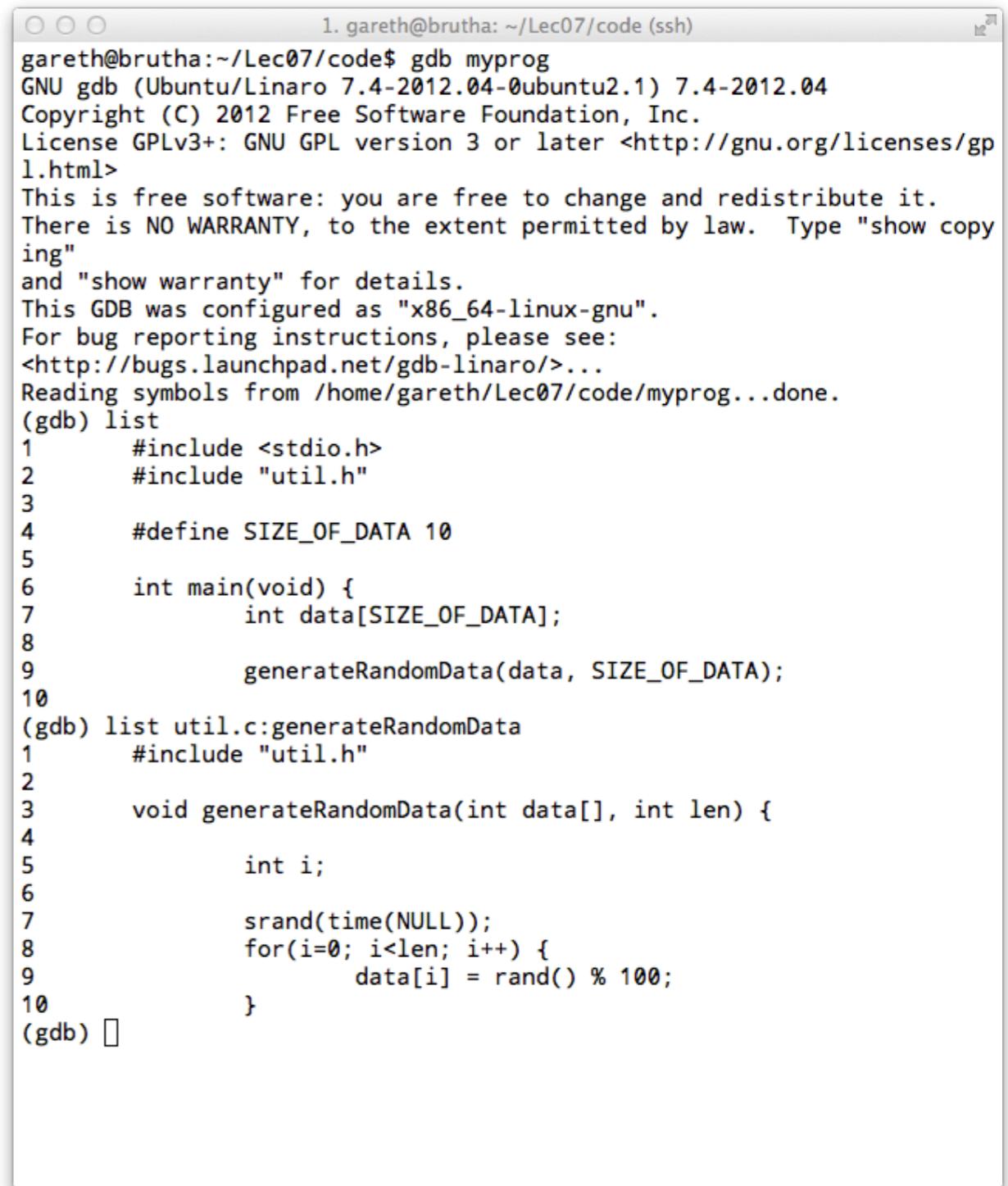
```
3. gareth@brutha: ~/Lec07/code (ssh)
gareth@brutha:~/Lec07/code$ gdb myprog
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/gareth/Lec07/code/myprog...done.
(gdb) 
```

```
1. gareth@brutha: ~/Lec07/code (ssh)
main.c
1      #include <stdio.h>
2      #include "util.h"
3
4      #define SIZE_OF_DATA 10
5
6      int main(void) {
7          int data[SIZE_OF_DATA];
8
9          generateRandomData(data, SIZE_OF_DATA);
10
11         int i;
12         for(i=0; i<SIZE_OF_DATA; i++){
13             printf("%d: %d\n", i, data[i]);
14         }
15     }
16
17
18
19
20
21
22
23
24
25

exec No process In:                                     Line: ??   PC: ?? 
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/gareth/Lec07/code/myprog...done.
(gdb) 
```

