Why Linux?

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

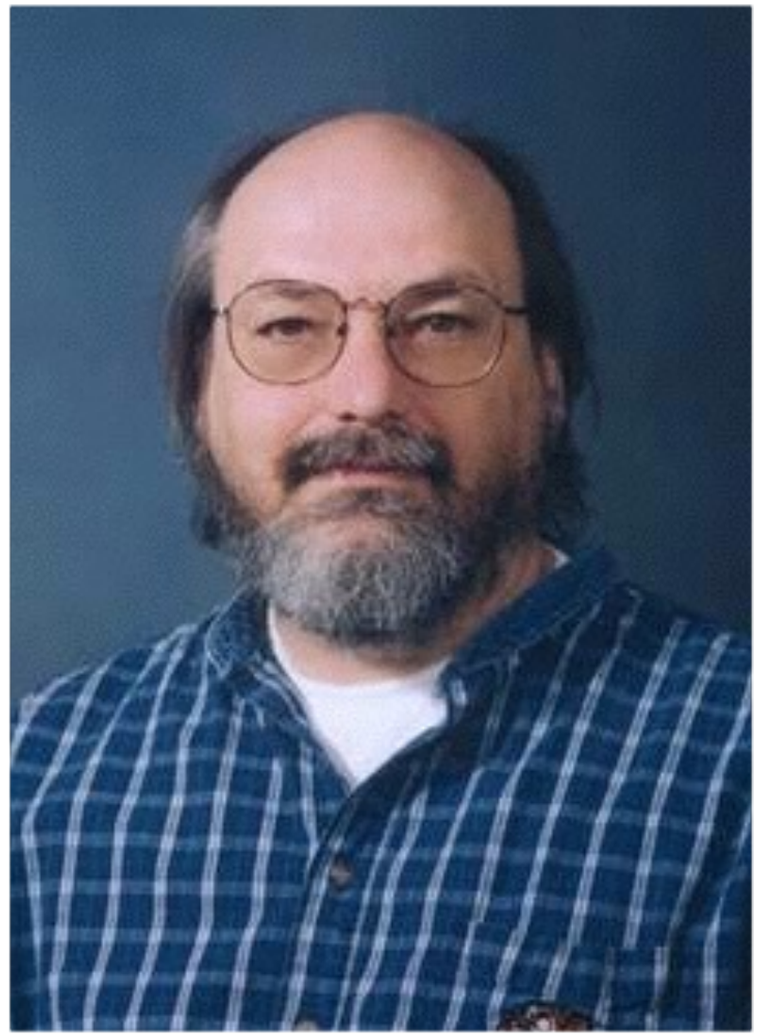
- A brief history lesson
- Why Linux?
- The parts of an Operating System
- The Command Line

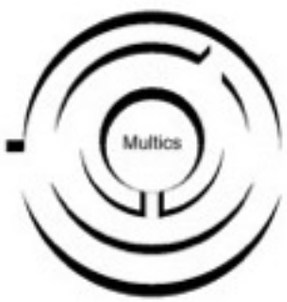
- A brief history lesson
- Why Linux?
- The parts of an Operating System
- The Command Line


IBM
IBM
DATACENTRE

Kenneth Lane Thomson (1943 -)

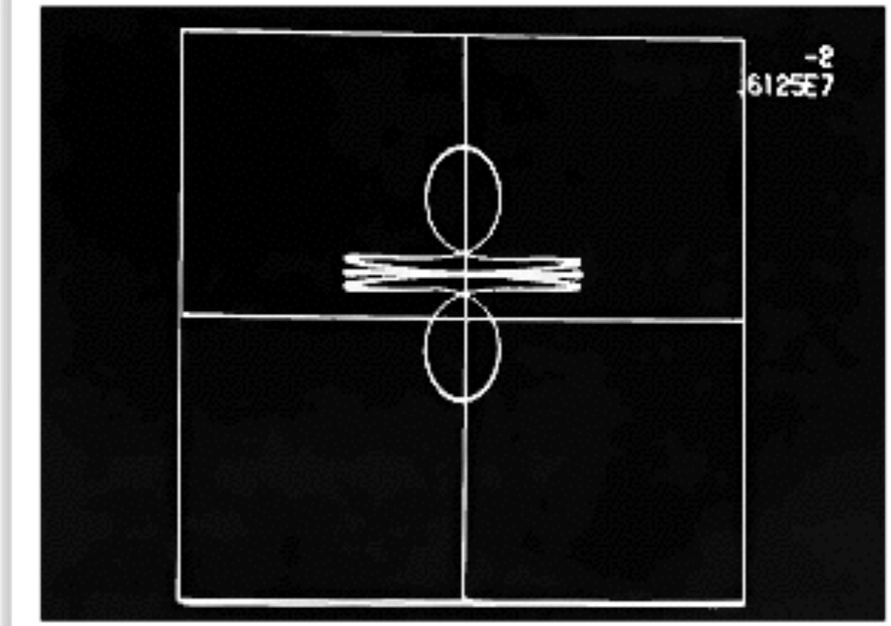
- Received his Bachelor's and Master's degrees in E&EE from the University of California at Berkley
- In 1966 was hired by Bell Labs (owned by AT&T) to work on the MULTICS project.
- MULTICS (**Mult**iplexed **I**nformation and **C**omputing **S**ervice) was a mainframe time sharing operating system.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

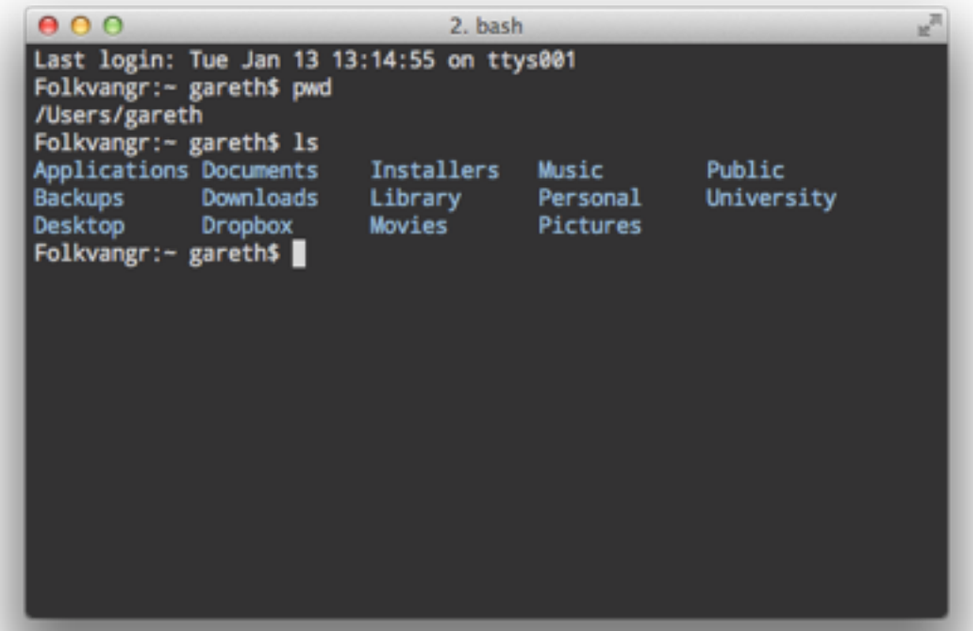
Ken Thomson and Dennis Ritchie working on the PDP-11 used to develop UNIX

What is UNIX

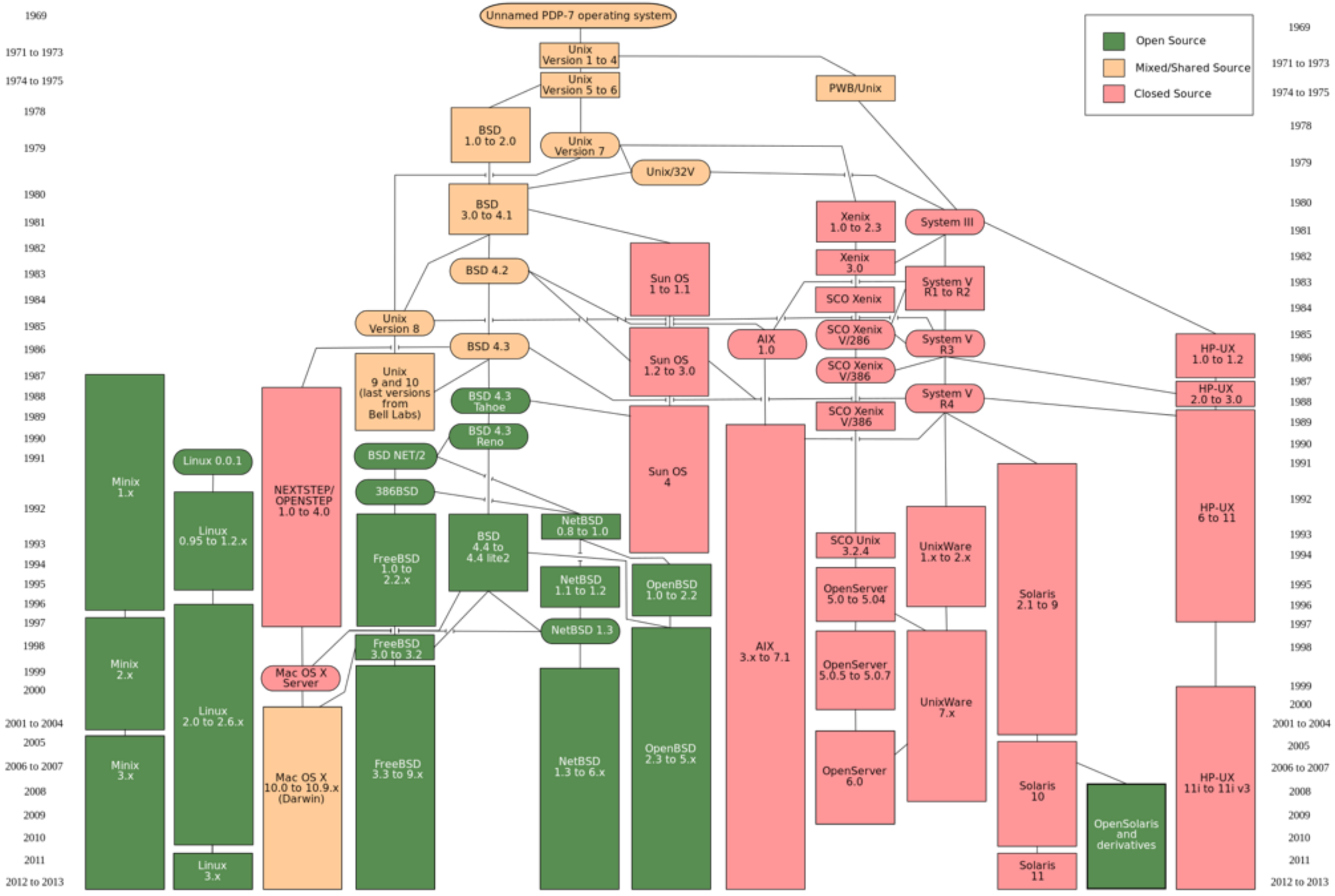
- UNIX is a Multi-user, Multi-tasking operating system (at present a family of Operating Systems).
- Any UNIX system has some key features:
  - the use of plain text for storing data
  - a hierarchical file system
  - devices and processes represented as files
  - the use of a large number of software 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

UNIX®

Celebrating 40 years uptime



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

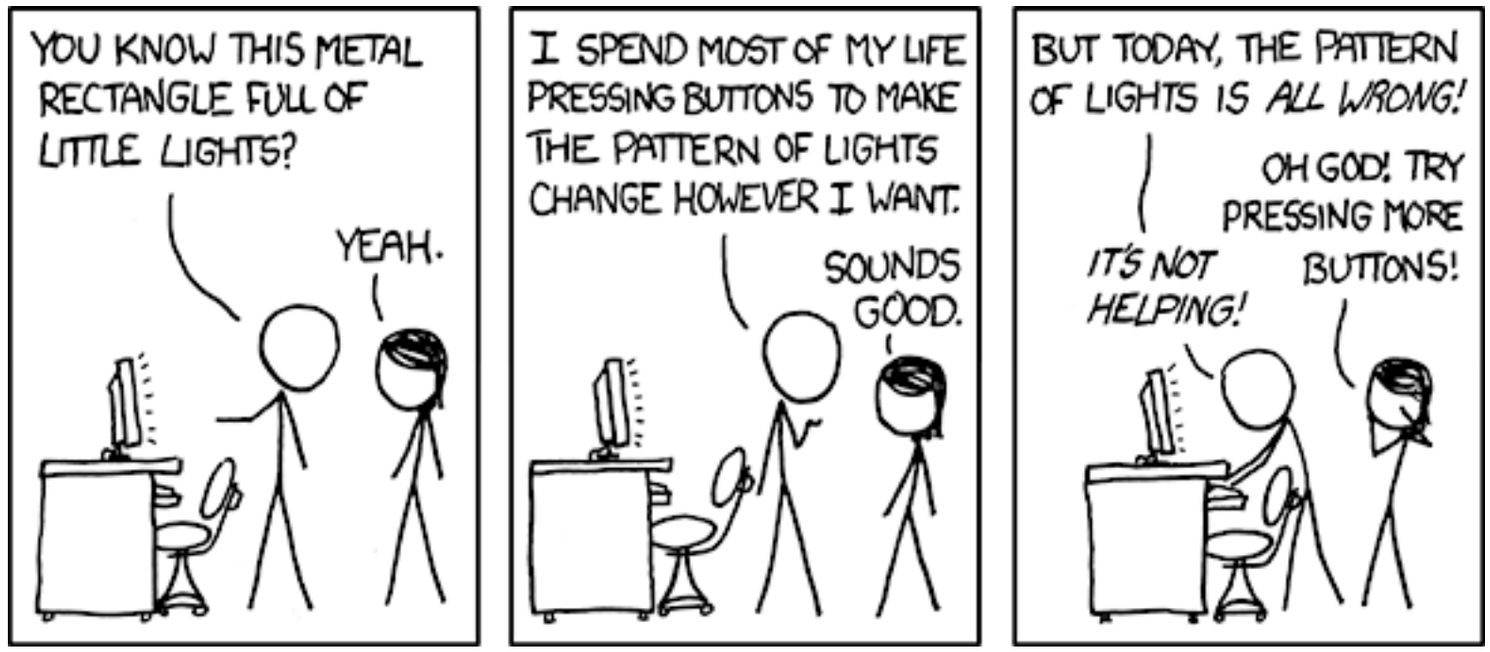

Open Source
Mixed/Shared Source
Closed Source
Unnamed PDP-7 operating system
Unix Version 1 to 4
Unix Version 5 to 6
Unix Version 7
PWB/Unix
BSD 1.0 to 2.0
Unix/32V
BSD 3.0 to 4.1
Xenix 1.0 to 2.3
System III
Xenix 3.0
BSD 4.2
Sun OS 1 to 1.1
System V R1 to R2
SCO Xenix
Unix Version 8
SCO Xenix V/286
AIX 1.0
System V R3
BSD 4.3
HP-UX 1.0 to 1.2
Unix 9 and 10 (last versions from Bell Labs)
Sun OS 1.2 to 3.0
SCO Xenix V/386
BSD 4.3 Tahoe
HP-UX 2.0 to 3.0
System V R4
BSD 4.3 Reno
Minix 1.x
Linux 0.0.1
NEXTSTEP/ OPENSTEP 1.0 to 4.0
BSD NET/2
Sun OS 4
SCO Xenix V/386
386BSD
HP-UX 6 to 11
Linux 0.95 to 1.2.x
FreeBSD 1.0 to 2.2.x
BSD 4.4 to 4.4 lite2
NetBSD 0.8 to 1.0
SCO Unix 3.2.4
UnixWare 1.x to 2.x
NetBSD 1.1 to 1.2
OpenBSD 1.0 to 2.2
OpenServer 5.0 to 5.04
Solaris 2.1 to 9
NetBSD 1.3
Minix 2.x
FreeBSD 3.0 to 3.2
OpenServer 5.0.5 to 5.0.7
UnixWare 7.x
Linux 2.0 to 2.6.x
Mac OS X Server
AIX 3.x to 7.1
OpenBSD 2.3 to 5.x
OpenServer 6.0
Solaris 10
OpenSolaris and derivatives
HP-UX 11i to 11i v3
Minix 3.x
Mac OS X 10.0 to 10.9.x (Darwin)
FreeBSD 3.3 to 9.x
NetBSD 1.3 to 6.x
Linux 3.x
Solaris 11
1969
1971 to 1973
1974 to 1975
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001 to 2004
2005
2006 to 2007
2008
2009
2010
2011
2012 to 2013

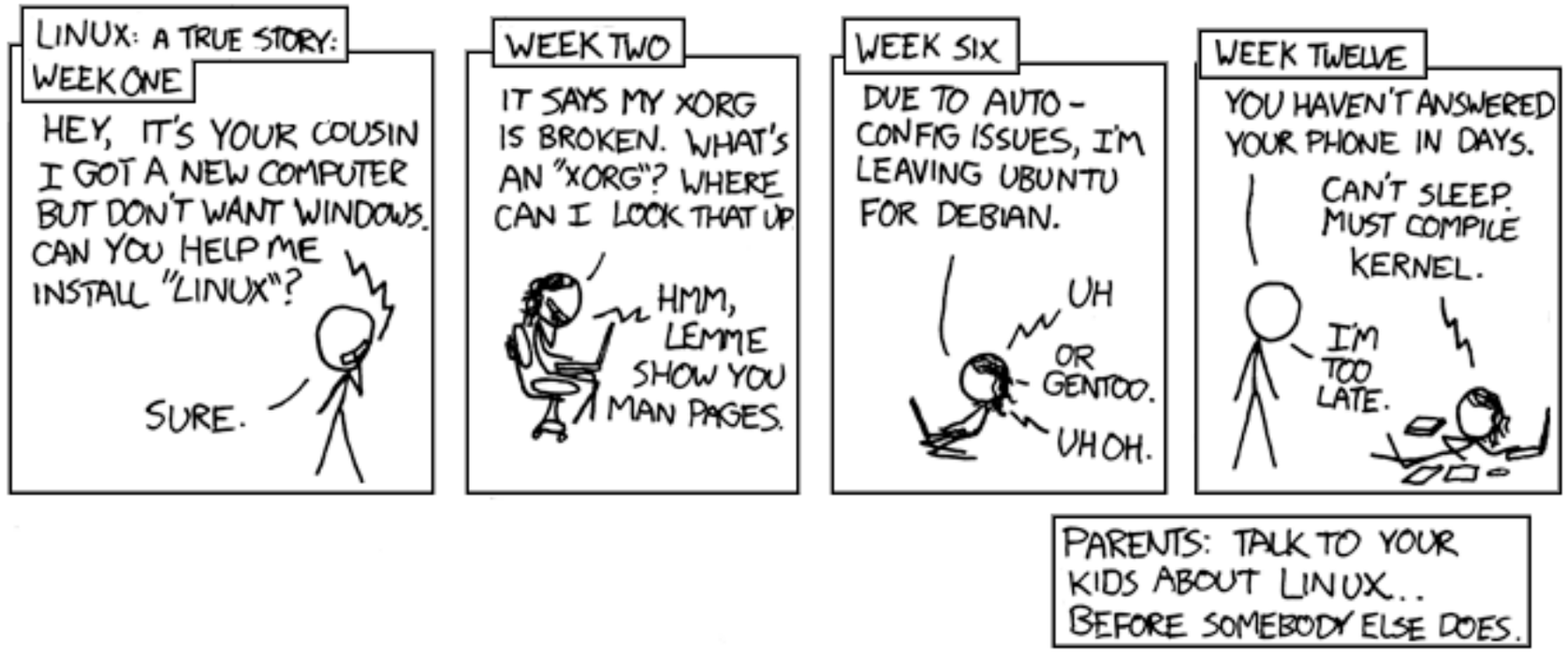
Linus Benedict Torvalds (1969 -)

- 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 place 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


YOU KNOW THIS METAL RECTANGLE FULL OF LITTLE LIGHTS?
YEAH.
I SPEND MOST OF MY LIFE PRESSING BUTTONS TO MAKE THE PATTERN OF LIGHTS CHANGE HOWEVER I WANT.
SOUNDS GOOD.
BUT TODAY, THE PATTERN OF LIGHTS IS ALL WRONG!
IT'S NOT HELPING!
OH GOD! TRY PRESSING MORE BUTTONS!


LINUX: A TRUE STORY:
WEEK ONE
HEY, IT'S YOUR COUSIN I GOT A NEW COMPUTER BUT DON'T WANT WINDOWS. CAN YOU HELP ME INSTALL "LINUX"?
SURE.
WEEK TWO
IT SAYS MY XORG IS BROKEN. WHAT'S AN "XORG"? WHERE CAN I LOOK THAT UP
HMM, LEMME SHOW YOU MAN PAGES.
WEEK SIX
DUE TO AUTO-CONFIG ISSUES, I'M LEAVING UBUNTU FOR DEBIAN.
UH
OR GENTOO.
UH OH.
WEEK TWELVE
YOU HAVEN'T ANSWERED YOUR PHONE IN DAYS.
CAN'T SLEEP. MUST COMPILE KERNEL.
I'M TOO LATE.
PARENTS: TALK TO YOUR KIDS ABOUT LINUX.. BEFORE SOMEBODY ELSE DOES.

xkcd.com

Why Linux?

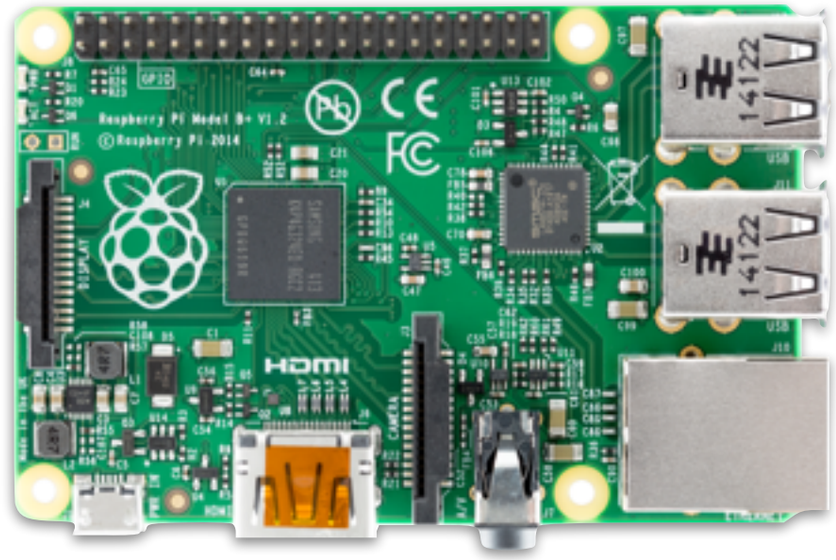
- Linux is a free, with no cost required to obtain a complete fully featured OS.
- Linux source code is freely available and can be modified to fit user needs.
- Linux is a multi-user, multi-tasking operating system based on the tested principles of UNIX.
- Linux is ubiquitous in scientific computing 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.

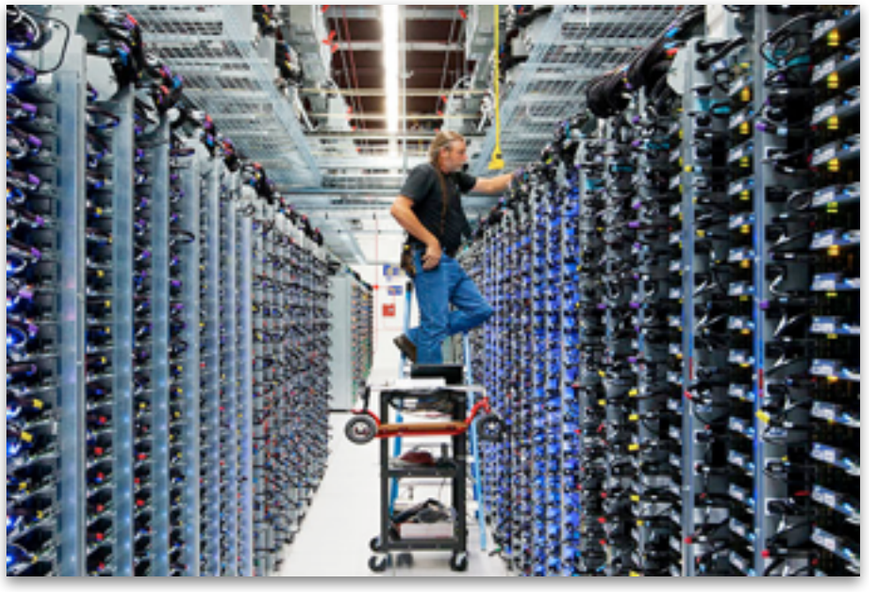
Tianhe-2 (MilkyWay-2)

- 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 exits but in general many of these are "Forks" of either Debian or Fedora.

distrowatch.com

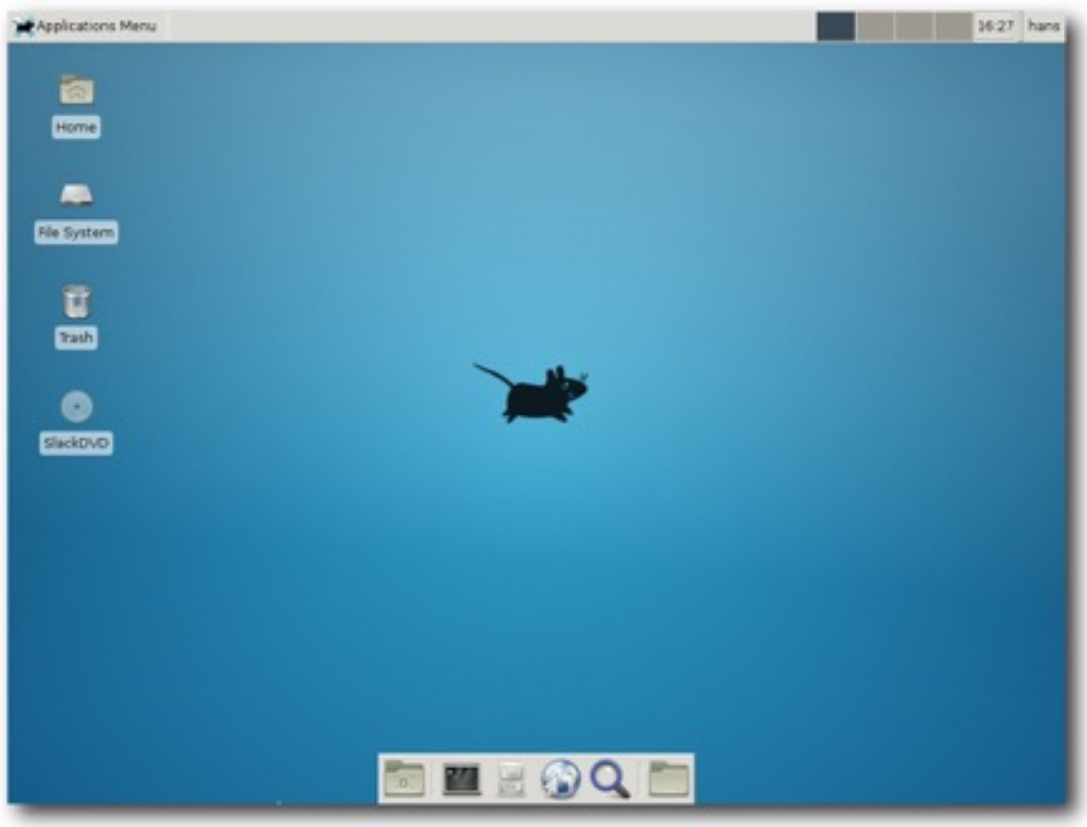
Installing Linux in a VM

- 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.