Seeing code in plain GDB

- If we use gdb without the terminal UI, we need to be able to see the code.
- To do this we can use the list command:
 - **list** - by default prints ten lines.
 - **list <line>** - prints the code at the line number specified.
 - **list <function>** - prints the code of the given function
 - **list <filename:function>** - prints the **function** in **filename**.
 - **list 1,4** - prints lines, 1 to 4
- It is often difficult to see what is going on by just using **gdb** so in the following examples we'll be using **gdbtui**.



```
1. gareth@brutha: ~/Lec07/code (ssh)
gareth@brutha:~/Lec07/code$ gdb myprog
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/gareth/Lec07/code/myprog...done.
(gdb) list
1      #include <stdio.h>
2      #include "util.h"
3
4      #define SIZE_OF_DATA 10
5
6      int main(void) {
7          int data[SIZE_OF_DATA];
8
9          generateRandomData(data, SIZE_OF_DATA);
10
(gdb) list util.c:generateRandomData
1      #include "util.h"
2
3      void generateRandomData(int data[], int len) {
4
5          int i;
6
7          srand(time(NULL));
8          for(i=0; i<len; i++) {
9              data[i] = rand() % 100;
10
(gdb) 
```

- Getting started
- Running GDB
- Interacting with our Code

Breakpoints

- Breakpoints are how we tell gdb to pause execution.
- We can set breakpoints at any statement in our code, or at function definitions.
- We set a breakpoint using the **break** or **b** keywords.
 - **break <line number>**
 - **break <function>**
 - **break <filename:function>**
- Once we've set our breakpoints we can start our code running with the **run** command (if we hadn't set any breakpoints the code would just run to completion).
- You can see a list of all the breakpoints in your current session by doing
 - **info breakpoints**
- You can use **delete** to remove set breakpoints:
 - **delete** - deletes all breakpoints
 - **delete <number>** - deletes breakpoint at number

The screenshot shows a terminal window with a GDB session. The top part of the window displays the source code for 'main.c' with line numbers. Line 9 is highlighted with a black background. The bottom part of the window shows the GDB command-line interface with the following output:

```
1. gareth@brutha: ~/Lec07/code (ssh)
main.c
1 #include <stdio.h>
2 #include "util.h"
3
4 #define SIZE_OF_DATA 10
5
6 int main(void) {
7     int data[SIZE_OF_DATA];
8
B+>9     generateRandomData(data, SIZE_OF_DATA);
10
11     int i;
12     for(i=0; i<SIZE_OF_DATA; i++){
13         printf("%d: %d\n", i, data[i]);
14     }
15     return 0;
16 }
17
18
19
20
21
22
23
24
25

child process 10660 In: main                                     Line: 9   PC: 0x4005cc
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/gareth/Lec07/code/myprog...done.
(gdb) break main
Breakpoint 1 at 0x4005cc: file main.c, line 9.
(gdb) run
Starting program: /home/gareth/Lec07/code/myprog
warning: no loadable sections found in added symbol-file system-supplied DSO at 0x7fffff7ffa000
dl-debug.c:77: No such file or directory.
dl-debug.c:77: No such file or directory.

Breakpoint 1, main () at main.c:9
(gdb) 
```

Running the code

- As mentioned in the previous slide we run our program in the debugger by using the **run** command.
- Once we've stopped at a breakpoint we can start execution again by using **continue** or **c**.
- This will run the code until the next breakpoint.
- We can also execute that code line by line.
- To execute line by line, and enter into functions we use **step** or **s** (this will also enter into system functions which can lead to issues as they normally have no debug info).
- To execute line by line, but not enter into a function we use **next** or **n**.
- We can tell each of these commands the number of lines to execute:
 - **step 3**
 - **next 3**
- We can stop the execution of our program by using the **kill** command.



The screenshot shows a terminal window with a GDB session. At the top, it says "1. gareth@brutha: ~/Lec07/code (ssh)". Below that is a code editor window titled "util.c" showing the following C code:

```
1 #include "util.h"
2
3 void generateRandomData(int data[], int len) {
4     int i;
5
6     > 7         srand(time(NULL));
7         for(i=0; i<len; i++) {
8             data[i] = rand() % 100;
9         }
10    }
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
```

Below the code editor, the terminal output shows:

```
child process 42858 In: generateRandomData      Line: 7      PC: 0x400628
(gdb) break main
Breakpoint 1 at 0x4005cc: file main.c, line 9.
(gdb) run
Starting program: /home/gareth/Lec07/code/myprog
warning: no loadable sections found in added symbol-file system-supplied DSO at 0x7fffff7ffa000
dl-debug.c:77: No such file or directory.
dl-debug.c:77: No such file or directory.

Breakpoint 1, main () at main.c:9
(gdb) s
generateRandomData (data=0x7fffffff450, len=10) at util.c:7
(gdb) █
```

Examining Variables

- Now that we can run our program, we want to be able to examine the contents of the variables.
- We can display the contents of each variable using the **print** or **p** command.
 - print <variable>** - print variables contents
 - print &<variable>** - print the address of variable
 - print *<variable>** - print to data pointed to by variable
 - ptype <variable>** - print type of variable
 - print *<variable>@<number>** - print at number of elements pointed to by variable
- We can also specify the output format by appending a type:
 - p/x <variable>** - hexadecimal
 - p/u <variable>** - unsigned integer
 - p/d <variable>** - signed integer
 - p/o <variable>** - octal
 - p/f <variable>** - floating point

The screenshot shows a terminal window with a GDB session. The code in main.c defines a function that generates random data and prints it. The GDB session shows breakpoints being set and variables being examined using various print commands.

```
1. gareth@brutha: ~/Lec07/code (ssh)
main.c
1 #include <stdio.h>
2 #include "util.h"
3
4 #define SIZE_OF_DATA 10
5
6 int main(void) {
7     int data[SIZE_OF_DATA];
8
9     generateRandomData(data, SIZE_OF_DATA);
10
11    int i;
12    for(i=0; i<SIZE_OF_DATA; i++){
13        printf("%d: %d\n", i, data[i]);
14    }
15    return 0;
16
17
18
19
20
21
22
23
24
25
```

child process 58387 In: main Line: 12 PC: 0x4005dd

```
Breakpoint 1, main () at main.c:12
(gdb) p data
$1 = {88, 91, 33, 95, 87, 85, 98, 56, 41, 43}
(gdb) p &data
$2 = (int (*)[10]) 0x7fffffff450
(gdb) p *data
$3 = 88
(gdb) ptype data
type = int [10]
(gdb) p *data@3
$4 = {88, 91, 33}
(gdb)
```

Watch points

- Often we have problems with variables changing during the execution of a program.
- For instance if we have written beyond the end of an array we could be overwriting locations in memory.
- Or we may have multiple pointers to an area of memory which is being accessed.
- We'd like to be able to observe a variable and stop execution when it is changed.
- To do this we can set a watch point using the **watch** command.
 - **watch <variable>**
 - i.e **watch i**
 - **watch <condition>**
 - i.e **watch i > 7**

The screenshot shows a terminal window with the following details:

- File:** main.c
- Content:**

```
1 #include <stdio.h>
2 #include "util.h"
3
4 #define SIZE_OF_DATA 10
5
6 int main(void) {
7     int data[SIZE_OF_DATA];
8
9     generateRandomData(data, SIZE_OF_DATA);
10
11    int i;
12    for(i=0; i<SIZE_OF_DATA; i++){
13        printf("%d: %d\n", i, data[i]);
14    }
15    return 0;
16
17
18
19
20
21
22
23
24
25}
```
- Execution:** The program is running in a child process (pid 2733). The current line is 12, and the PC is at 0x4005e4.
- GDB Session:**

```
0: 36
child process 2733 In: main
Line: 12  PC: 0x4005e4
dl-debug.c:77: No such file or directory.

Breakpoint 1, main () at main.c:12
(gdb) watch i
Hardware watchpoint 2: i
(gdb) continue
Continuing.
Hardware watchpoint 2: i

Old value = 0
New value = 1
0x00000000040060a in main () at main.c:12
(gdb) 
```