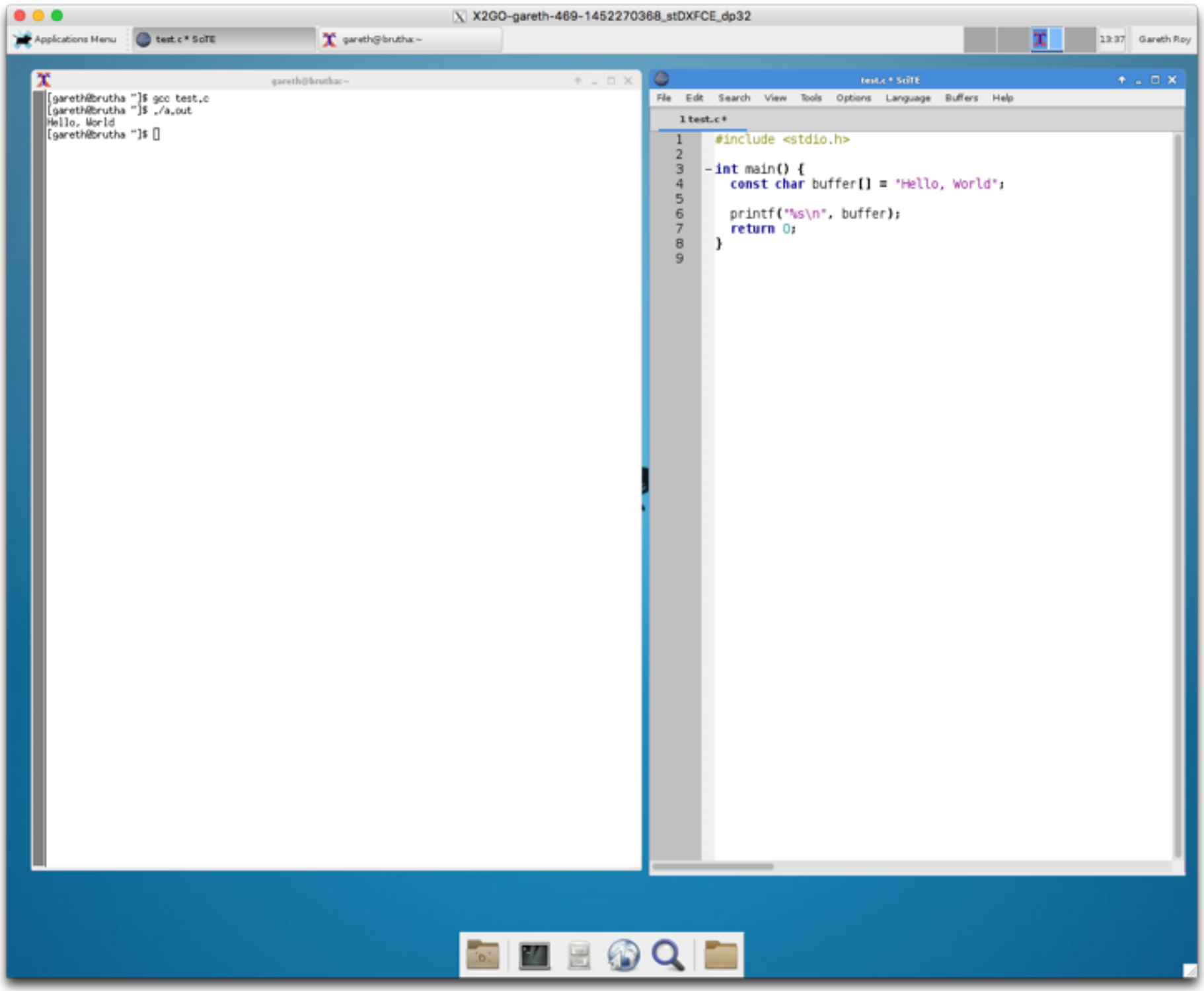
What you'll be using for this course

- 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

CentOS

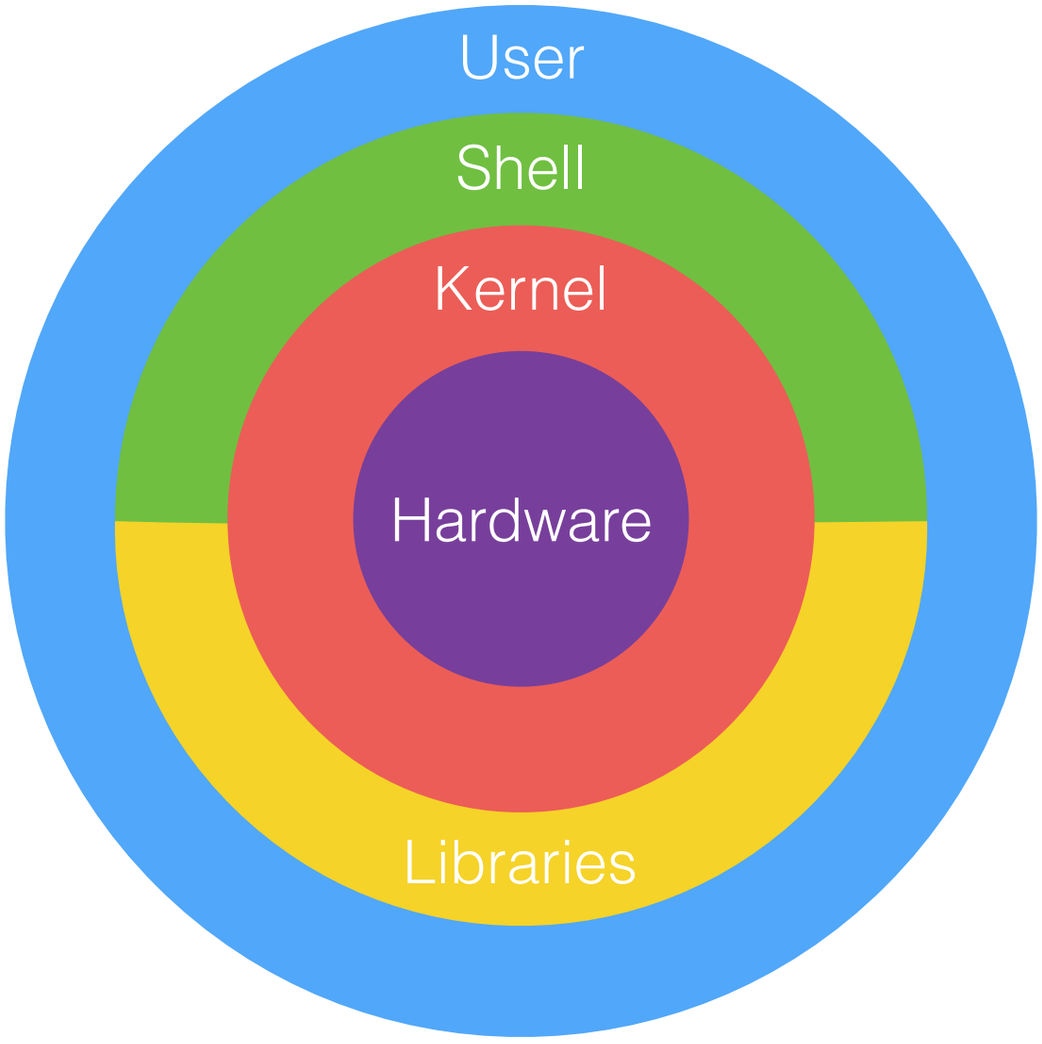


Accessing brutha.physics.gla.ac.uk



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

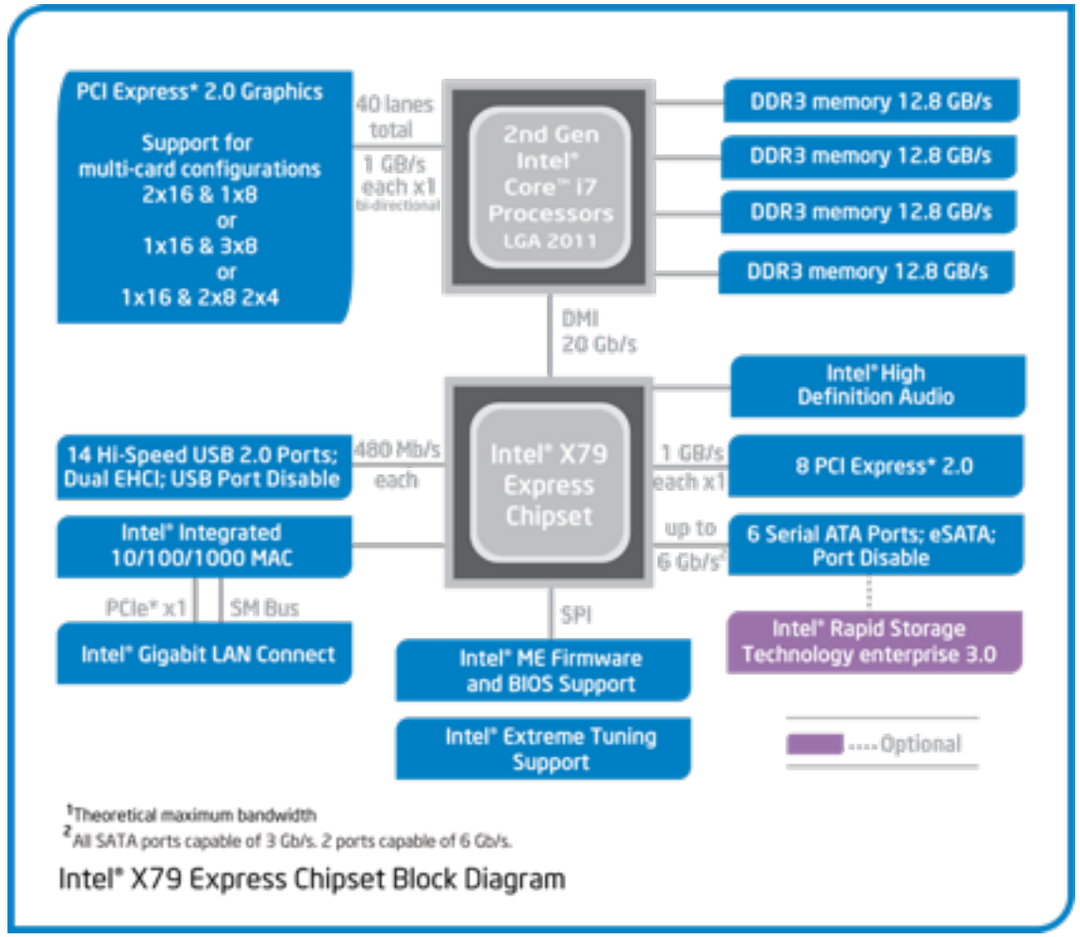
Linux Architecture

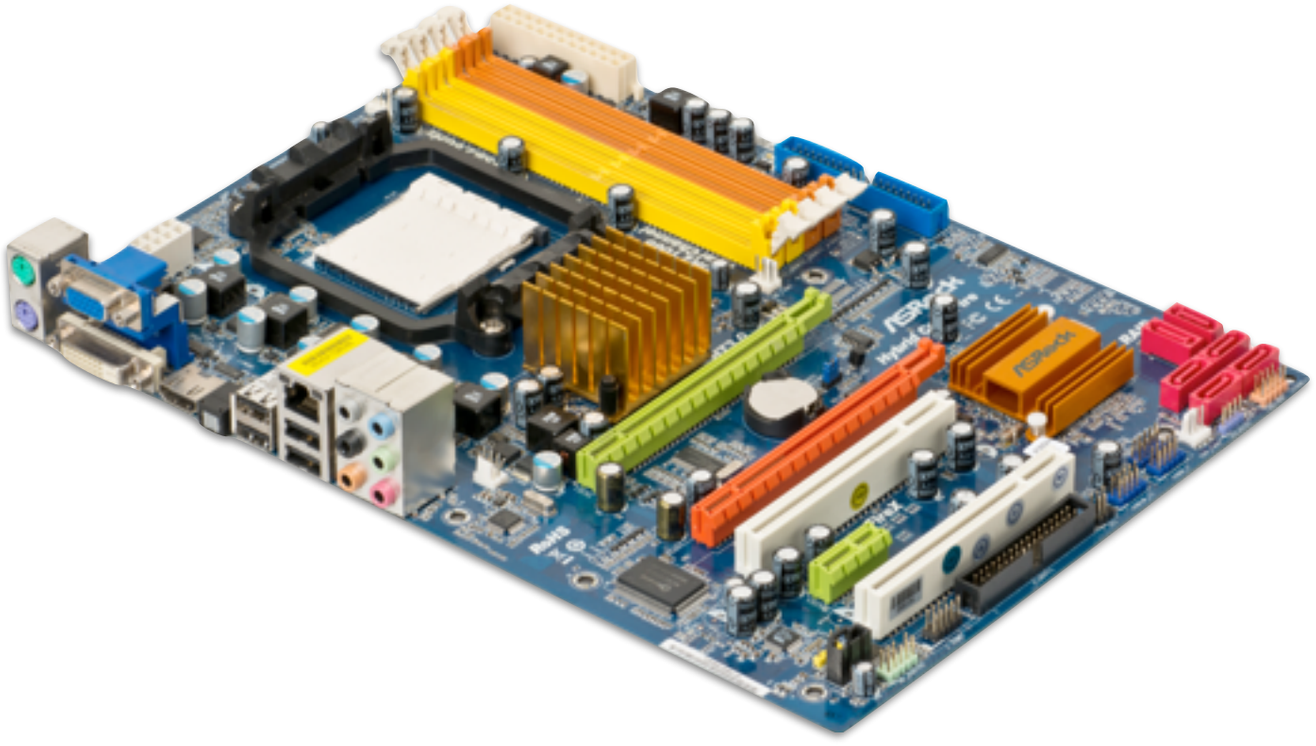


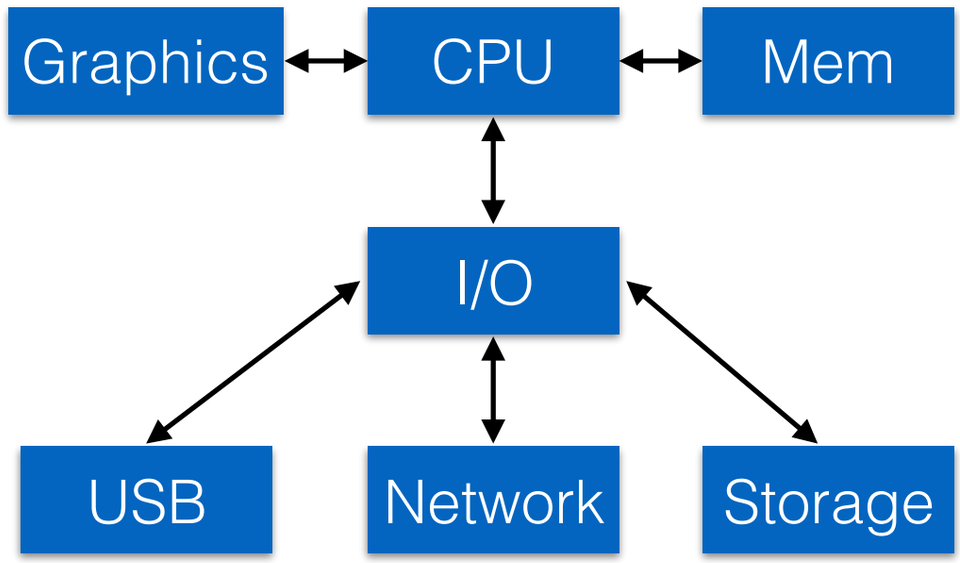
- The Kernel is the core part of Linux, it includes all the drivers need to interact with the underlying hardware.
- The Shell allows interactive access to system resource through a set of builtin commands or by running commands found on the filesystem
- Additionally access to system resources can be 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

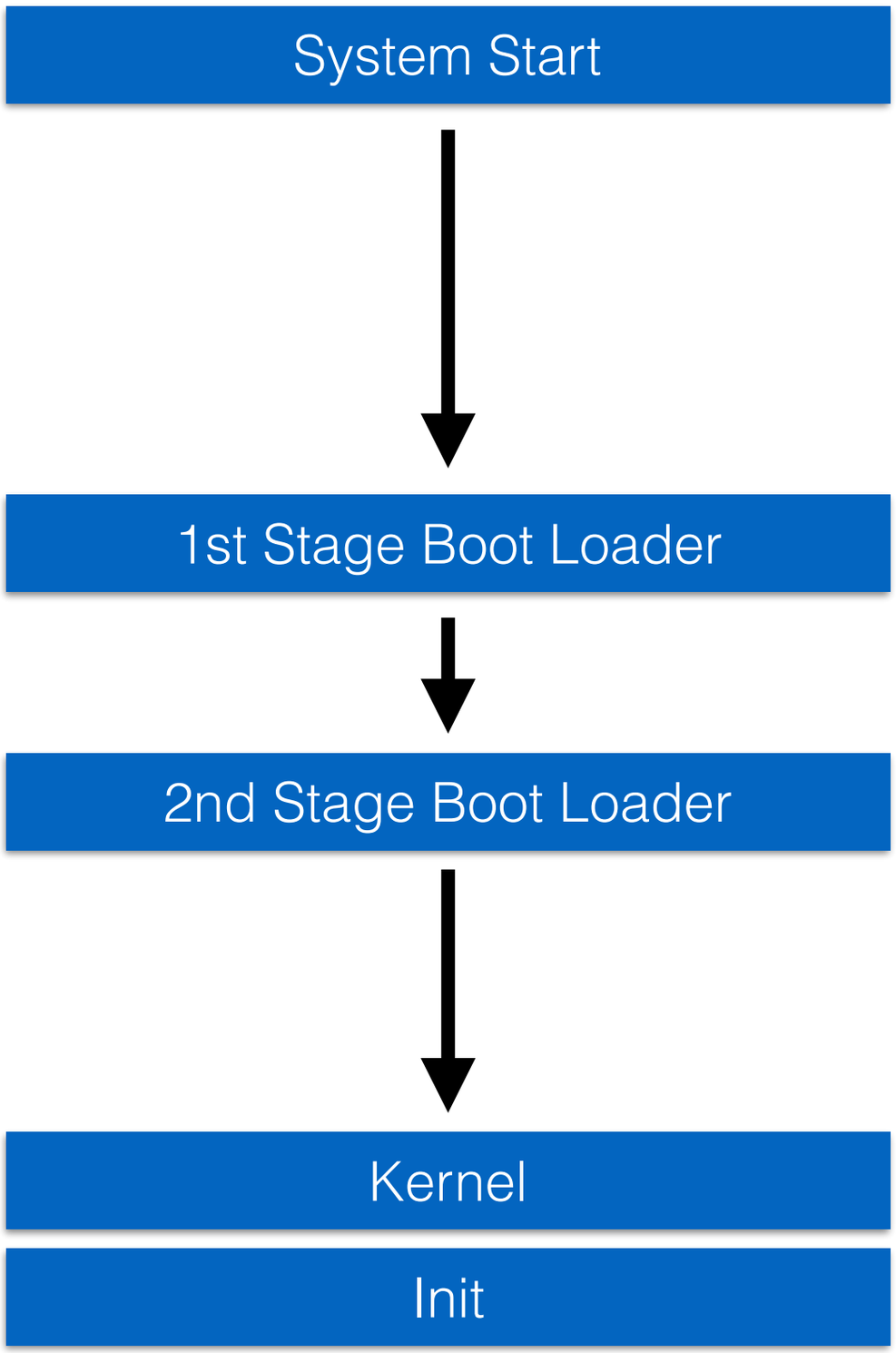
Component	Size	Purpose
CPU	GHz	Numeric / Logic / Shift / Vector Operations
Memory	GB	Running Software and Data storage.
Network	Gbit	Data transmission, TCP/IP packet decoding.
Storage	TB	Data storage preserved without power requirements
Graphics	MOps	VGA Display, 2D/3D Acceleration, Floating point ops.







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

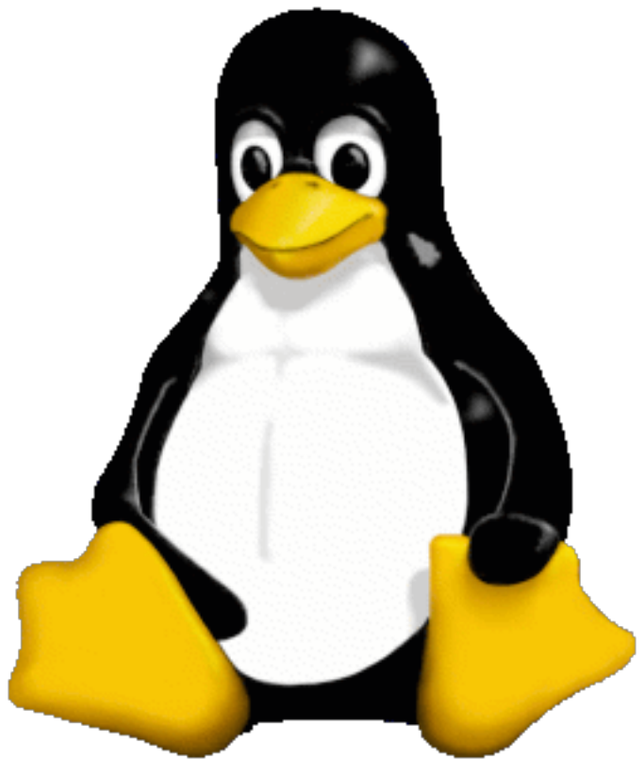
Loads the Kernel

- 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.