- Getting started
- Running GDB
- Interacting with our Code



**KEEP
CALM
AND
REVISE**

Revision

Dr. Gareth Roy (x6439)
gareth.roy@glasgow.ac.uk

- The Filesystem
- Processes
- Bash Scripting
- Compilation
- Makefiles
- Git
- GDB

- The Filesystem
- Processes
- Bash Scripting
- Compilation
- Makefiles
- Git
- GDB

The Filesystem

- You should be able to:
 - Understand the difference between relative and absolute paths
 - Know what a hidden file is, and give examples of some.
- Navigate and explore the filesystem using commands such as:

`pwd, tree, ls, ls -l, ls -a, cd, pushd, popd, find, locate, whereas`

- Modify file permissions with chmod (see next slide)
- Create files and directories:

`touch, mkdir, mkdir -p`

- Move and copy files:

`mv, cp, cp -r`

- Delete files:

`rm, rm -r, rm -f`

- Explore the contents of files with:

`file, cat, less, more, head, tail, grep, wc, sort`

Permissions

Filetype → -rwxrw-rw-

User Group Other

- First part is filetype
 - **d** - directory
 - **l** - link
 - **-** - normal file
- Next is permissions, split into three parts
 - **User** permissions
 - **Group** permissions
 - **Other** permissions (everyone on the system)
- Permissions are
 - **r** - read access
 - **w** - write access
 - **x** - execute

chmod [who][op][what] filename

- Can change permissions of a file using the **chmod** command
- who can be:
 - **u** - user permissions
 - **g** - group permissions
 - **o** - other permissions
 - **a** - all permissions
- op can be:
 - **+** - grant permissions
 - **-** - remove permissions
- what is one of the three permission types (**r,w,x**)

- The Filesystem
- Processes
- Bash Scripting
- Compilation
- Makefiles
- Git
- GDB

Processes

- You should be able to:
- Explain what an environment variable is, how to set one and how to get its contents:

```
MYVAR='somevalue'; echo ${MYVAR}
```

- Understand what important system variables do:

```
$PATH, $HOME, $PWD, $USER, $LD_LIBRARY_PATH
```

- How to set persistent variables in .bash_profile and .bashrc
- How to monitor processes:

```
pstree, ps, ps -elf, ps -auxwf, top
```

- How to work with processes and basic job control:

```
&, fg, bg, jobs, ctrl-c, ctrl-z, kill, kill -9, pidof
```

- How to work with IO Streams:

```
stdin, stdout, stderr
```

- Redirection and Pipes:

```
<, >, >>, 2>&1, |
```

- The Filesystem
- Processes
- Bash Scripting
- Compilation
- Makefiles
- Git
- GDB

Bash Scripting

- You should be able to:
- Make a simple script and know how to run it using source and by making it executable (chmod):

```
#!/bin/bash
echo "Hello, World!"
```

- Know the difference between single quotes and double quotes and why I needed quotes above (c.f. Bash word splitting).
- Understand and be able to write Bash ranges and lists

```
{1..3}, {a..z}, `seq 1 10`
```

- Know how to use variables in scripts:

```
CALCFUNC=${HOME}/bin/thermCoeff
echo ${CALCFUNC}
```

- Know basic control flow constructions:

```
if [ "${CALCFUNC}" = "" ]; then
    echo "${CALCFUNC} is not set"
fi

while [ -e lock file ]; do
    echo "Cannot start - rm lock file"
    sleep 10
done

for ITEM in $LIST; do
    echo $ITEM
done
```

Bash Conditionals

```
[ -f tmp.txt ]
```

space ↑ ↑ space
flag file/string

String Comparision	Result
string 1 == string 2	True if the strings are equal
string 1 != string 2	True if the strings are different
-n string	True if the string is not null
-z string	True if the string is null

File Conditionals	Result
-d file	True if file is a directory
-e file	True if file exists
-f file	True if the file is regular
-r file	True if file is readable
-s file	True if file has nonzero size
-w file	True if file is writeable

Arithmetic Comparision	Result
exp 1 -eq exp 2	True if both are equal
exp 1 -ne exp 2	True if both are different
exp 1 -gt exp 2	True if exp1 is greater than exp2
exp 1 -ge exp 2	True if exp1 is greater than or equal to exp2
exp 1 -lt exp 2	True if exp1 is less than exp2
exp 1 -le exp 2	True if exp1 is less than or equal to exp2
! exp	Invertes exp, true if exp is false. False if exp is true