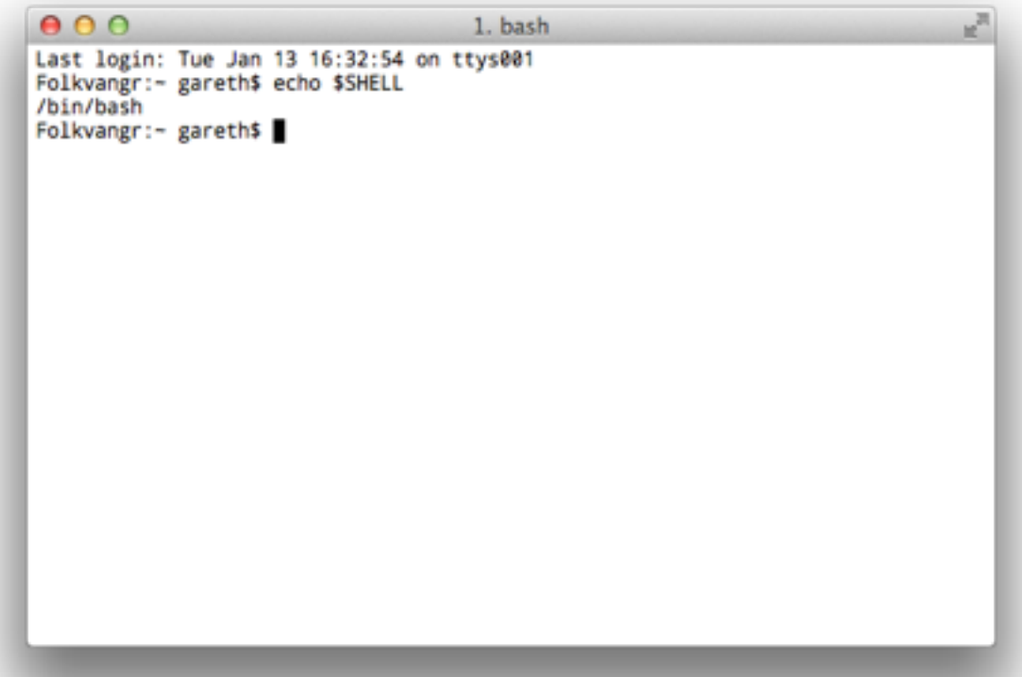
Kernel

- 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…

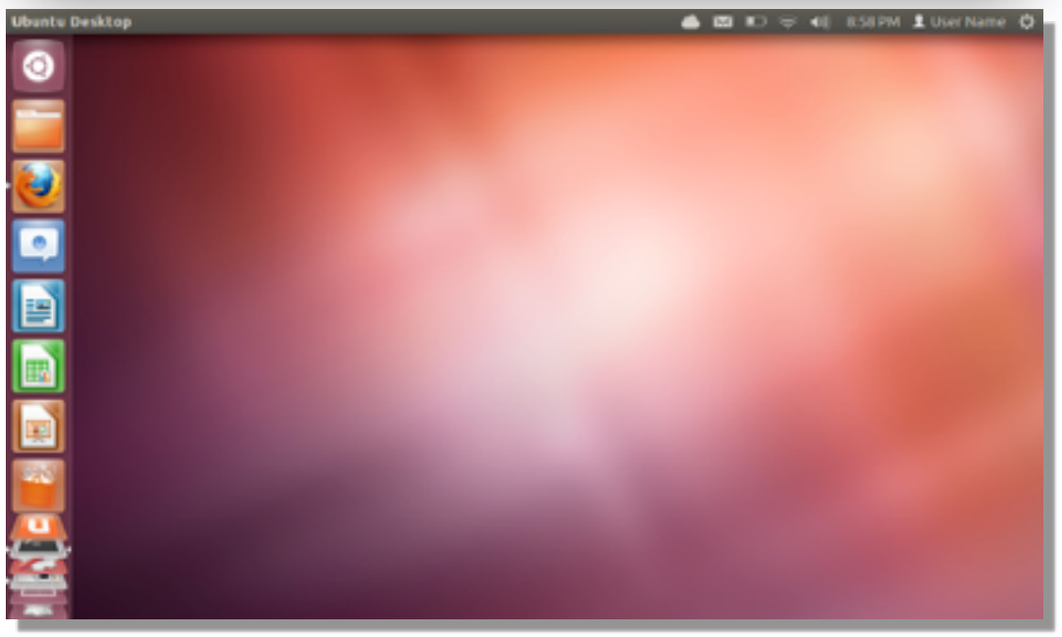


Shell

- In Linux the shell handles interaction between Users and the system.
- It can by via a text interface know as the command line as well as through a graphical interface, for instance the “gnome-shell”

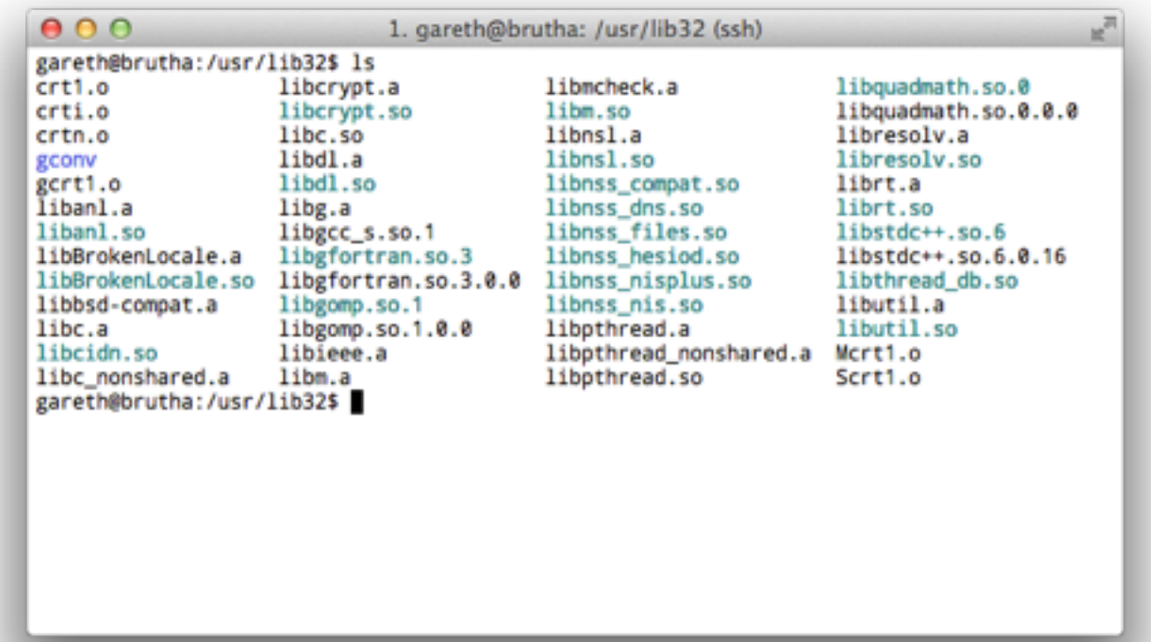


- It allows other process/ applications to be started such as editors, compilers, office suites.
- We’ll cover the command line in more detail later in this set of slides.



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.



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 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"!

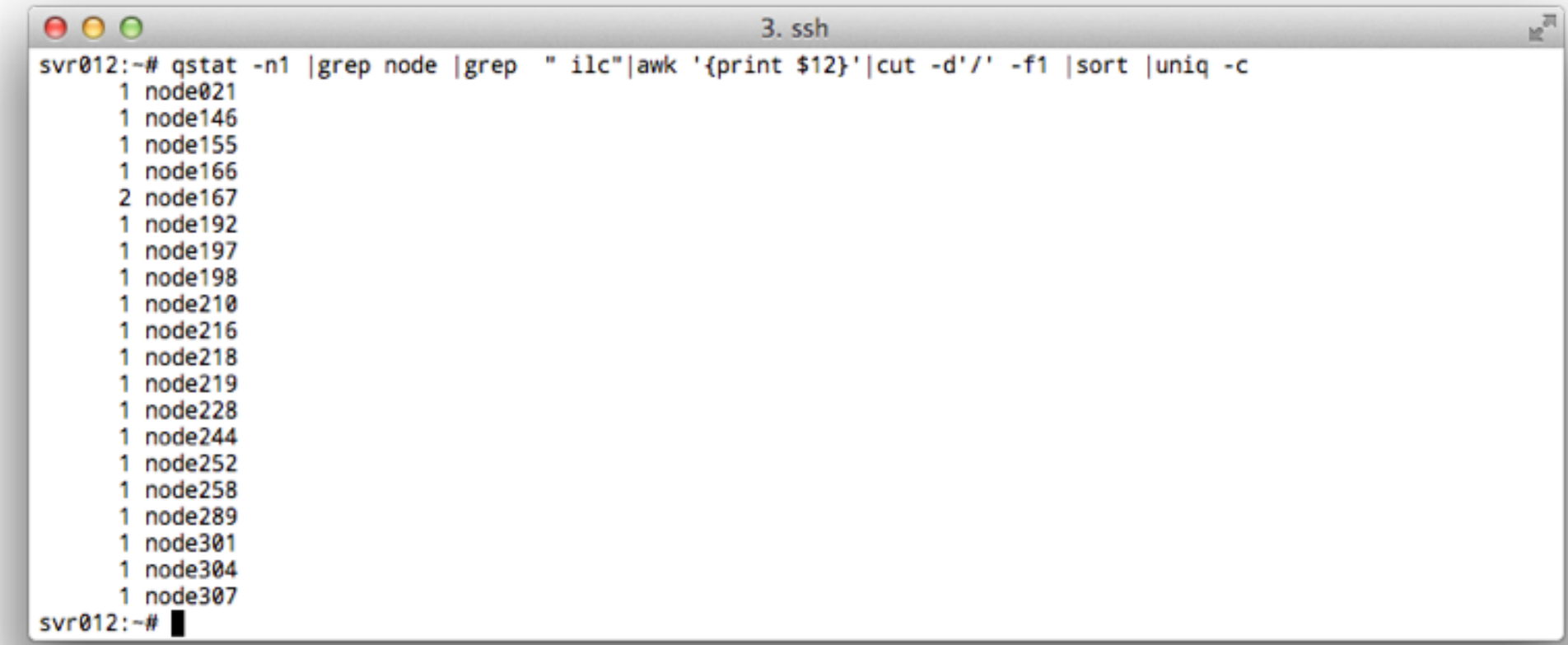
The Command Line

- 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>

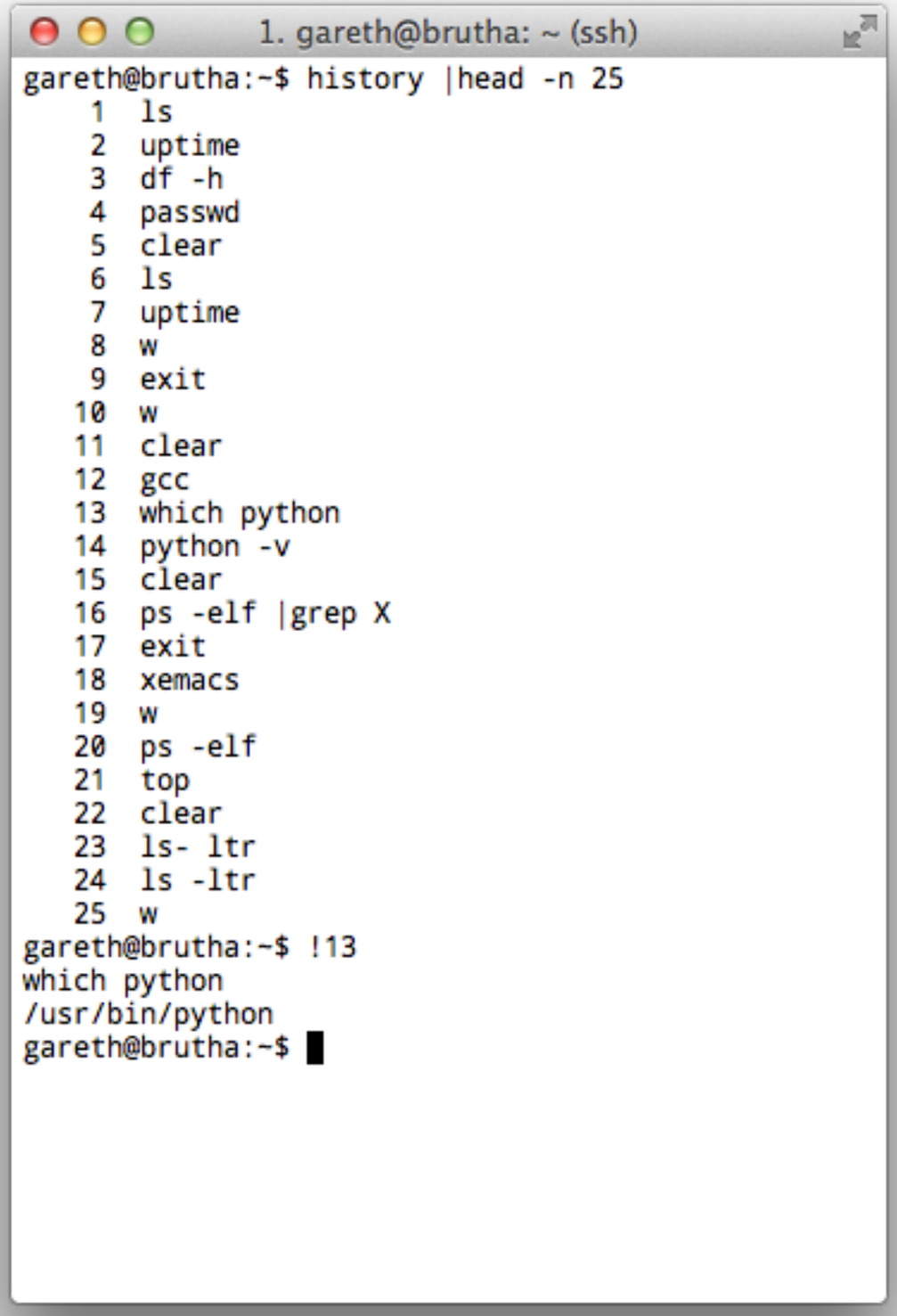
  prompt# du -h --max-depth=1

- Commands can be chained together to build up complex interactions and obtain the results we want.



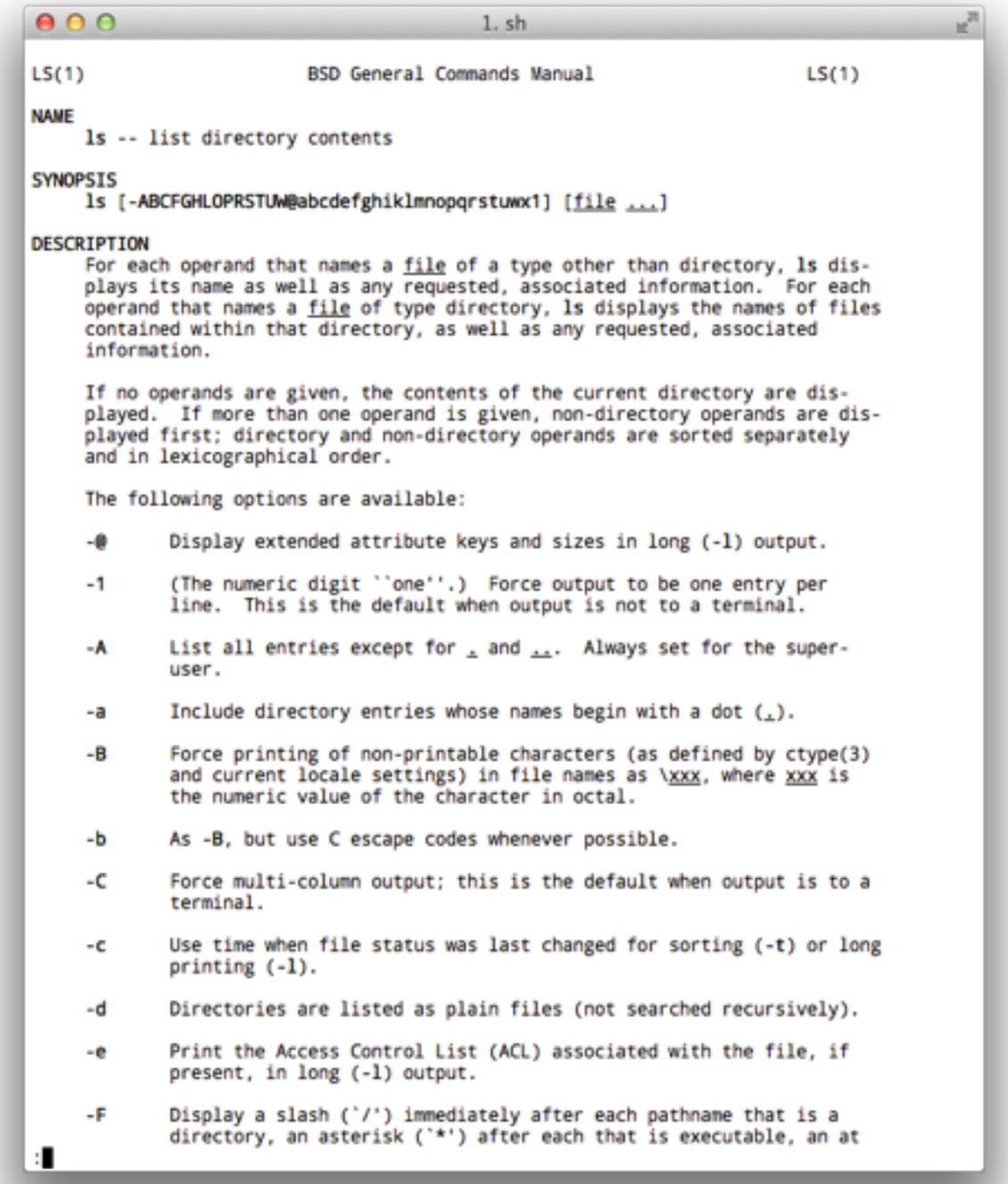
Useful Commands

- **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



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).



- 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

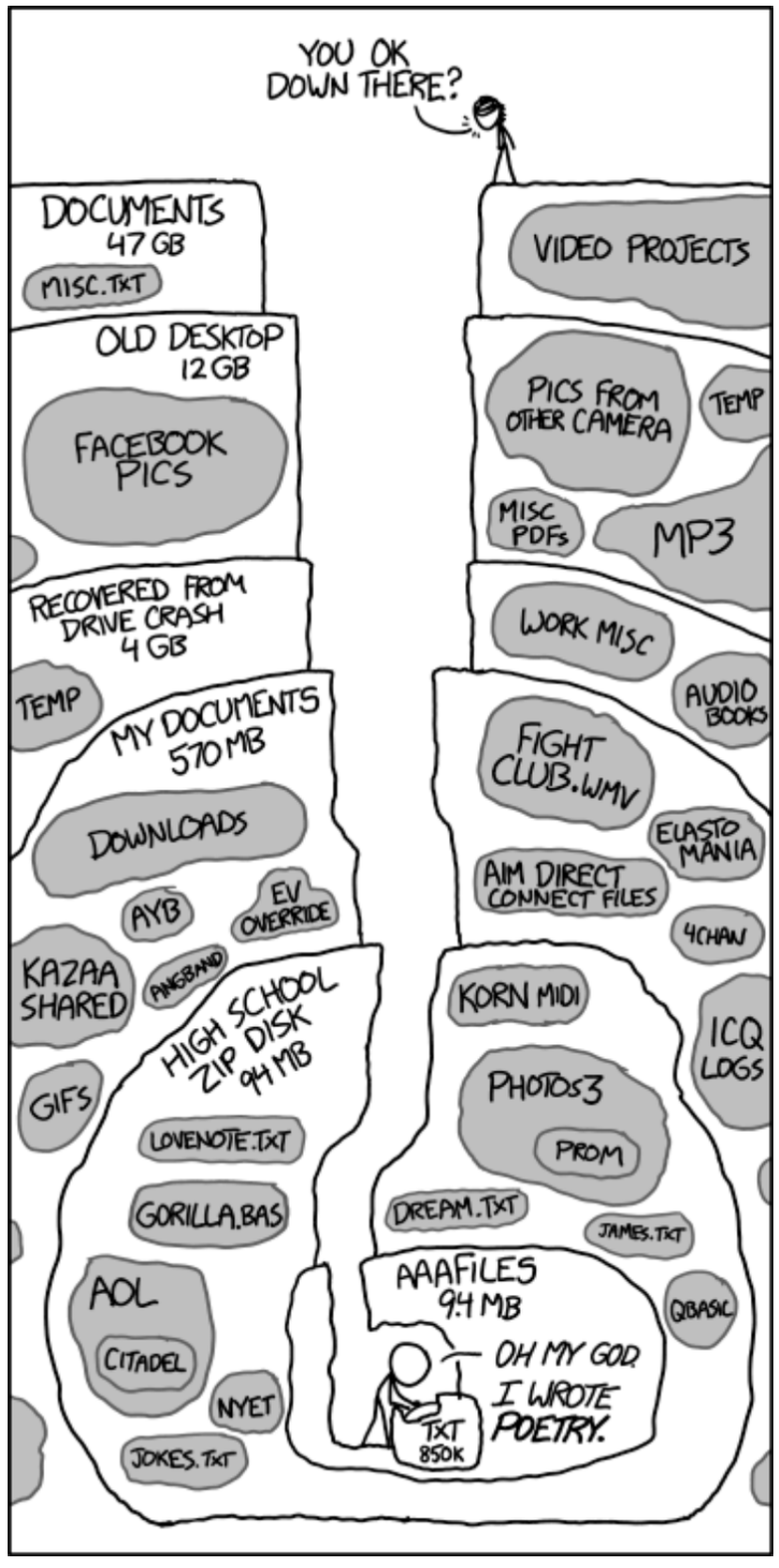
- The Filesystem
- Navigating the filesystem
- Users & Permissions
- Working with files
- Archives

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

— Ken Thompson

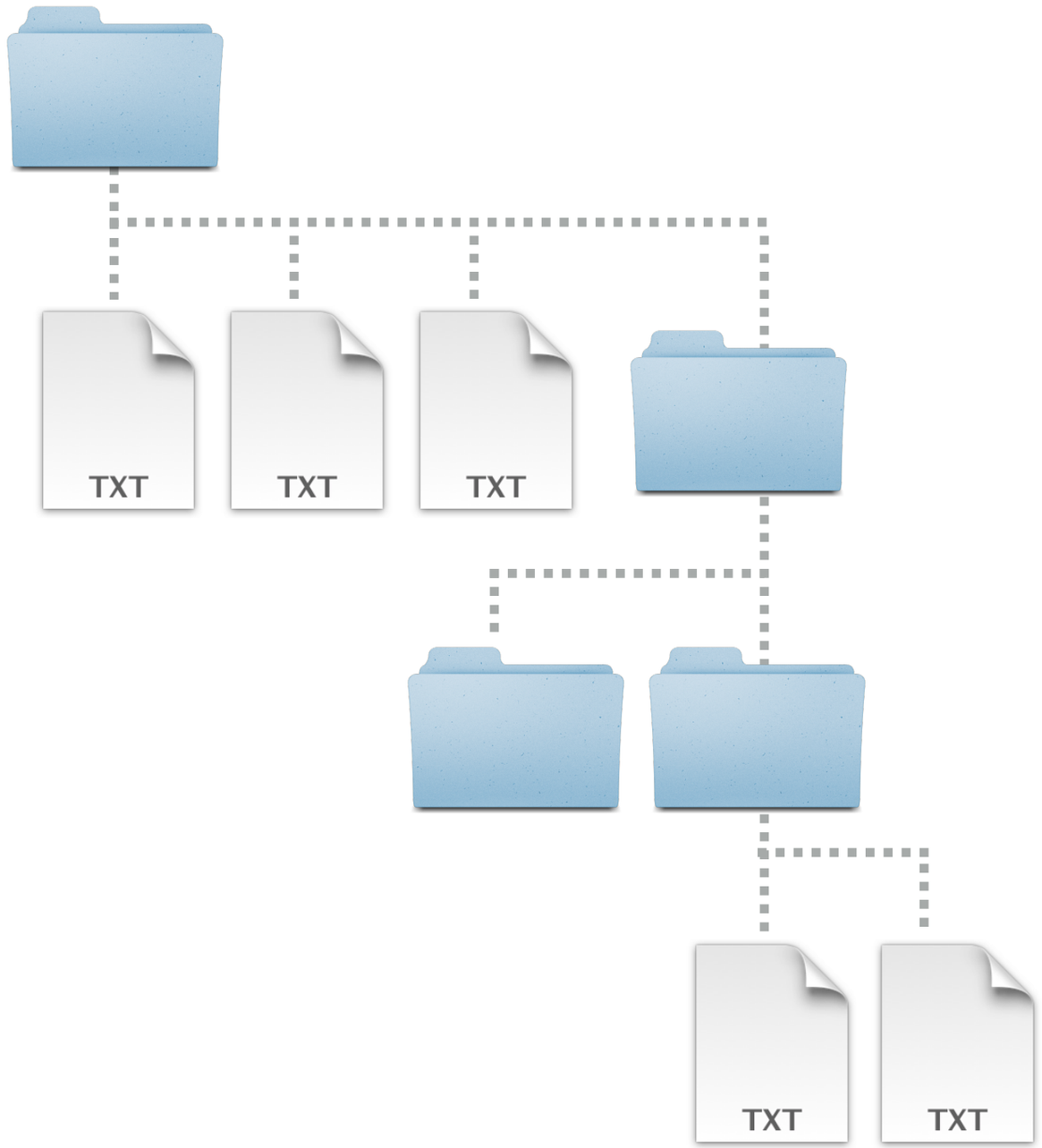
"On a UNIX system, everything is a file; if something is not a file, it is a process."

— Unknown


YOU OK DOWN THERE?
DOCUMENTS 47 GB
MISC.TXT
OLD DESKTOP 12 GB
FACEBOOK PICS
RECOVERED FROM DRIVE CRASH 4 GB
TEMP
MY DOCUMENTS 570 MB
DOWNLOADS
AYB
EV OVERRIDE
KAZAA SHARED
ANGSBAND
HIGH SCHOOL ZIP DISK 94 MB
GIFS
LOVENOTE.TXT
GORILLA.BAS
AOL
CITADEL
NYET
JOKES.TXT
VIDEO PROJECTS
PICS FROM OTHER CAMERA
TEMP
MISC PDFs
MP3
WORK MISC
AUDIO BOOKS
FIGHT CLUB.WMV
ELASTO MANIA
AIM DIRECT CONNECT FILES
4CHAN
KORN MIDI
ICQ LOGS
PHOTOS3
PROM
DREAM.TXT
JAMES.TXT
AAAFILES 9.4 MB
QBASIC
OH MY GOD I WROTE POETRY.
TXT 850K

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**.

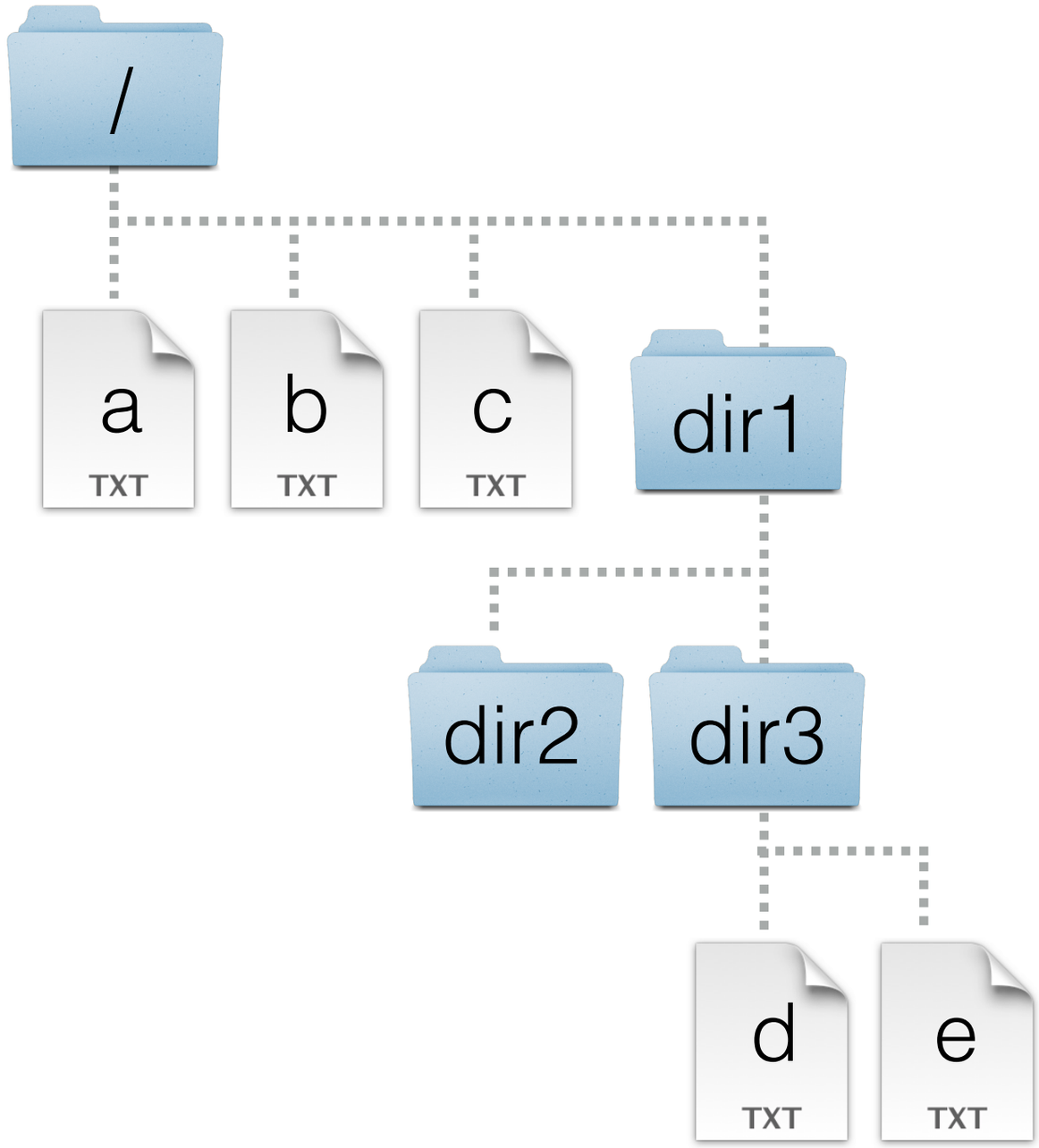


File types

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.