Bash Scripting

- Know how to run external commands and store output in variables.

```
VAR=`ls`  
VAR=$(wc bob.txt)
```

- Know about exit statuses:

```
exit 0  
exit 1
```

- Understand the meanings and uses for special variables:

```
$$, $0, $1..., $# , $? , $@
```

- How to get user input, and read a file line by line:

```
read, read -p,  
cat some file | while read LINE; do echo $LINE; done
```

- How to declare and call a Bash function:

```
function myfunc {  
    echo "Hello, $1!"  
}  
myfunc "Bob"
```

- How to pass arguments to a Bash function and access them using:

```
$@, $# , $1..., shift
```

- How to debug a Bash file using:

```
set -x, set -n, set-v
```

- The Filesystem
- Processes
- Bash Scripting
- Compilation
- Makefiles
- Git
- GDB

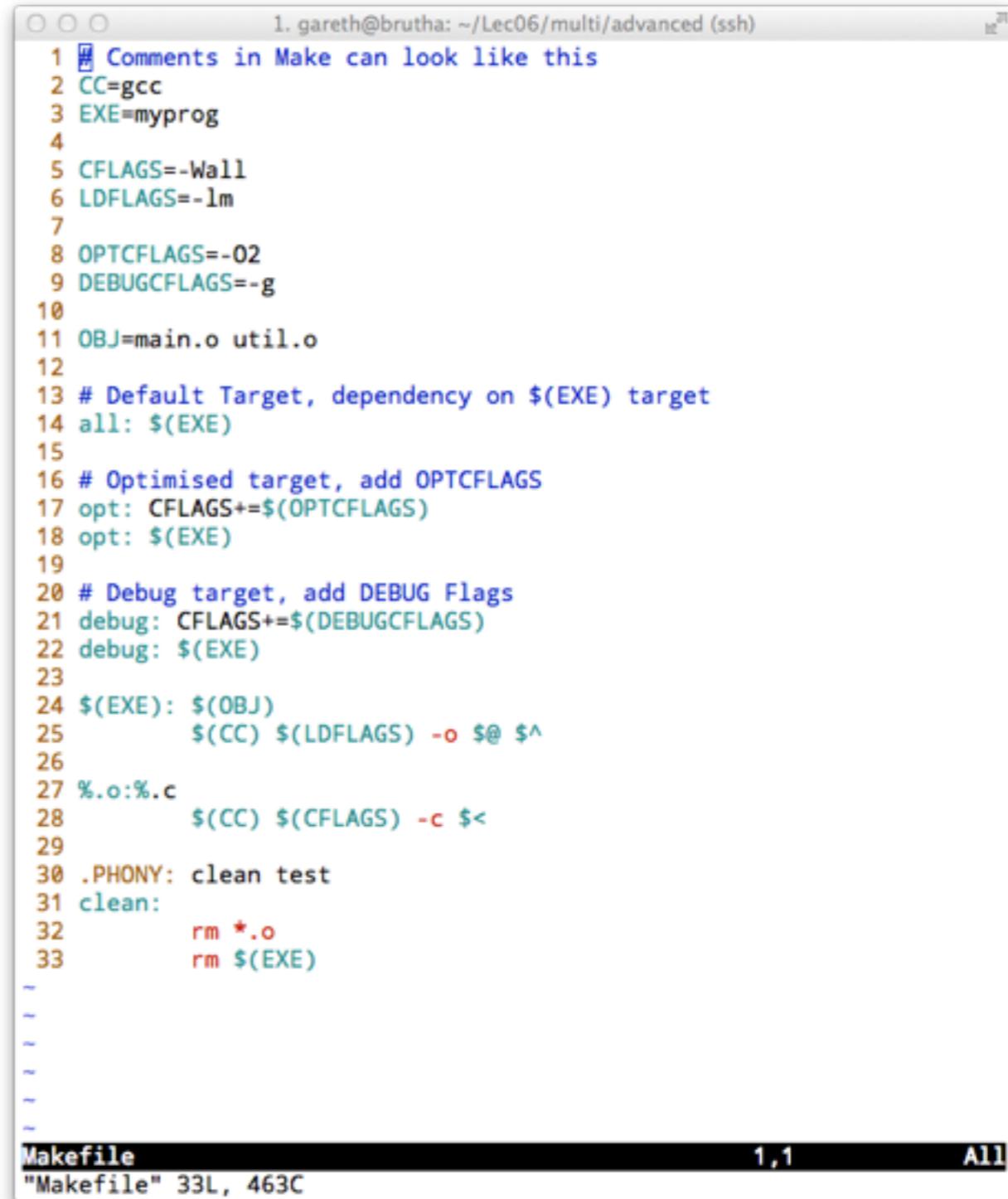
Compilation

- You should be able to:
- Compile C source code to create an executable.
- Understand what the pre-processor, compiler and linker do and how the compilation process works.
- How to use gcc to carry out a one line compilation:
 - `gcc -lm -o myprog *.c`
- How to use gcc to carry out an incremental build:
 - `gcc -c main.c`
 - `gcc -c util.c`
 - `gcc -lm -o myprog main.o util.o`
- Understand what libraries are and how to include them via the gcc compilation line.
- Understand the difference between a source file and a header file.
- The tools that can be used to explore the steps in the build process:
 - `nm`, `ldd`, `hexdump`

- The Filesystem
- Processes
- Bash Scripting
- Compilation
- Makefiles
- Git
- GDB

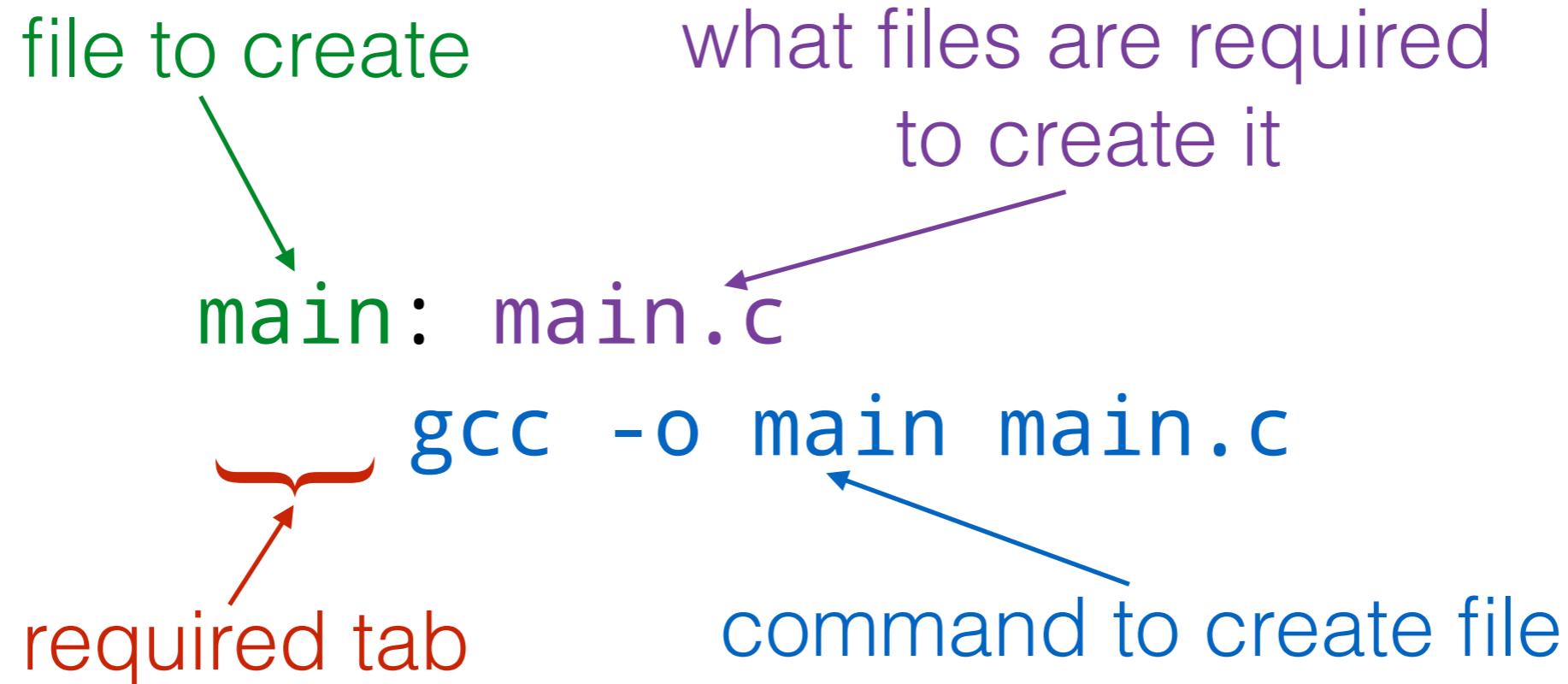
Makefiles

- You should know or be able to:
- Know what makefiles are for and how to write a basic makefile.
- Understand how they help with the incremental build process.
- Know how to create a rule for a specific target.
- Know how to run make from the command line, and how to run a specific target.
- Understand the automatic make variables: %, \$@, \$^, \$<, \$?.
- Know how to make a phoney target and why you may want to do that (c.f. clean, test).



```
1 # Comments in Make can look like this
2 CC=gcc
3 EXE=myprog
4
5 CFLAGS=-Wall
6 LDFLAGS=-lm
7
8 OPTCFLAGS=-O2
9 DEBUGCFLAGS=-g
10
11 OBJ=main.o util.o
12
13 # Default Target, dependency on $(EXE) target
14 all: $(EXE)
15
16 # Optimised target, add OPTCFLAGS
17 opt: CFLAGS+=${OPTCFLAGS}
18 opt: $(EXE)
19
20 # Debug target, add DEBUG Flags
21 debug: CFLAGS+=${DEBUGCFLAGS}
22 debug: $(EXE)
23
24 $(EXE): $(OBJ)
25         $(CC) $(LDFLAGS) -o $@ $^
26
27 %.o:%.c
28         $(CC) $(CFLAGS) -c $<
29
30 .PHONY: clean test
31 clean:
32         rm *.o
33         rm $(EXE)
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
748
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
778
779
779
780
781
782
783
784
785
786
787
788
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
838
839
839
840
841
842
843
844
845
846
847
847
848
849
849
850
851
852
853
854
855
856
857
858
858
859
859
860
861
862
863
864
865
866
867
867
868
868
869
869
870
871
872
873
874
875
876
876
877
877
878
878
879
879
880
881
882
883
884
885
886
886
887
887
888
888
889
889
890
891
892
893
894
895
895
896
896
897
897
898
898
899
899
900
901
902
903
904
905
905
906
906
907
907
908
908
909
909
910
911
912
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1
```

Make Rules