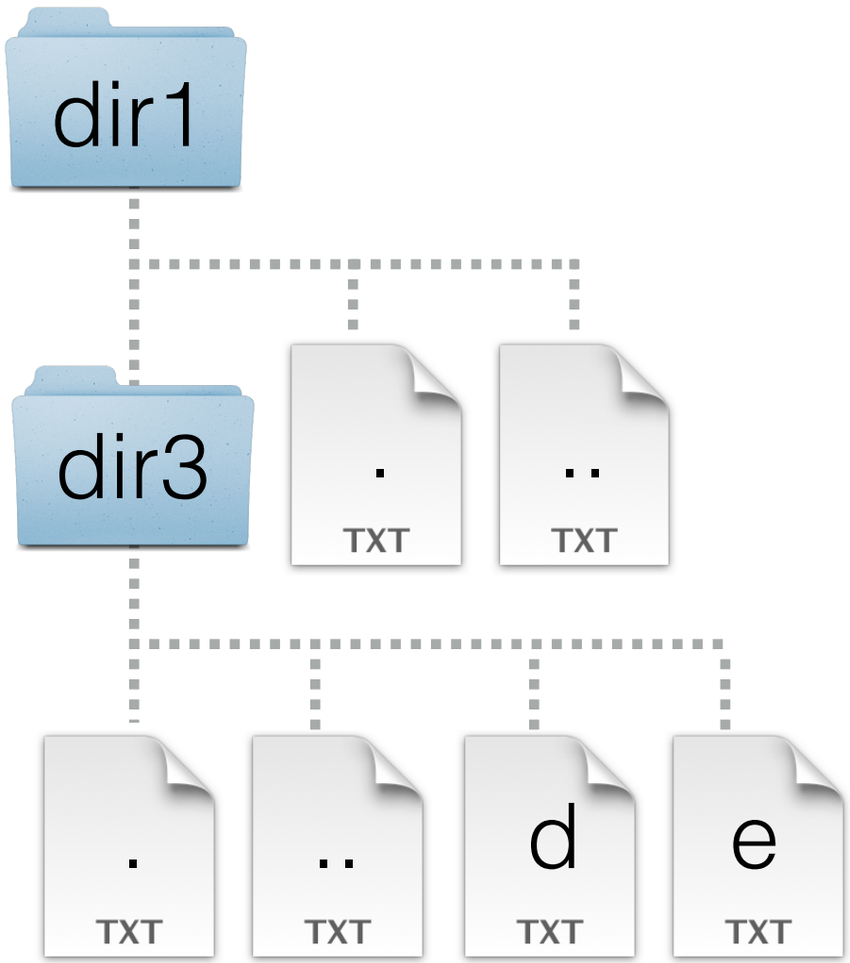
Absolute File Paths

- 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.



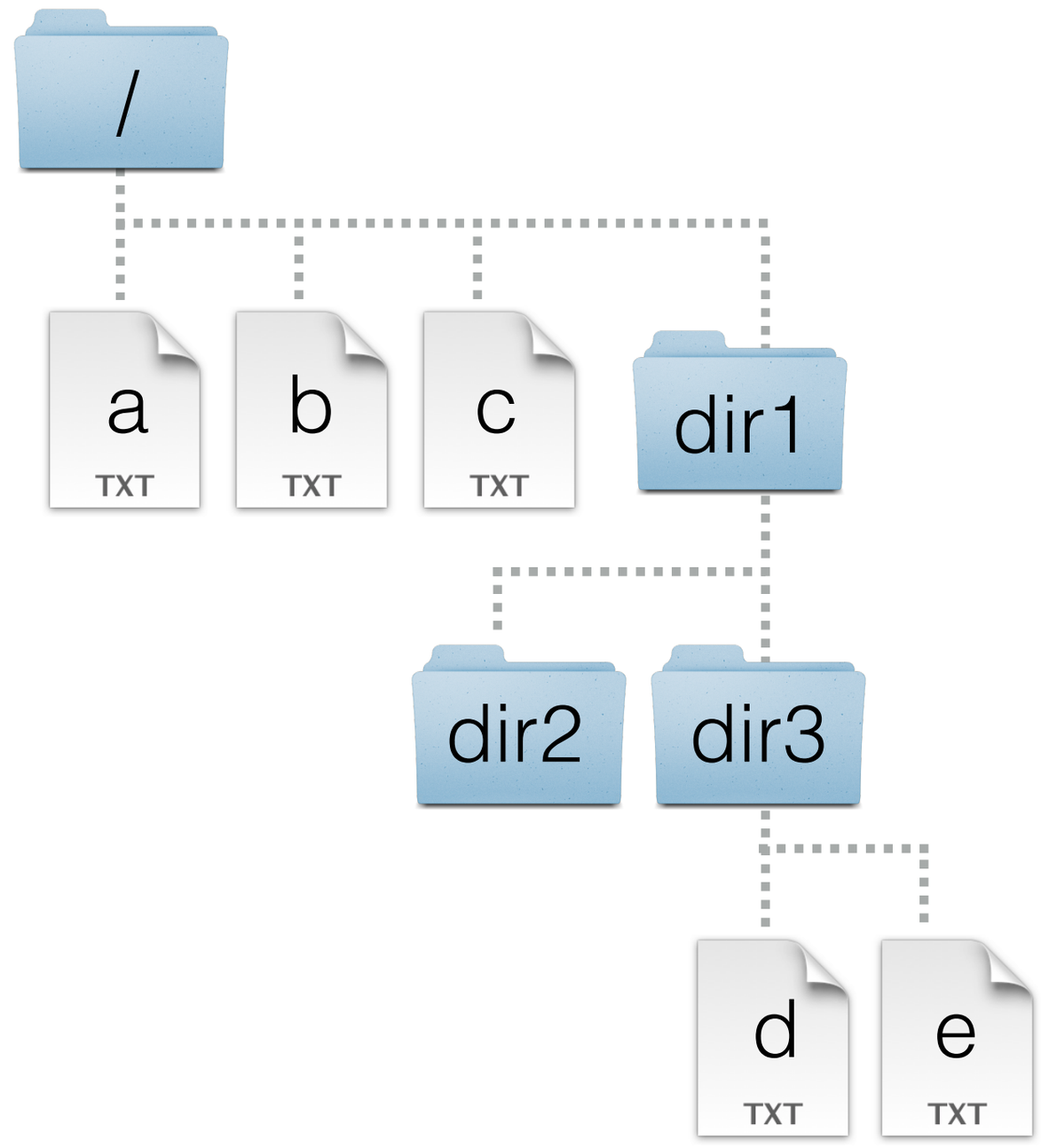
Hidden Files

- 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 fill '**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

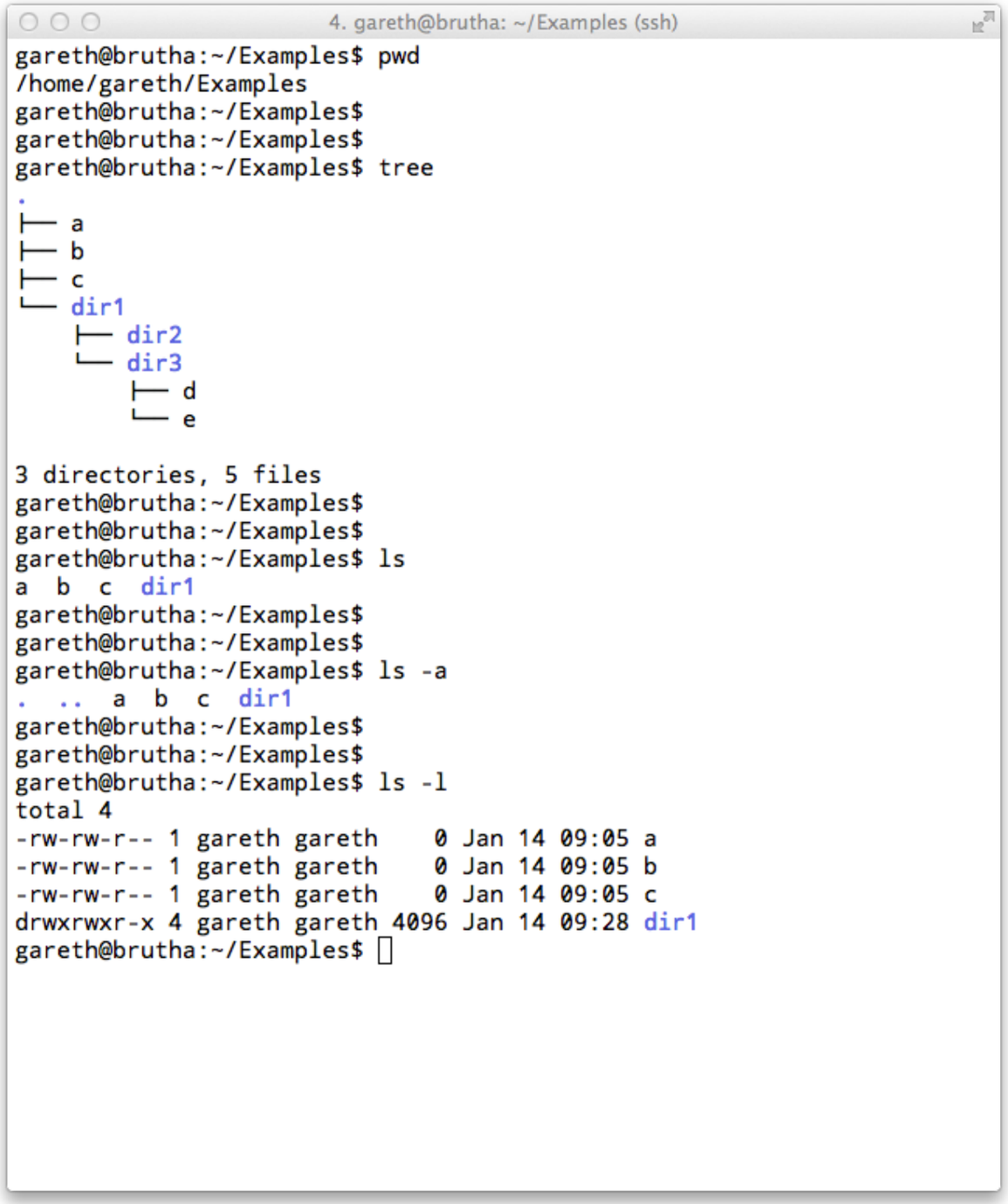
- **pwd** - tells us our current position in the filesystem (print working directory).

- **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.

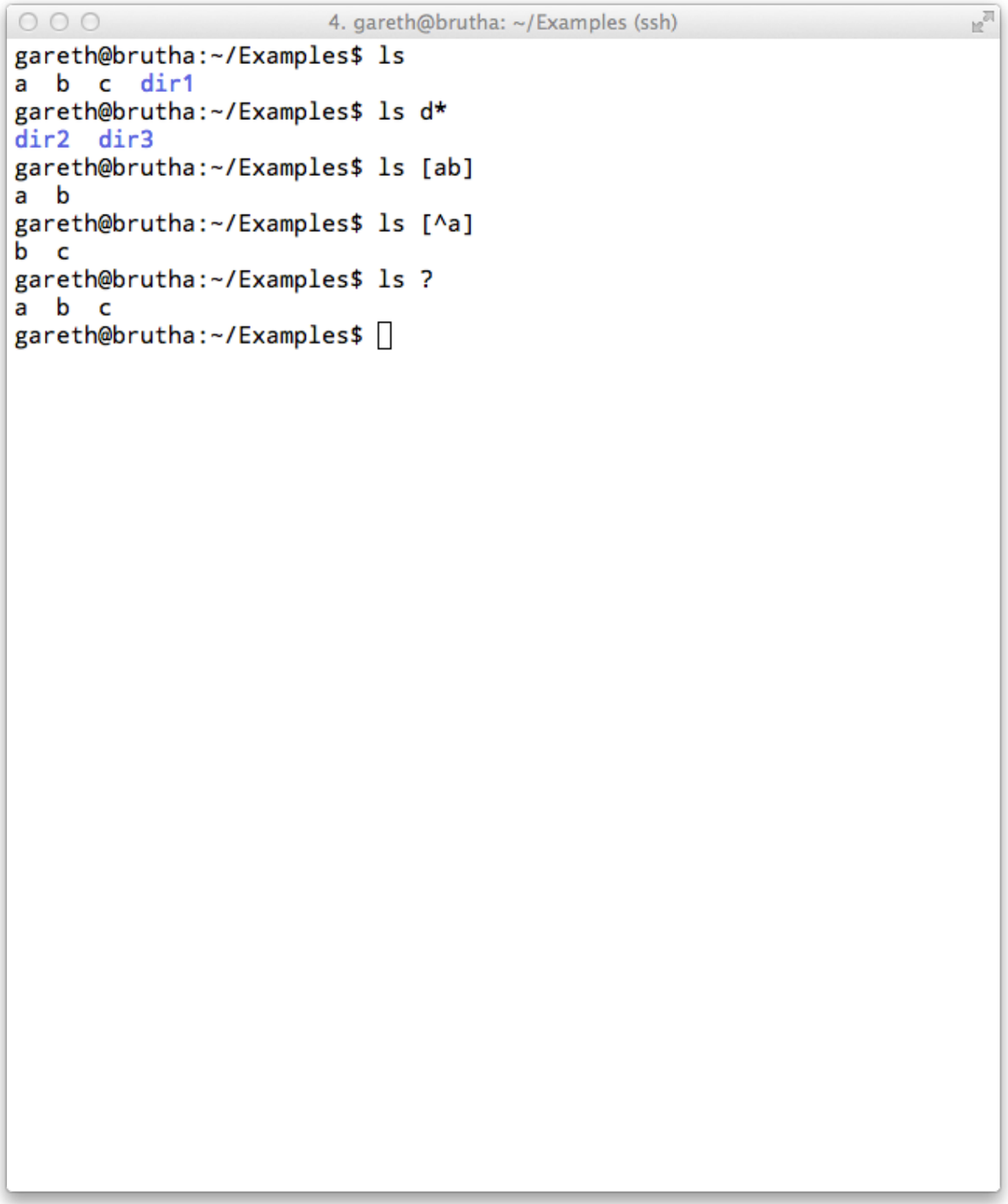
- **ls -a** - the minus a option tells ls to show hidden files.

- **ls -l** - the minus l option tells ls print the contents of a directory in long format.



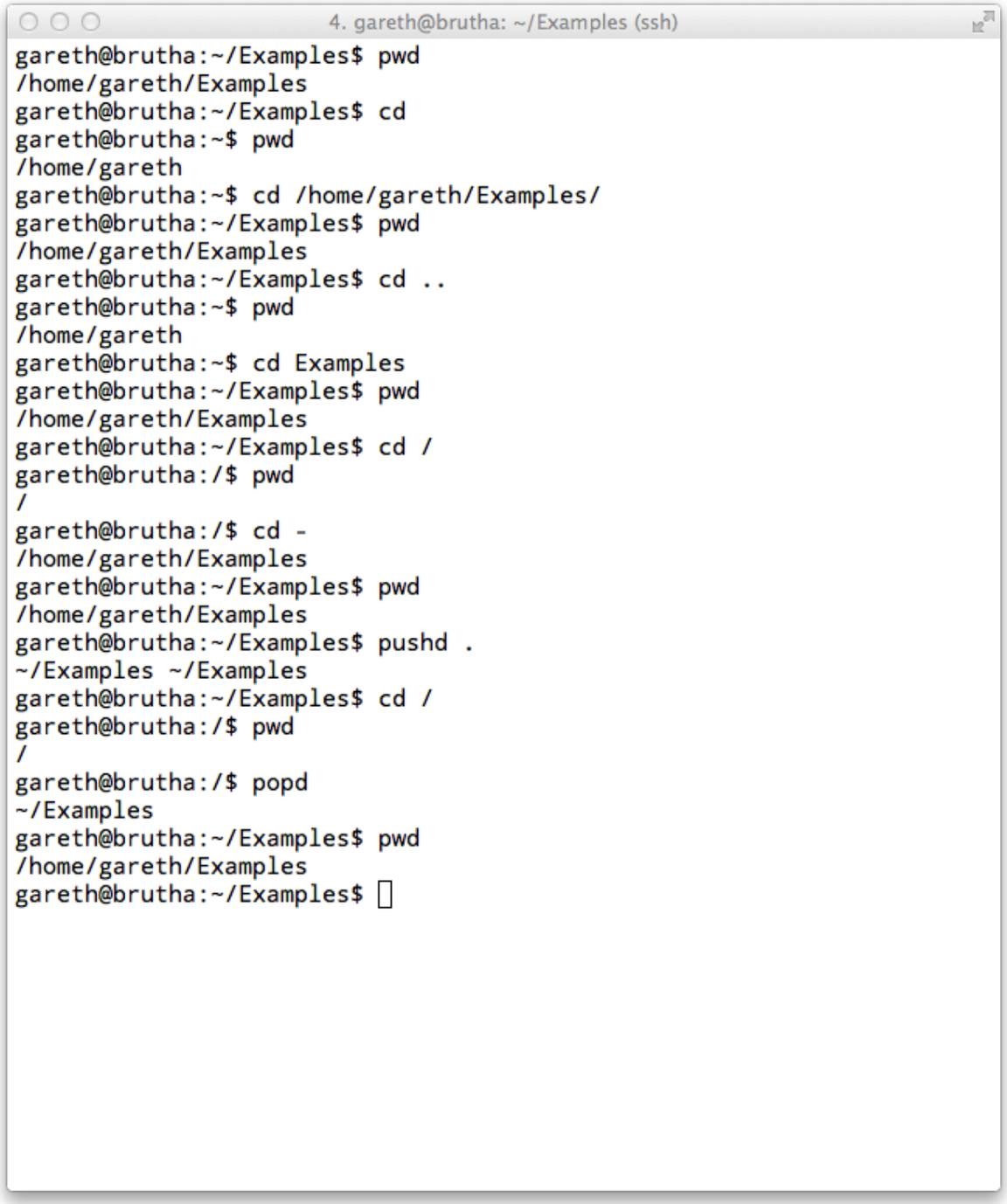
File Globbing

- **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 ?**



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.



Finding Files

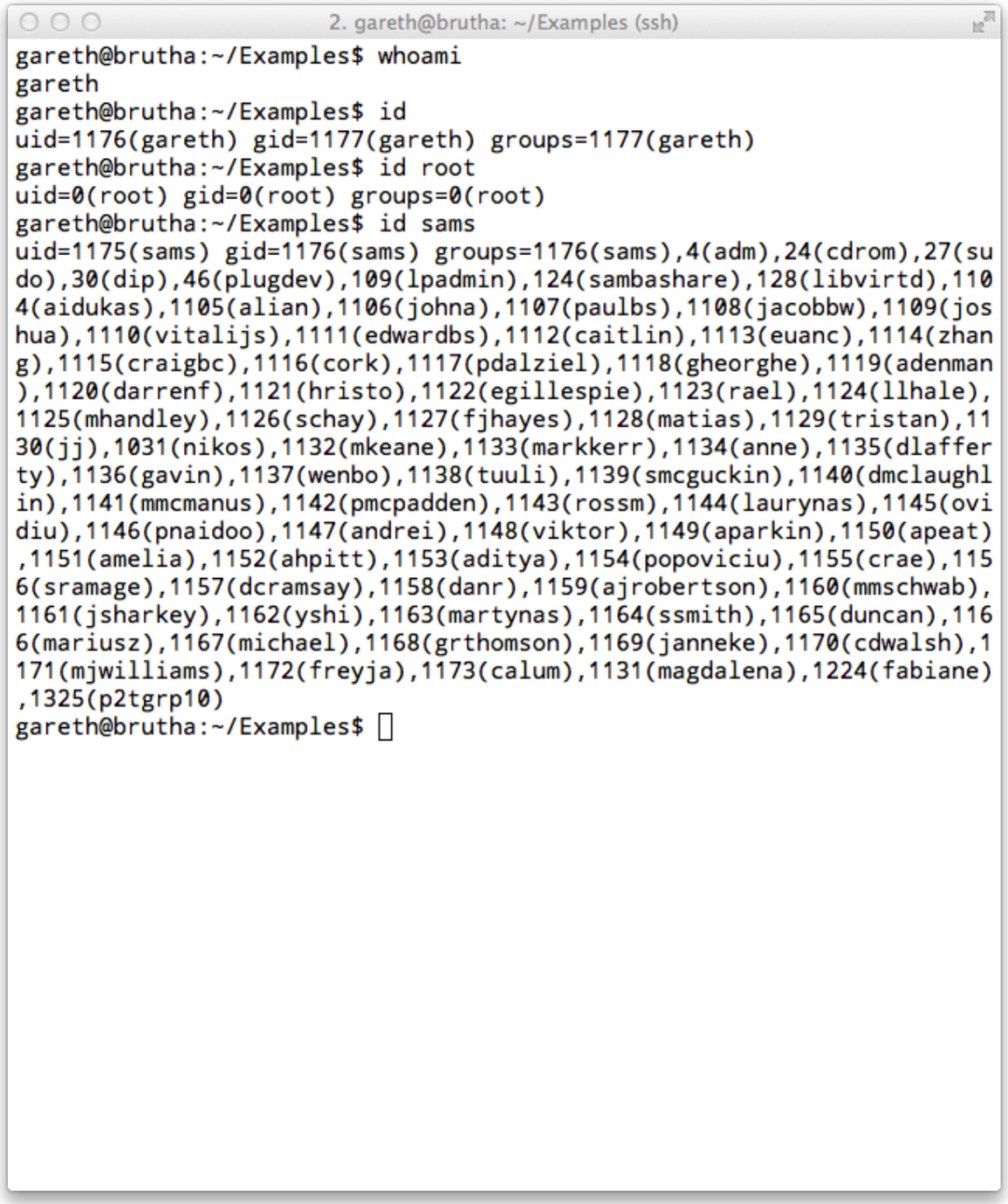
- **find** - used to search through the filesystem to find a file.
  - find ./ -name Lab04
  - find ./ -mtime +2
  - find ./ -mtime -3
- **locate** - accesses a database of files (usually built once a day), is faster than find but only as good as the database.
- Use find to get newer files, and if you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

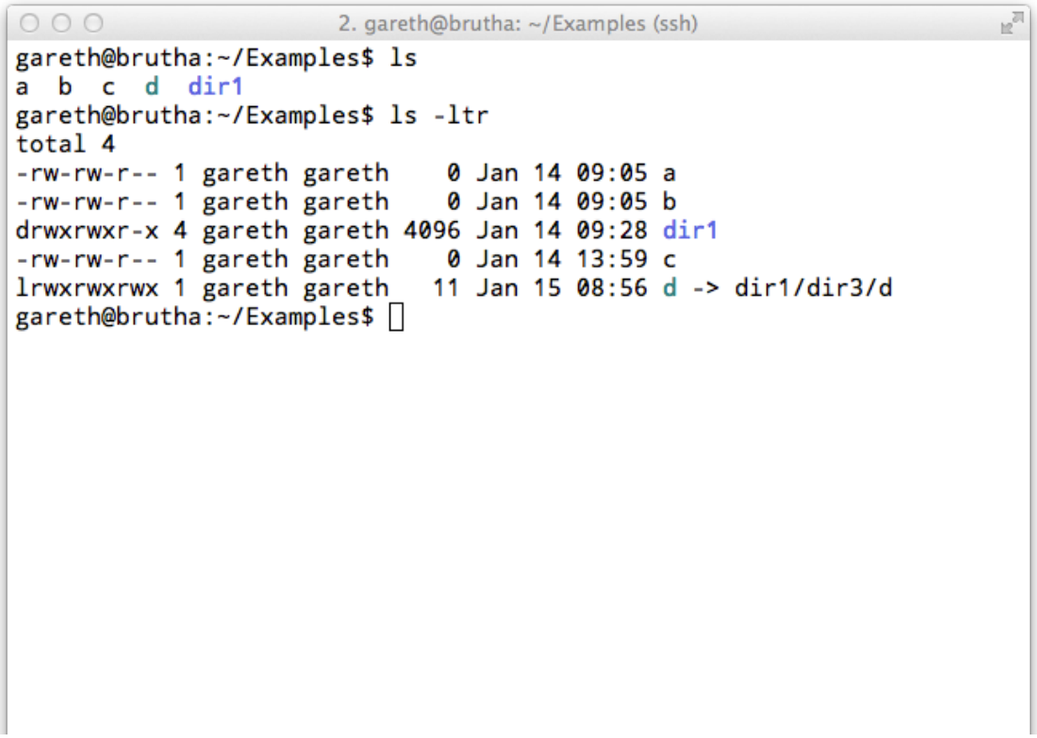
Users & Groups

- 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).