- Makefile rules require a **target**, its **dependencies** and the **command** needed to produce the target from its dependencies. In this instance `main` is produced from `main.c` by running `gcc`.
- Makefiles can be used for a number of things, not just compiling. For instance the following rule downloads the xkcd comic you saw before:

`compiling.png:`

`wget http://imgs.xkcd.com/comics/compiling.png`

- The Filesystem
- Processes
- Bash Scripting
- Compilation
- Makefiles
- Git
- GDB

Git

- You should be able to:
- Understand the purpose of revision control.
- Understand the basic concepts used in revision control:
 - repository, checkout, commit, branch, merge
- Understand the difference between distributed and centralised revision control systems.
- Initialise a basic git repository:
 - `git init project`
 - `git clone https://www.bitbucket.org/p2t/myrepo`
- Add and remove files
 - `git add myfile.c`
 - `git rm myfile.c`
- The importance if the commit command and what it is used for.
 - `git commit -m "My first commit"`
- How to check the status and get the commit history:
 - `git status`
 - `git log --oneline`
- How to work with branches:
 - `git branch my feature`
 - `git branch --list`
 - `git checkout myfeature`
 - `git merge myfeature`

- The Filesystem
- Processes
- Bash Scripting
- Compilation
- Makefiles
- Git
- GDB

GDB

- You should be able to:
- Compile code such that it contains the required information to run gdb (gcc -g)
- Run gdb from the command line:
 - `gdb`
 - `gdbtui`
- Understand the basic gdb commands
 - `list`, `run`, `break`, `next`, `step`, `continue`, `print`, `watch`
- Have a basic understanding of how you could use gdb to debug a program.

- The Filesystem
- Processes
- Bash Scripting
- Compilation
- Makefiles
- Git
- GDB & debugging