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 place 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 -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

- 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

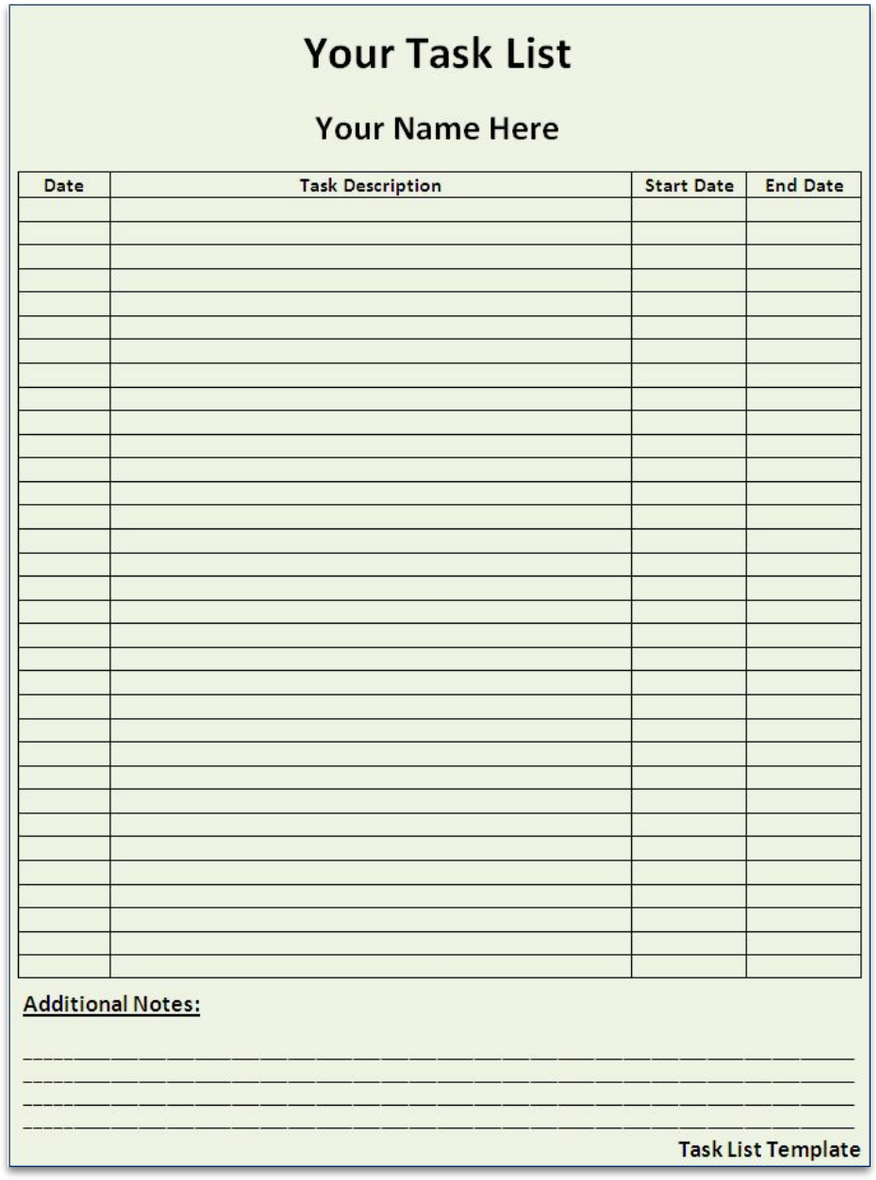
```
chmod [who][op][what] filename
```

- 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



Your Task List

Your Name Here

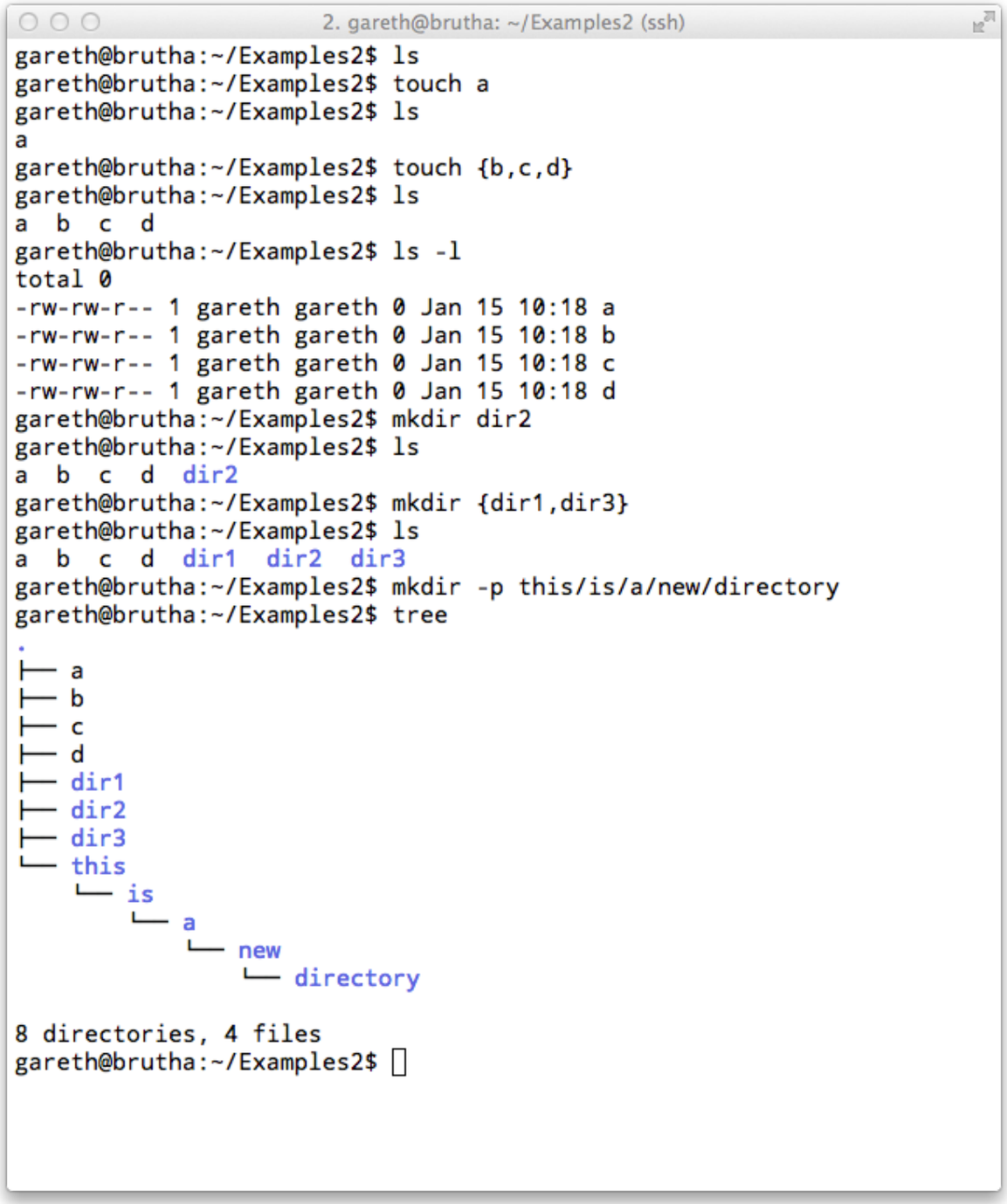
Date	Task Description	Start Date	End Date

Additional Notes:

Task List Template

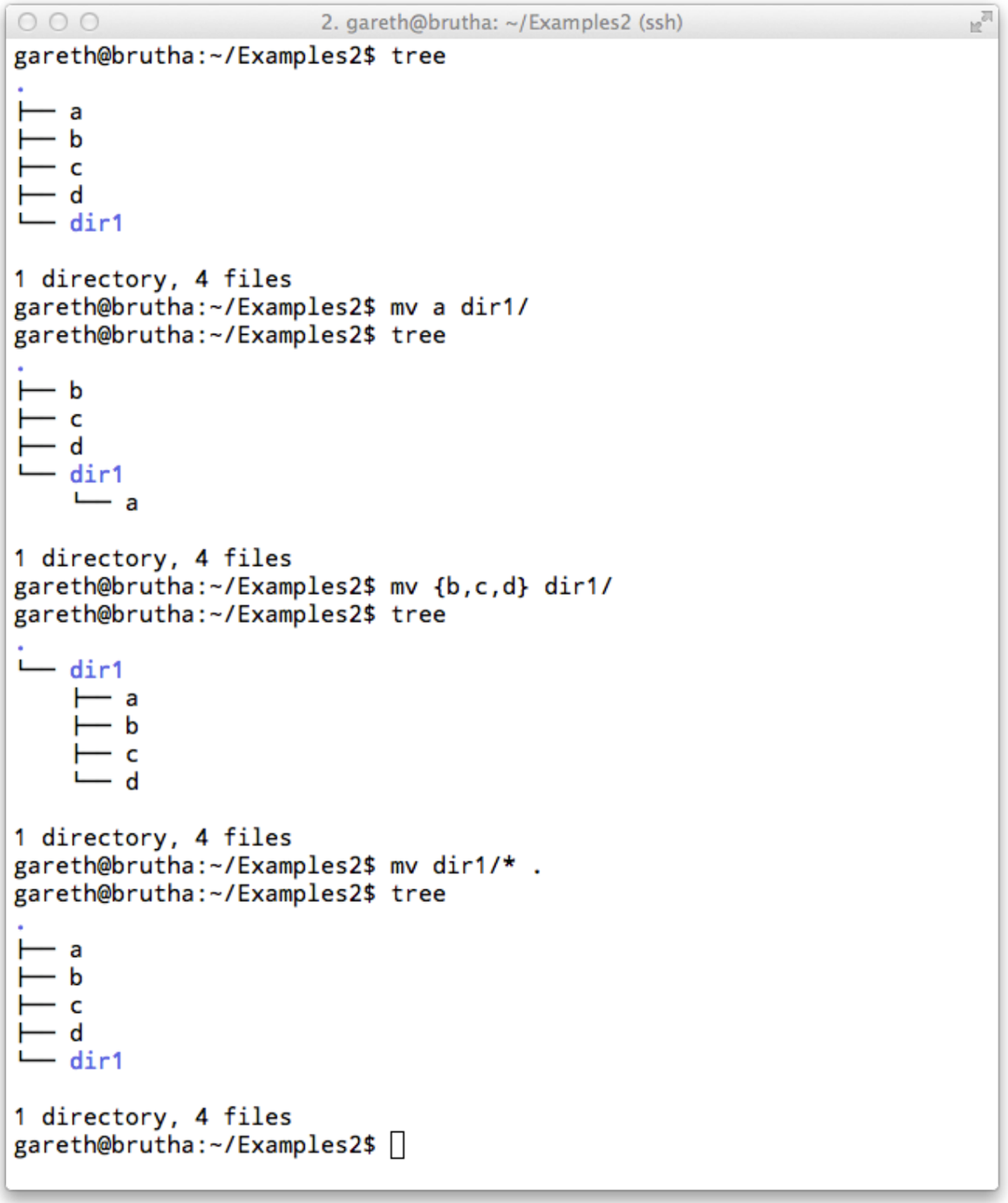
File Creation

- 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



Moving a File

- A file can 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



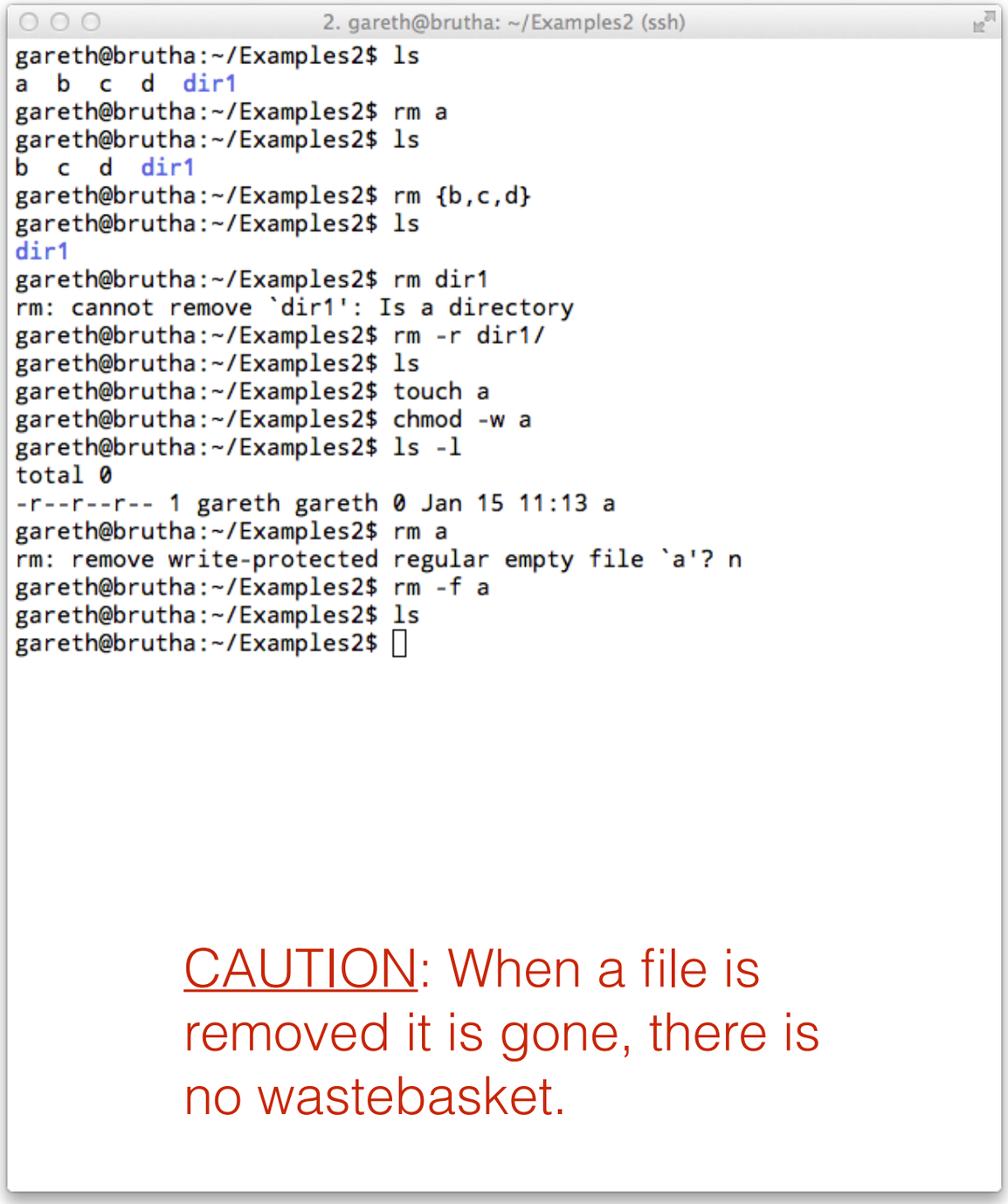
Copying a File

- A file can 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

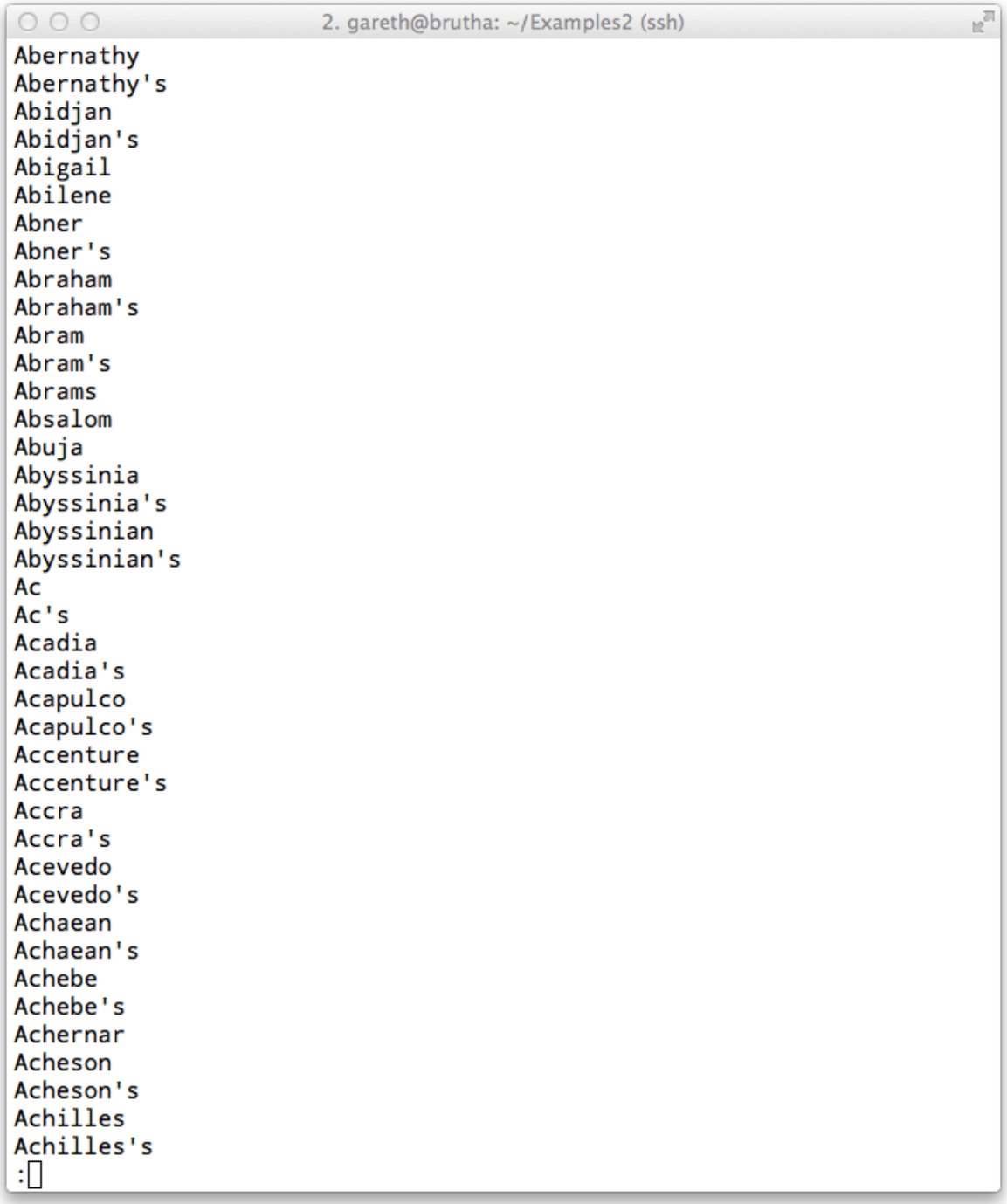
- A file can deleted or removed by:
  - **rm** <filename>
- Where <filename> can be either an absolute or relative path.
- Multiple files can be removed by giving a list or a glob:
  - **rm** {a,b,c,d}
  - **rm** dir1/*
- Like copying removing directories is a little different, you need to tell rm that 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



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

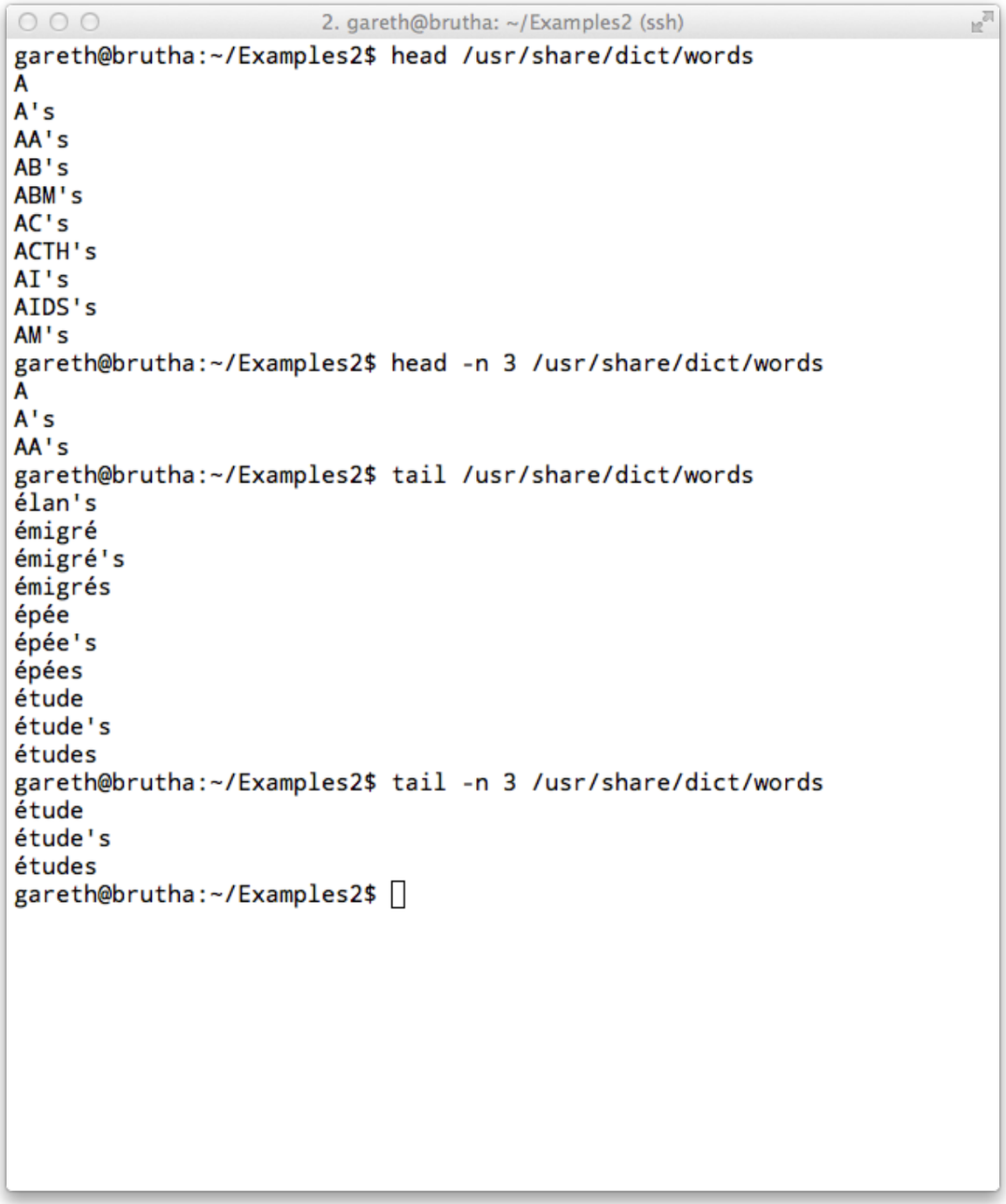
Looking at Files

- 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.



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>



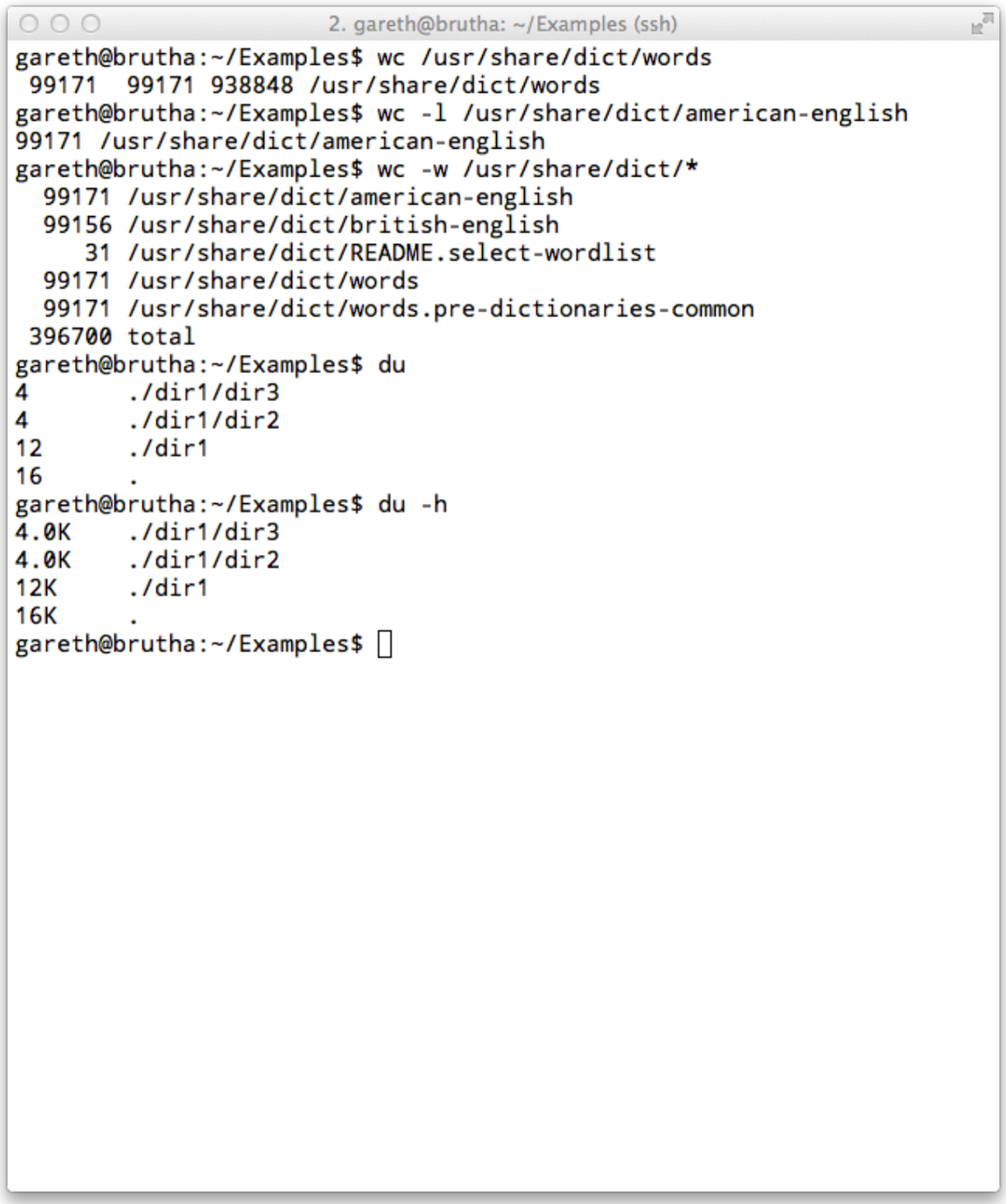
Searching in a File

- 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

```
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)

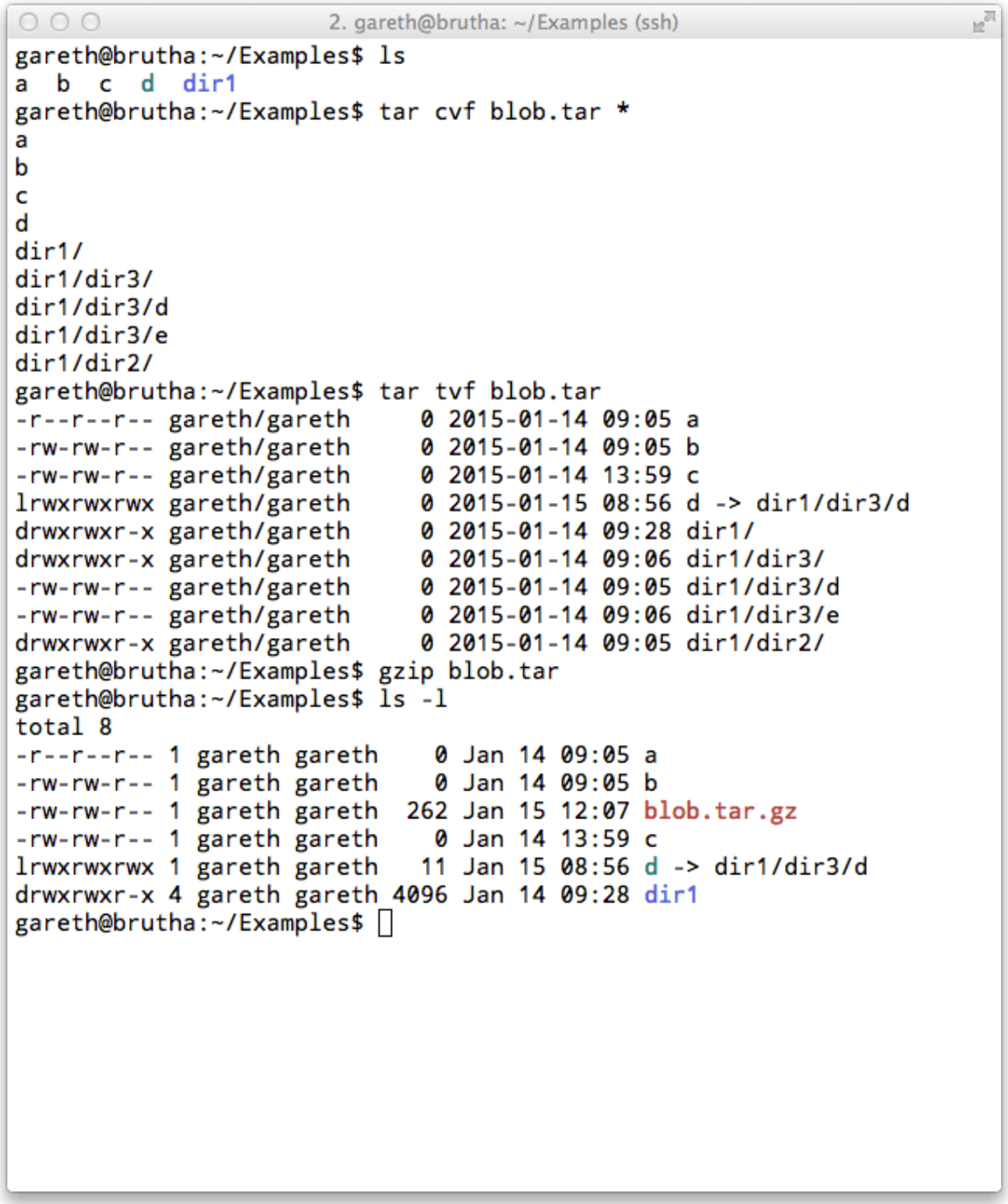


- You can sort the contents of a file using the sort command:
  - **sort** <filename>
- If you want to find out how much storage your files are taking up you can do:
  - **du** <directory>
  - **du -h** <directory>

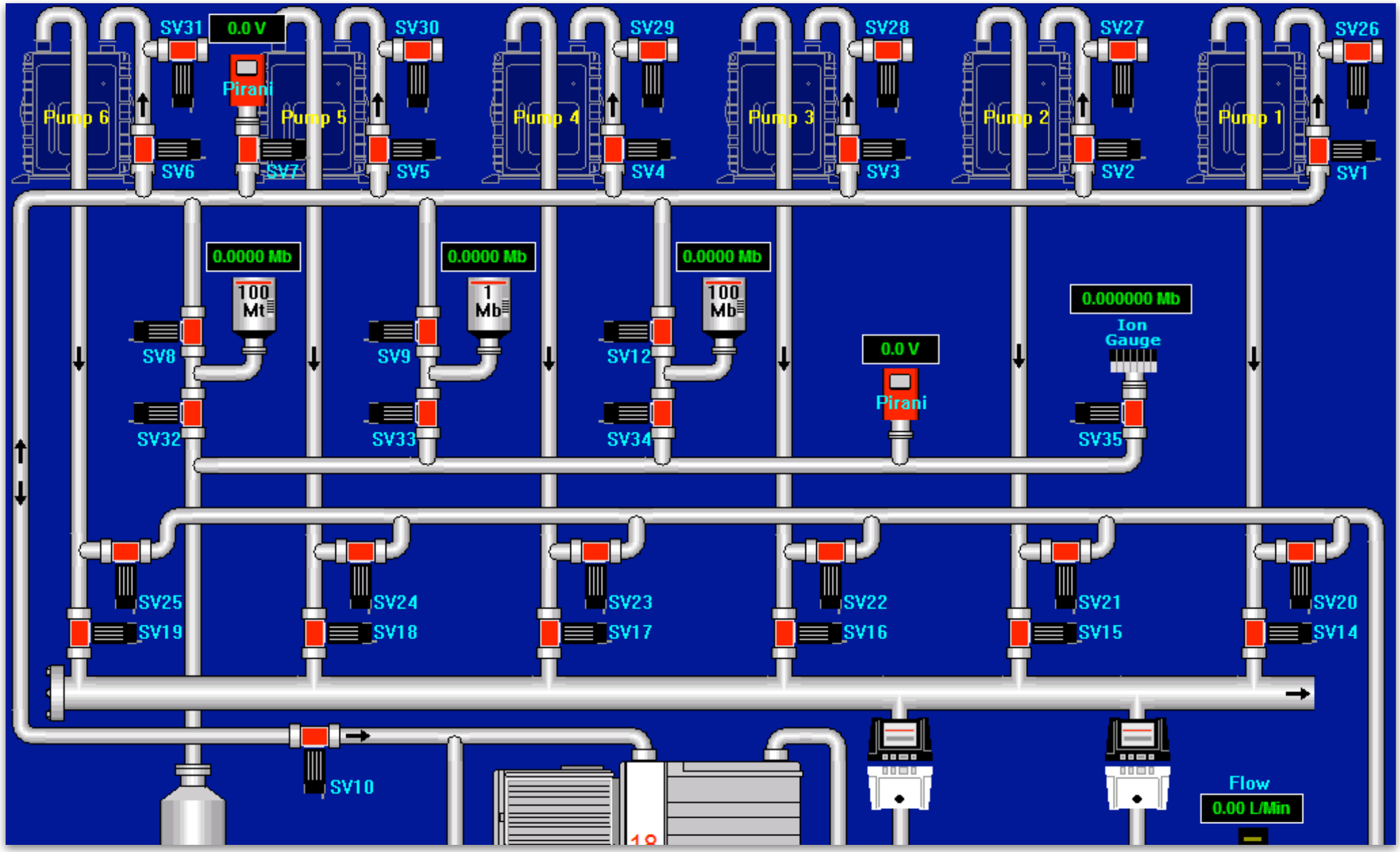
- The Filesystem
- Navigating the filesystem
- Users & Permissions
- Working with files
- Archives

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



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



Processes

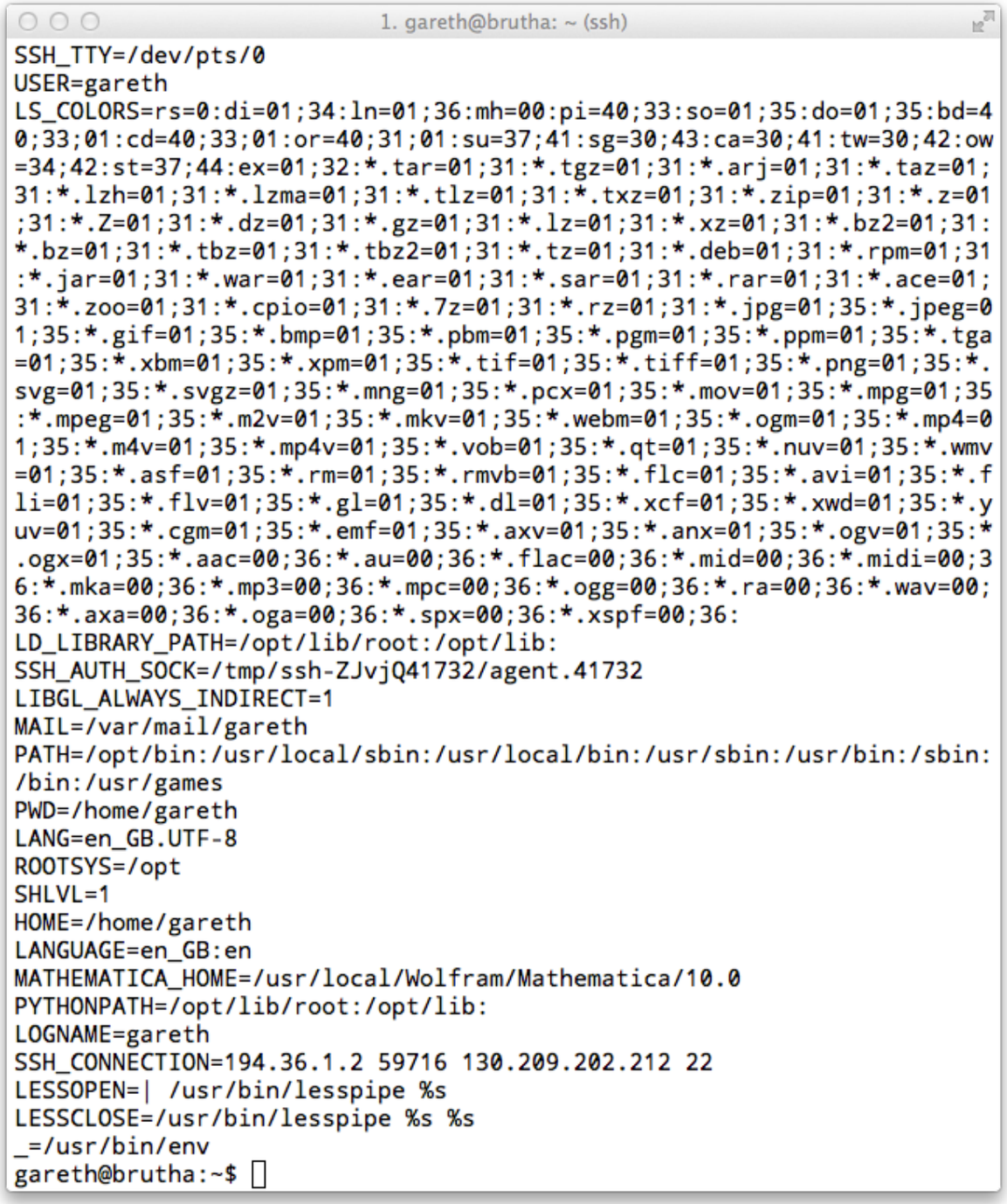
Dr. Gareth Roy (x6439)
gareth.roy@glasgow.ac.uk

- 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

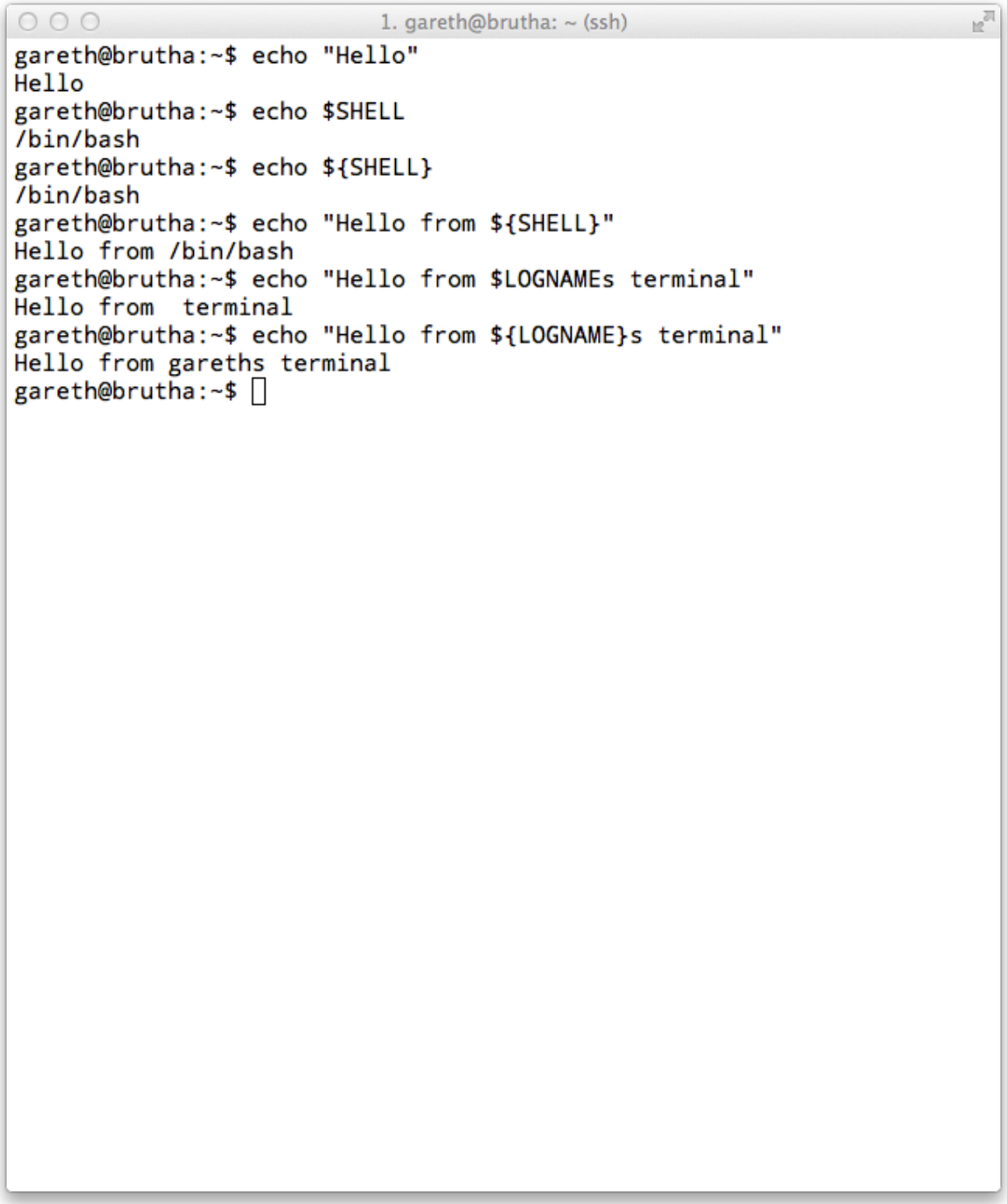
Variables

- 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:** their 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.



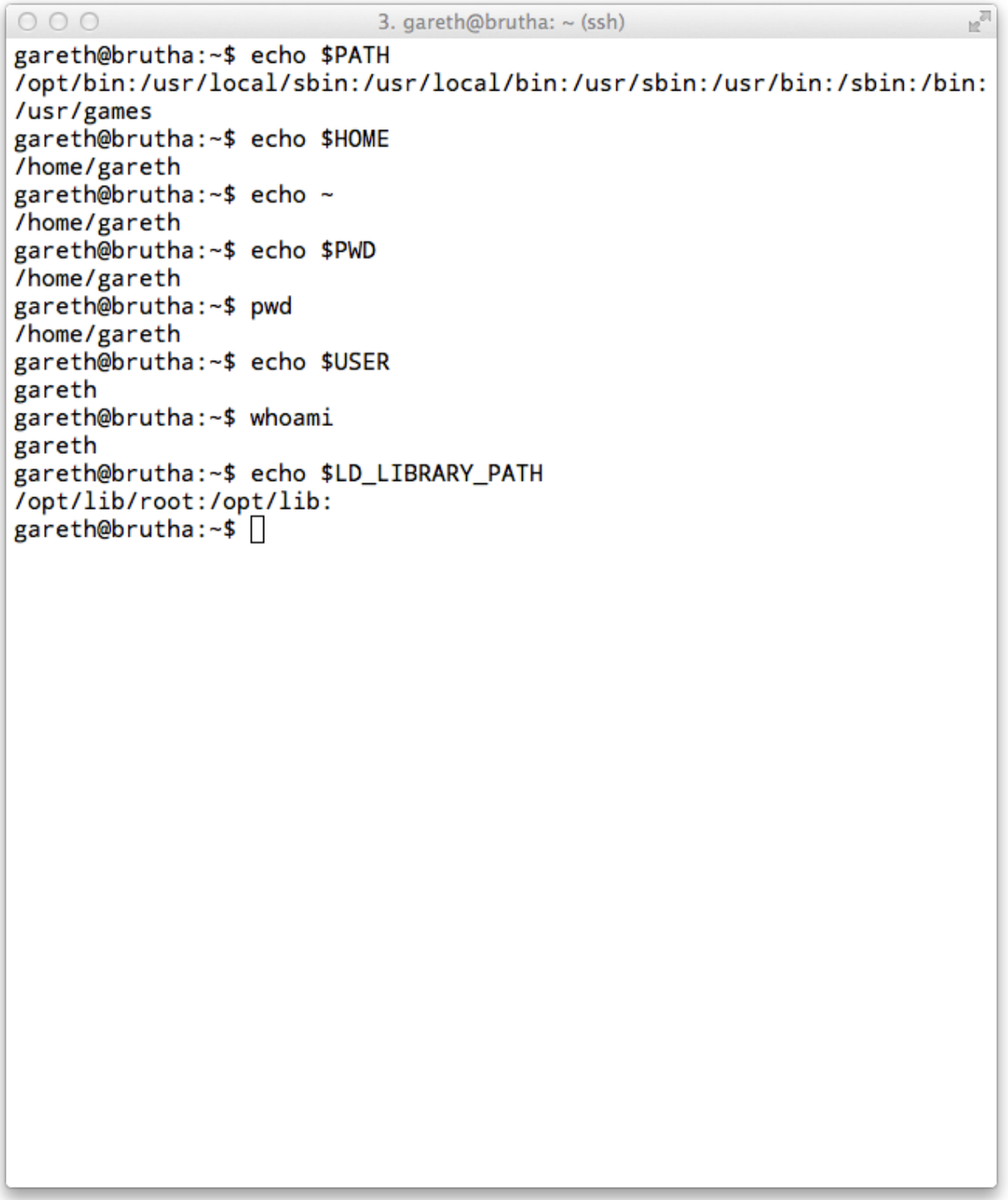
Variables

- 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 it’s 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}”



Important Variables

- Their 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.



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 classifer:
  - **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**

```
3. gareth@brutha: ~/Examples (ssh)
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 "$([ $? = 0 ] && echo termin
al || echo error)" "$(history|tail -n1|sed -e '\''s/^\s*[0-9]\+\s*//;s
/[;&|]\s*alert$//'\'')"'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -laF'
alias ls='ls --color=auto'
gareth@brutha:~/Examples$
```

Persistant Variables

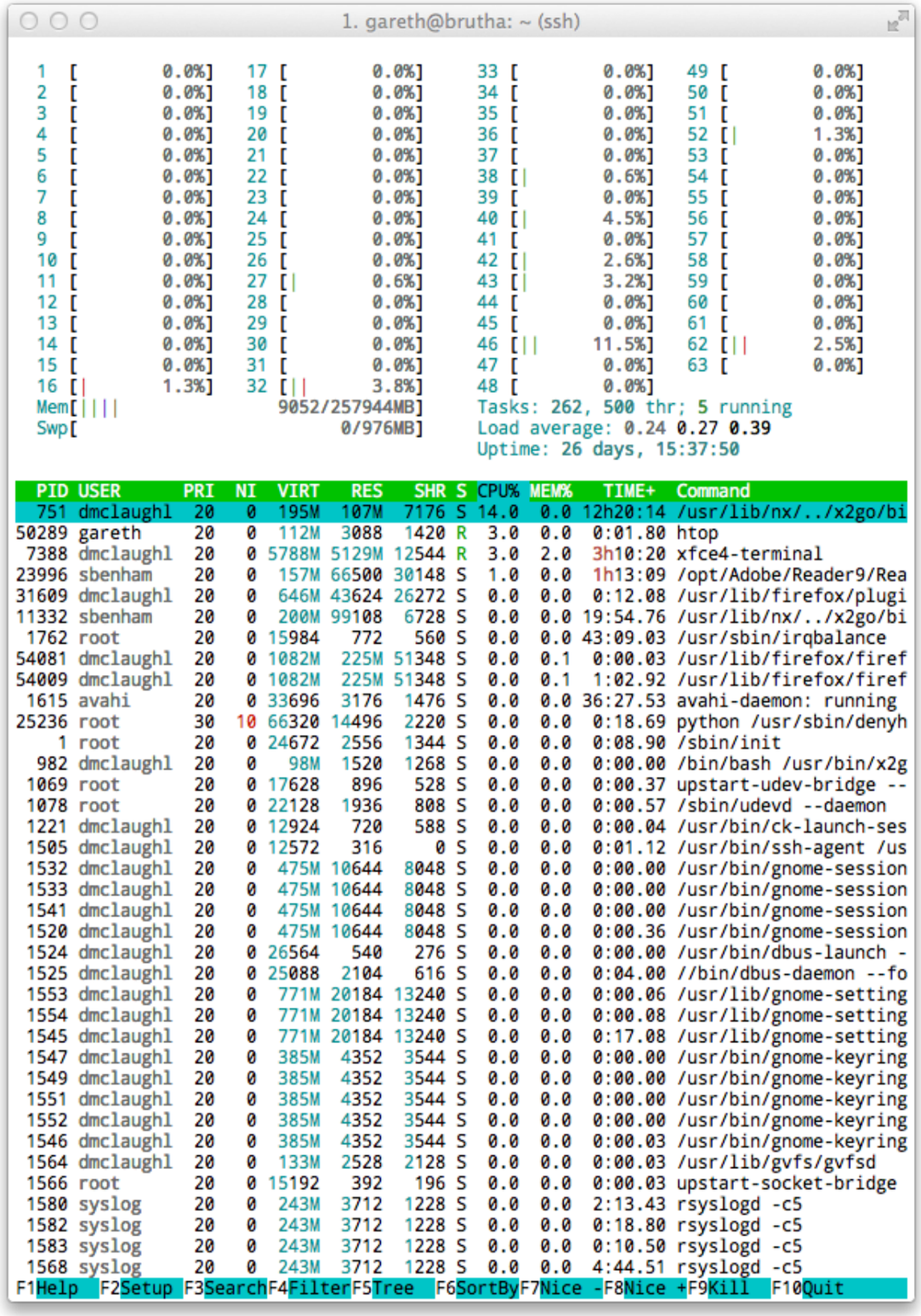
- All variables in **bash** shells are transient.
- Variables you set in one shell will not persist when you open a new one.
- When you open a new shell bash reads a number of configuration files to set it's state.
- We can make variables persist by adding them to one of these configuration files.
- Bash has two configuration files found in you home directory
  - **.bashrc**
  - **.bash_profile**
- The difference is subtle and most times you'll want to but 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

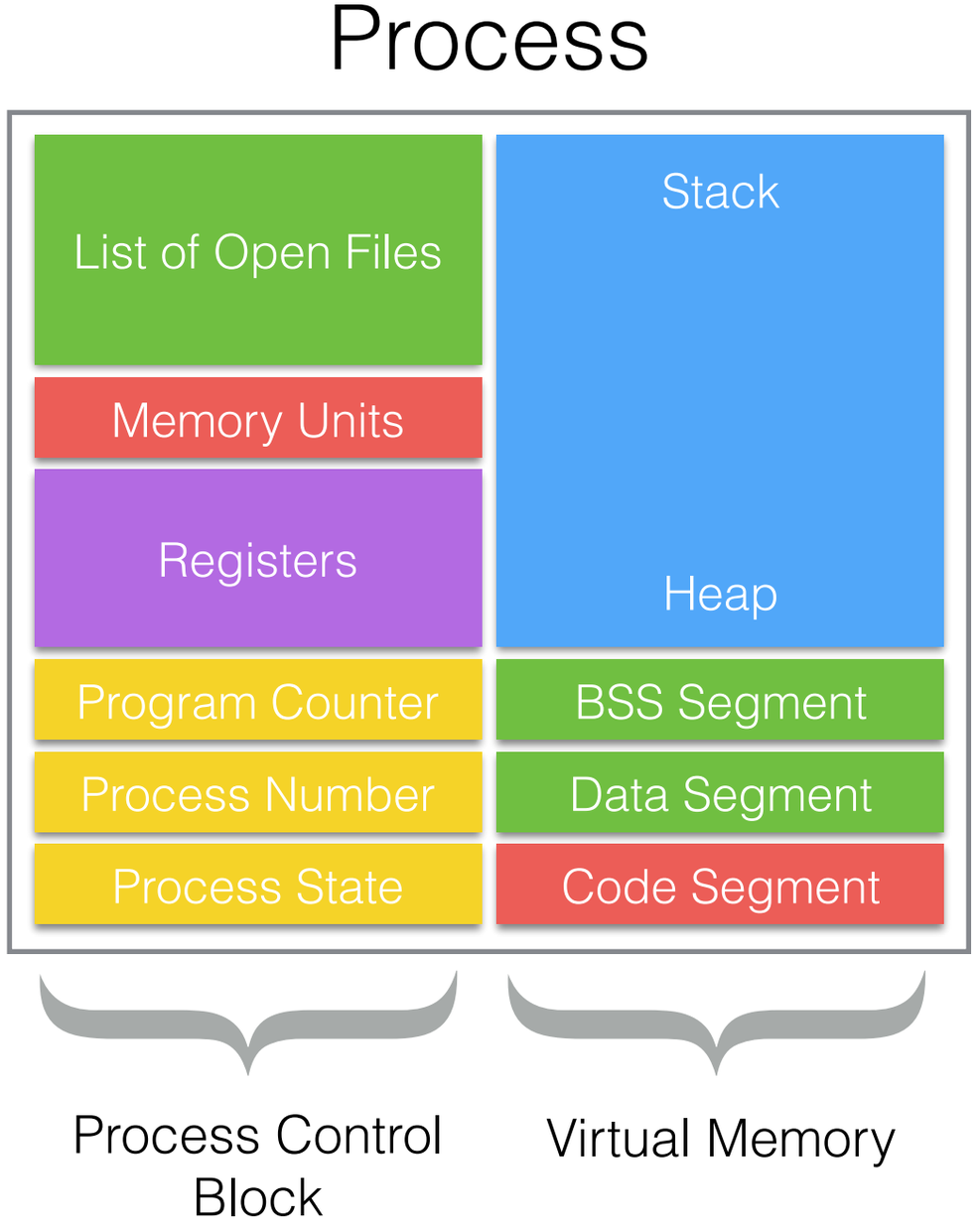
What is a Process?

- 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.



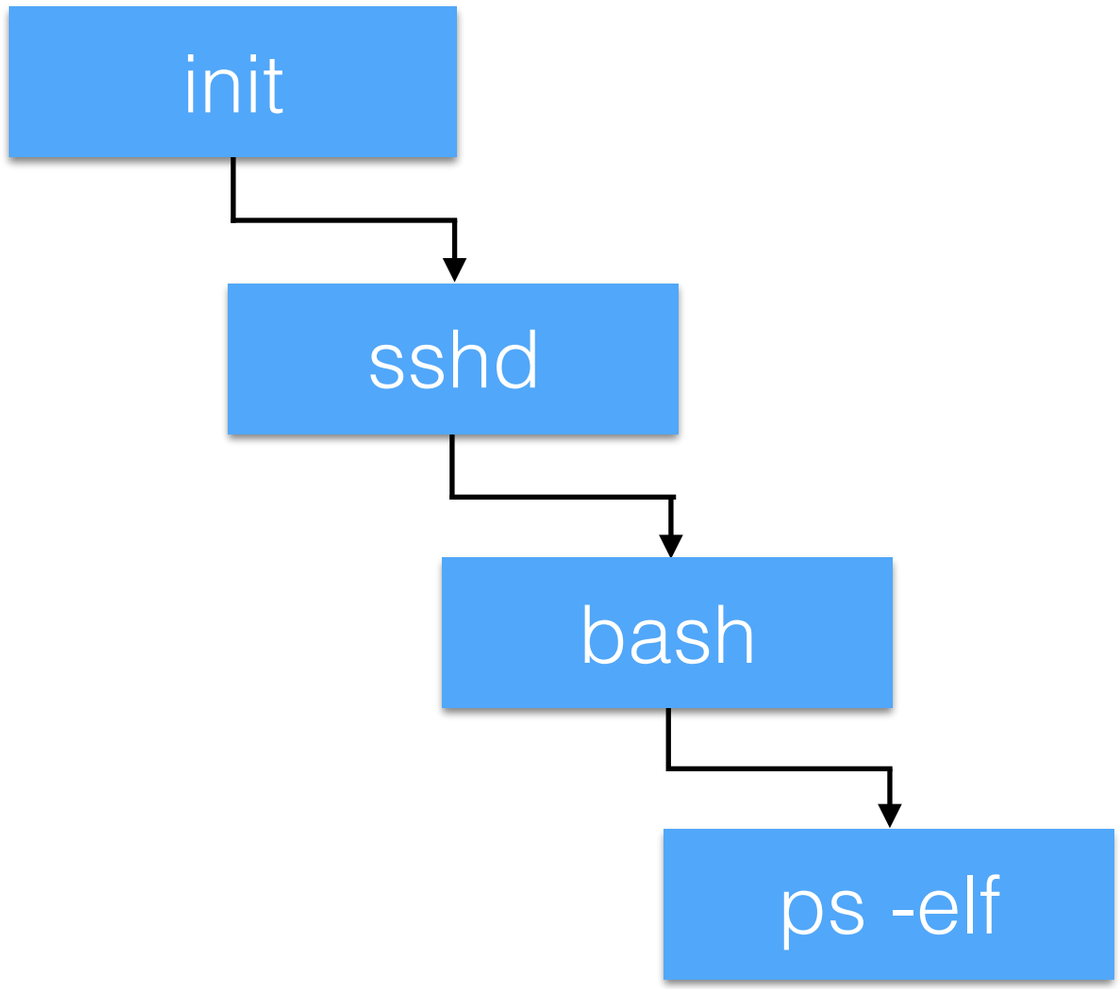
What's inside a Process?

- It contains:
  - a copy of the 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**"



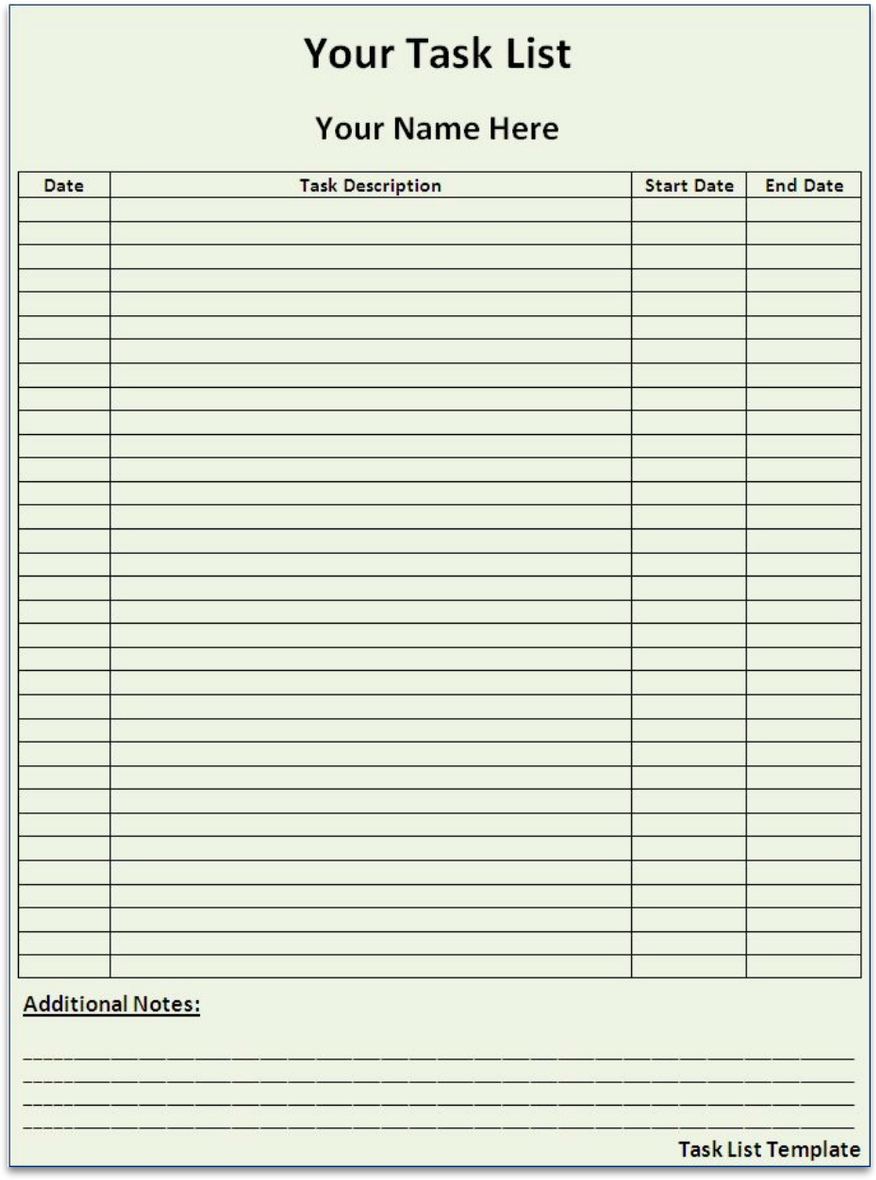
1. gareth@brutha: ~ (ssh)

```
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

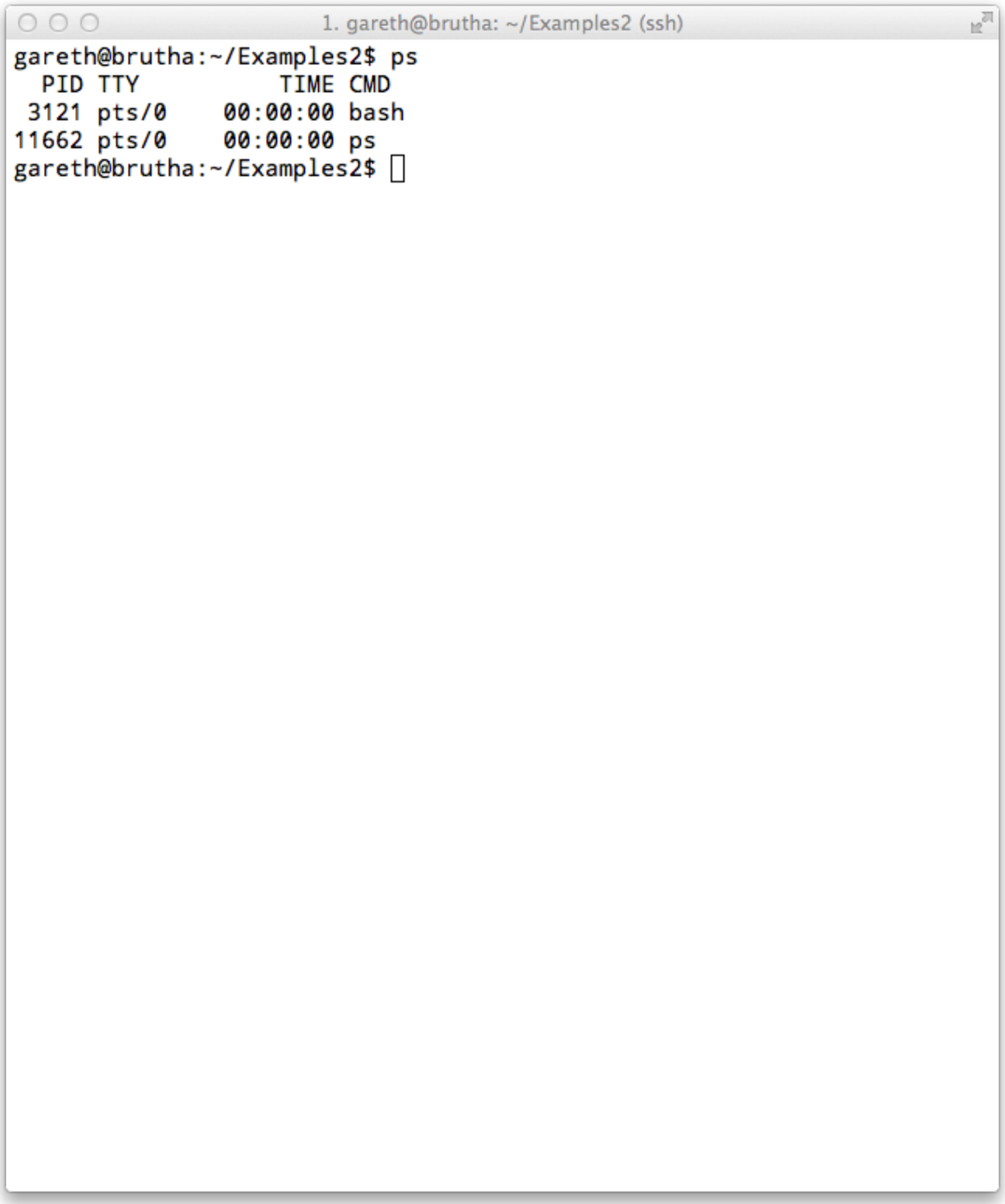
Common Tasks

- 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.



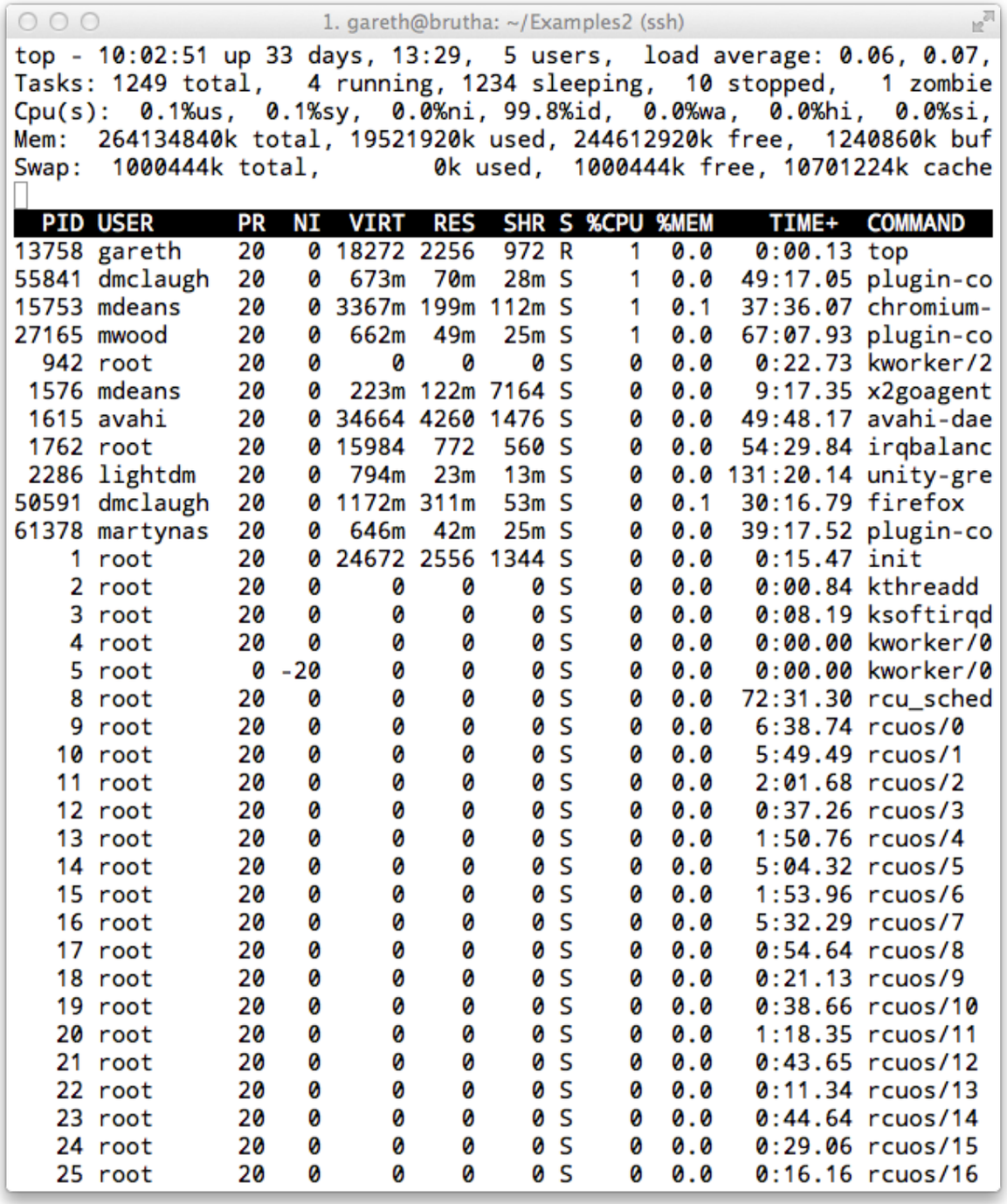
Seeing processes

- 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**



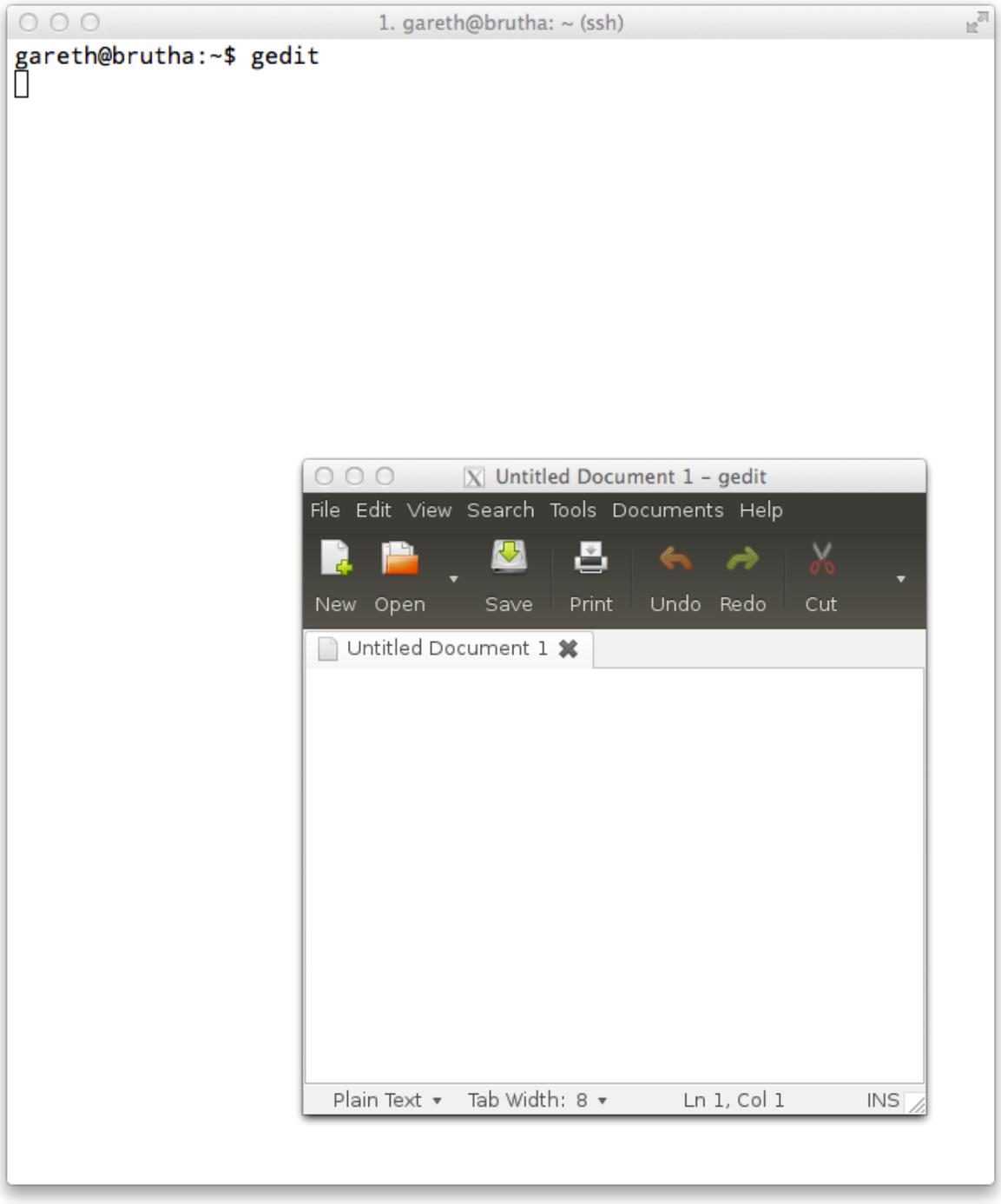
Other useful process tools

- 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



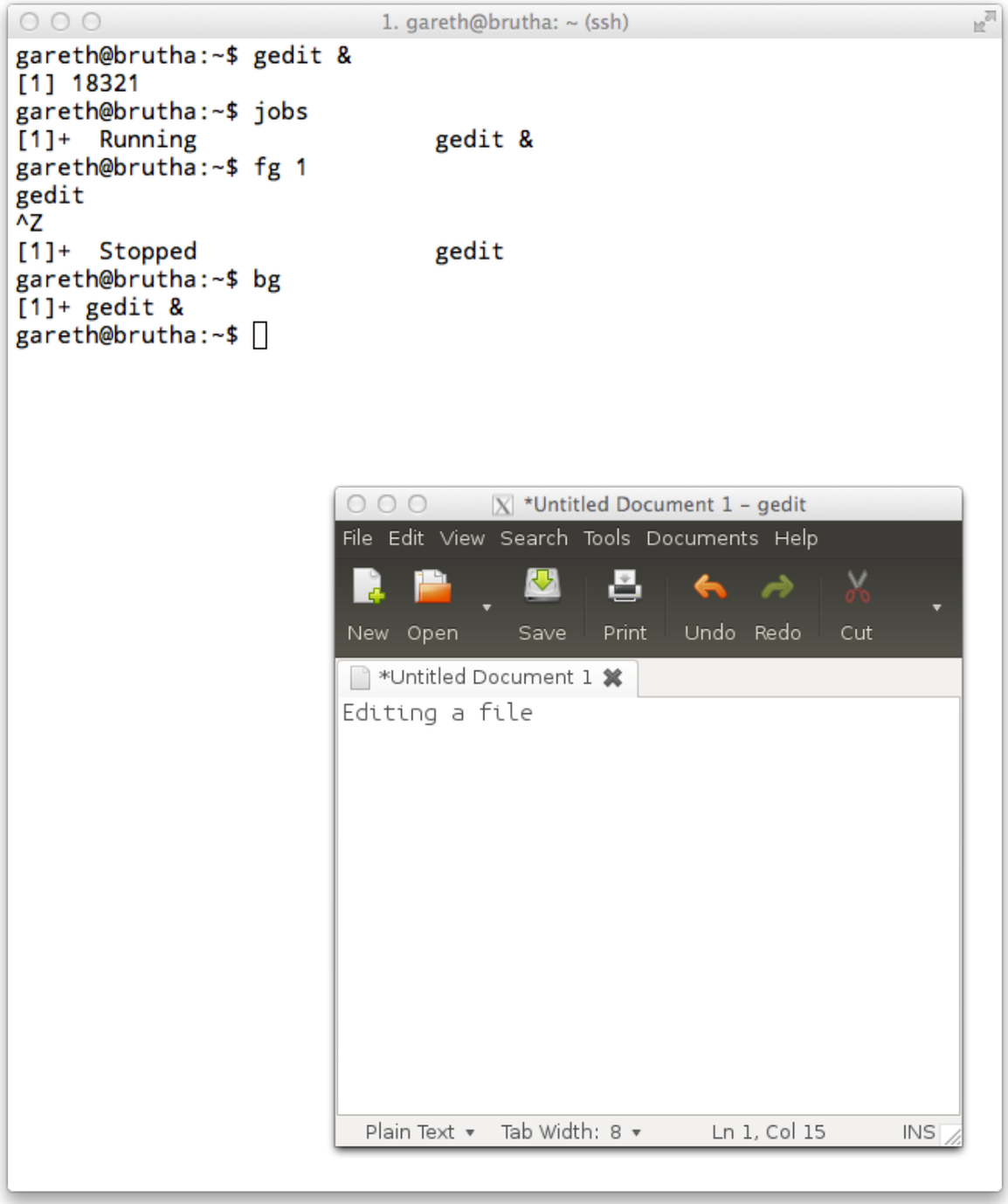
Foreground and Background

- 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.



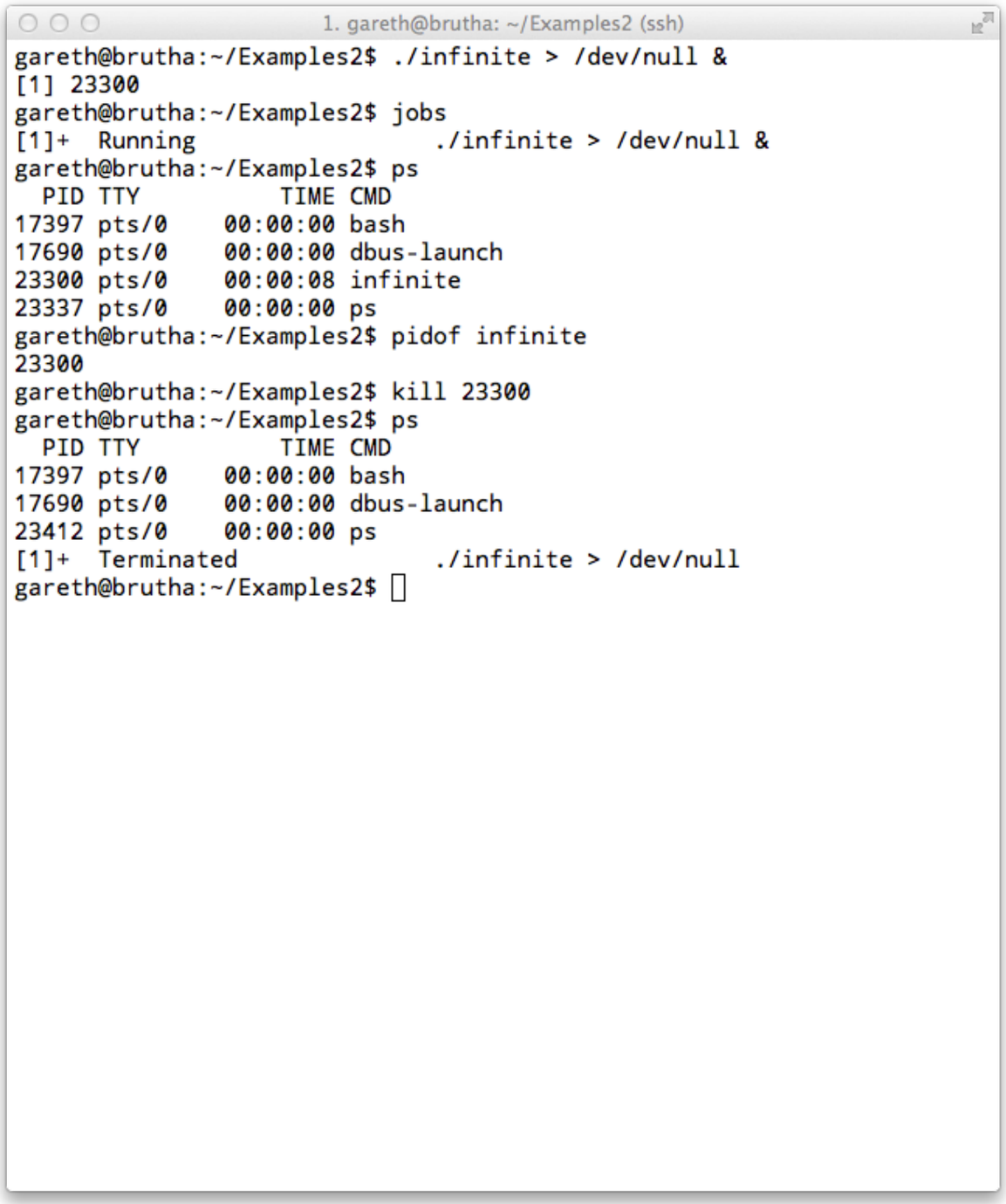
Job Control

- **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



Killing jobs

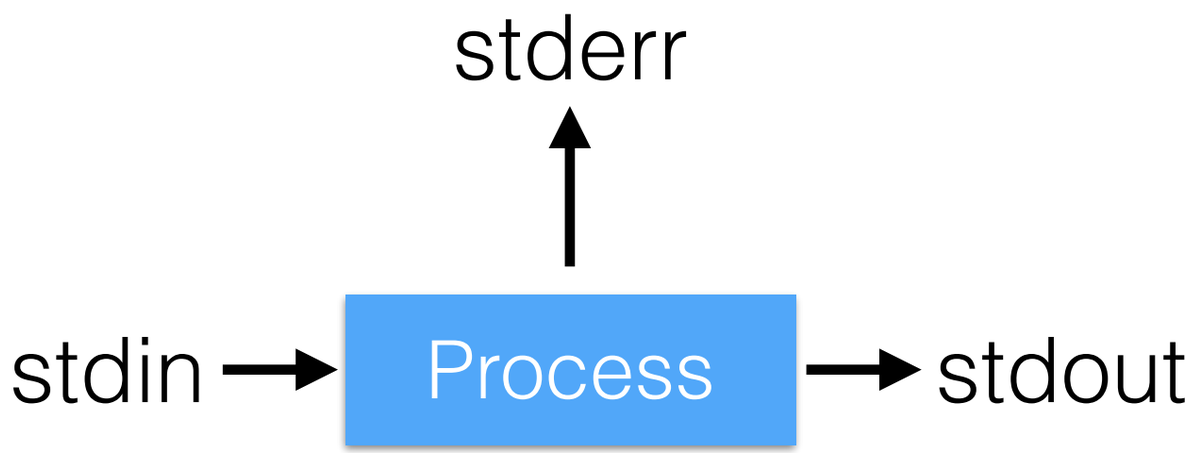
- Sometime we want to get rid of running processes.
- To remove a process from the system we use the **kill** command
  - **kill** <pid>
- You can find the pid either by running ps, or by using the pidof command
  - **pidof** <name>
- Sometimes we want to kill a process and all of it's children. to do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

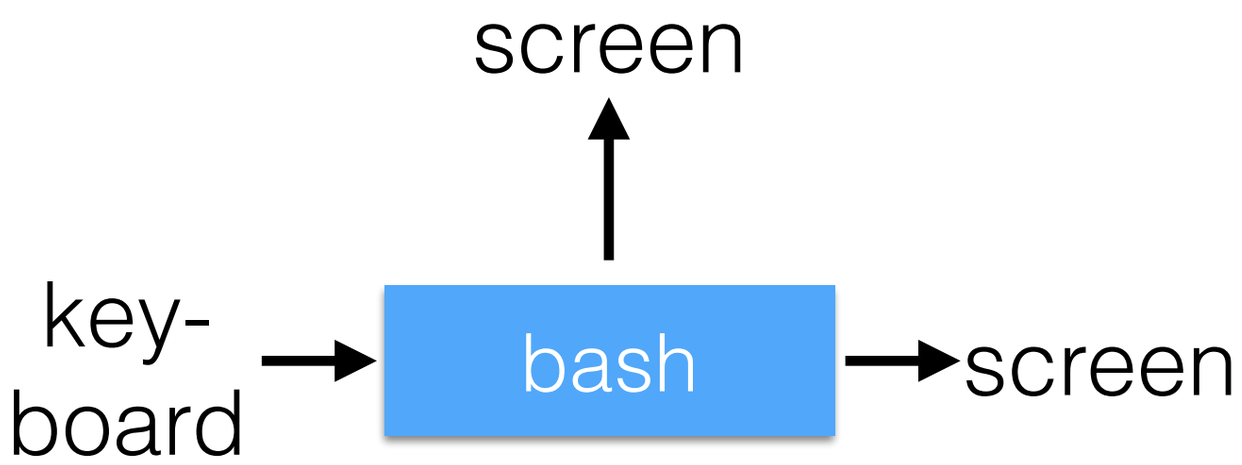


- Variables and the Shell
- Processes
- Working with Processes
- IO Streams
- Pipes and Redirections

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 see 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

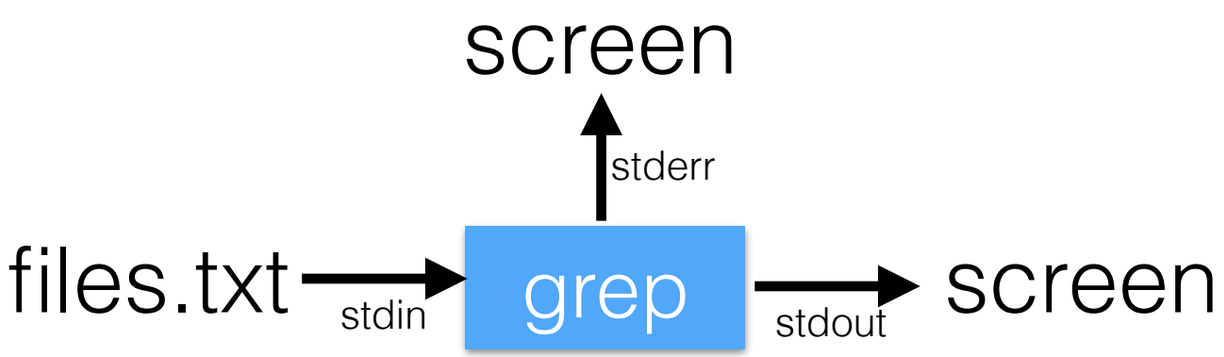
Redirection

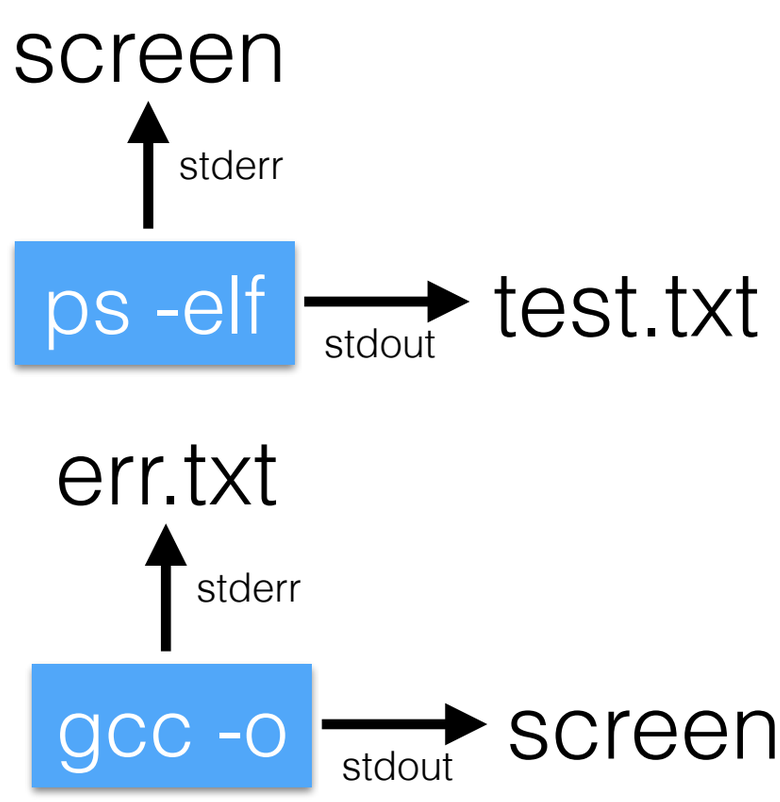
- The standard streams (**stdin**, **stderr** and **stderr**) can be redirected to point to places.

- 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.





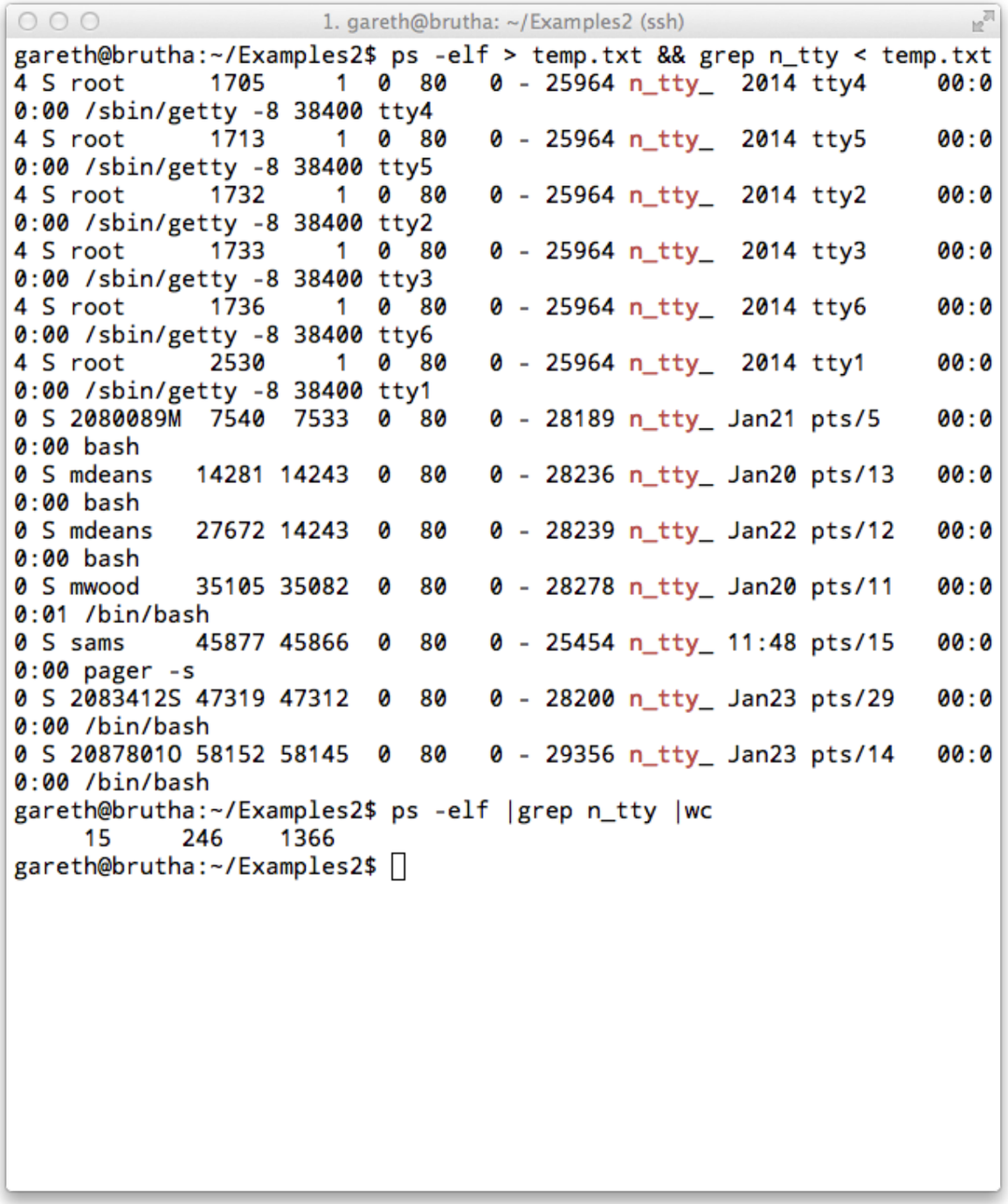
Redirection

- 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.

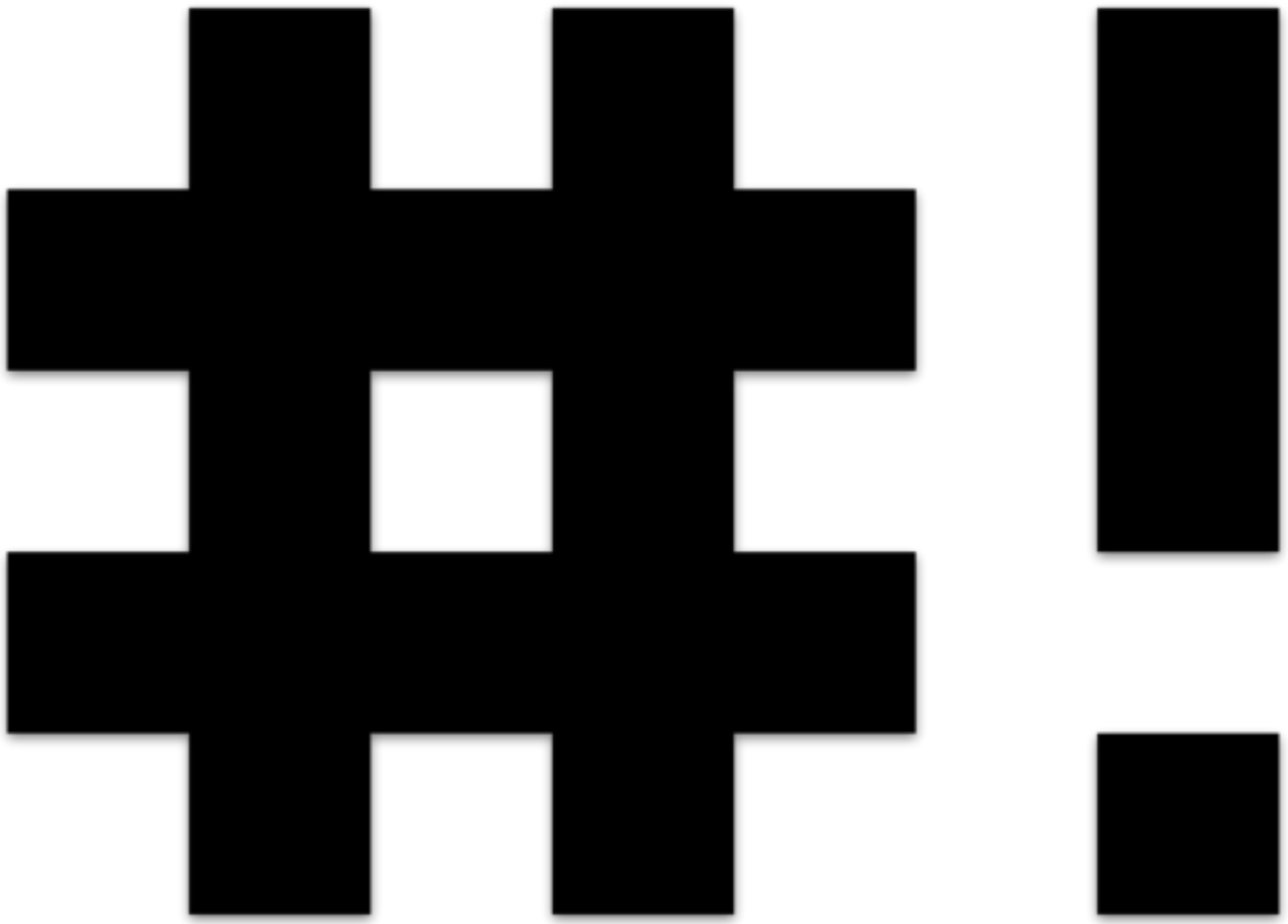
```
1. gareth@brutha: ~/Examples2 (ssh)
gareth@brutha:~/Examples2$ grep for < loop2.c
   for (i=0; i < length; i++){
    for (i=1; i <= length; i++) {
gareth@brutha:~/Examples2$ ps > process.txt
gareth@brutha:~/Examples2$ cat process.txt
  PID TTY          TIME CMD
17397 pts/0    00:00:00 bash
17690 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 l 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.



- 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

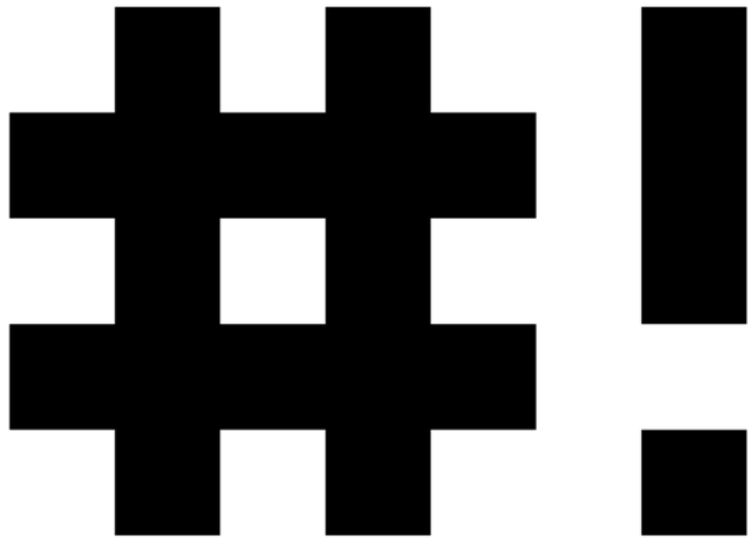
- Shell and script programming are useful tools in a physicist's armoury.
- Scripting allows the automation of easy, repetitive tasks.
- Example tasks include: checking webserver logs, backing up data to remote disk, creating directory structures or checking the workings of an automated experiment.
- 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.

2. ssh

```
#!/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" |g
rep 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 th
an threshold of ${sys_threshold}."
       text_message="${node:0:7} detected a high of ${temp} in 141 Sys
tems."
  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)

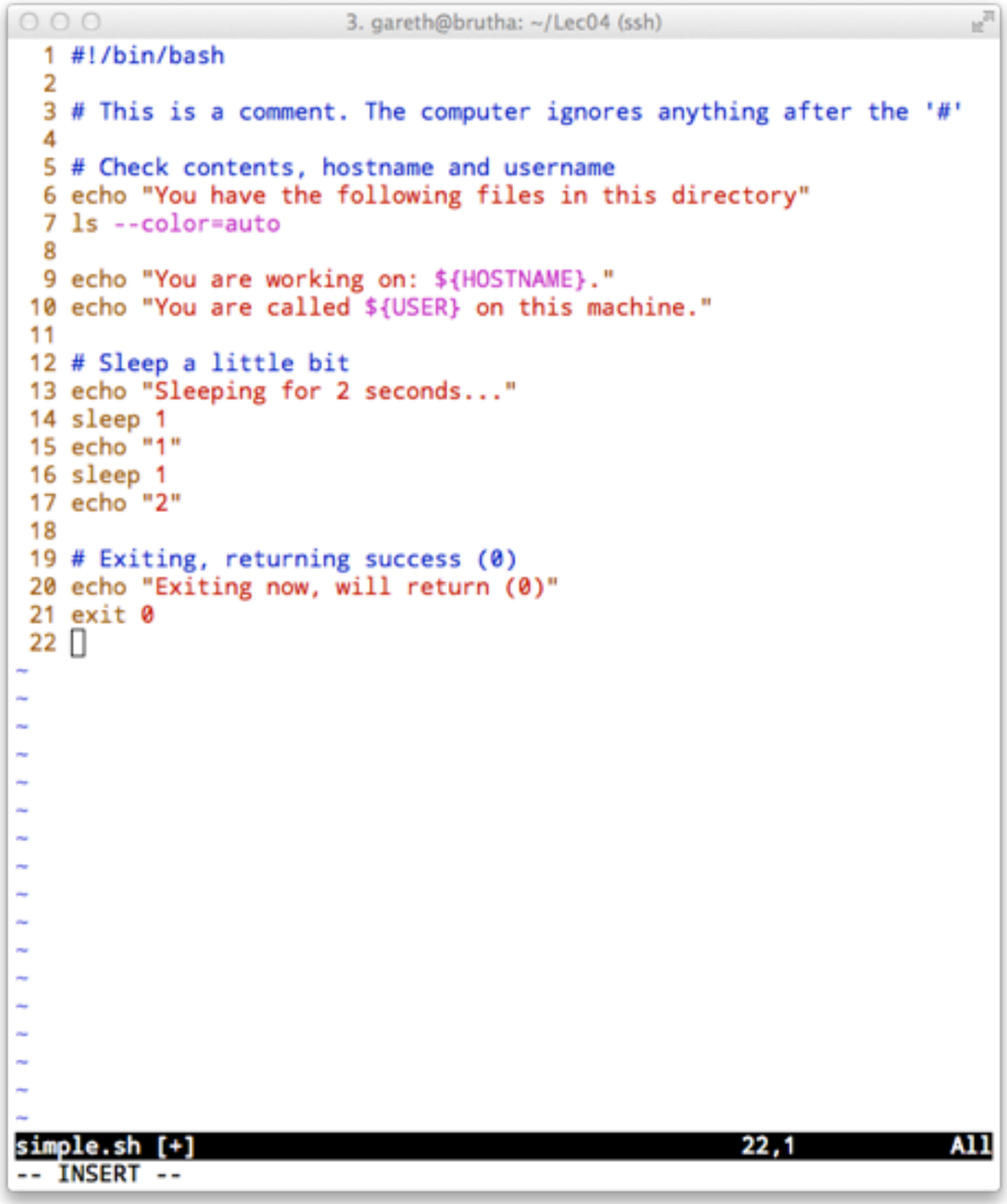
```
 1 #!/bin/bash
 2
 3 # This is a comment. The computer ignores anything after the '#'
 4
 5 # Check contents, hostname and username
 6 echo "You have the following files in this directory"
 7 ls --color=auto
 8
 9 echo "You are working on: ${HOSTNAME}."
10 echo "You are called ${USER} on this machine."
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$
```

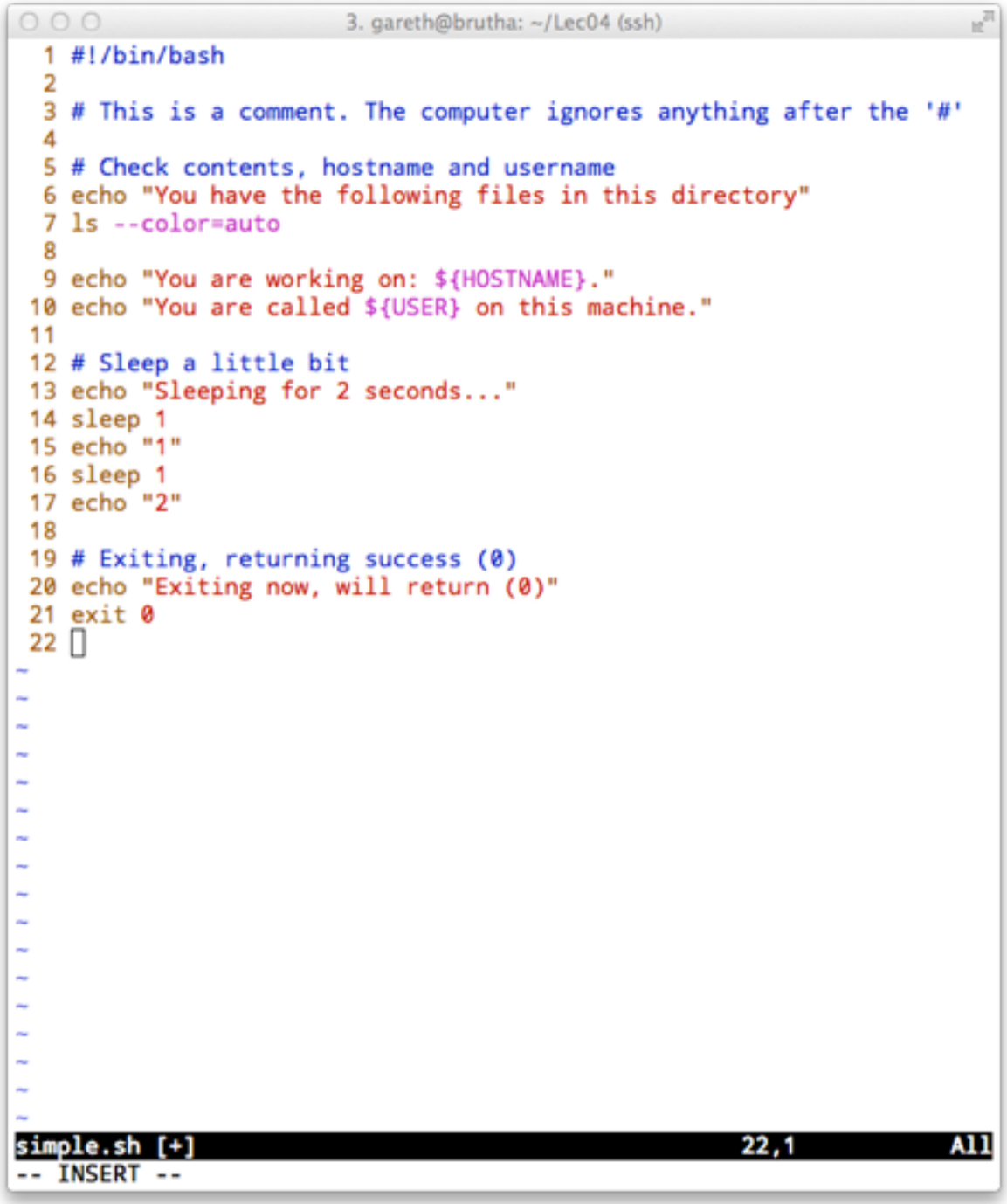
A Simple Script

- 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.



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.



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 you current session if it encounters an **exit** statement.

```
3. gareth@brutha: ~/Lec04 (ssh)
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.

3. gareth@brutha: ~/Lec04 (ssh)

```
#!/bin/bash

# Store the results of executing whoami
MYUSER=$(whoami)
echo "You are called ${MYUSER} on this machine."

# Another way to store the results
MYUSER2=`whoami`
echo "You are called ${MYUSER2} on this machine."

# Variables are not substitued in single quotes strings
MYUSER3=$(whoami)
echo 'You are called ${MYUSER3} on this machine.'

# Exiting, returning success (0)
echo "Exiting now, will return (0)"
exit 0
```

4. gareth@brutha: ~/Lec04 (ssh)

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

If, then, else

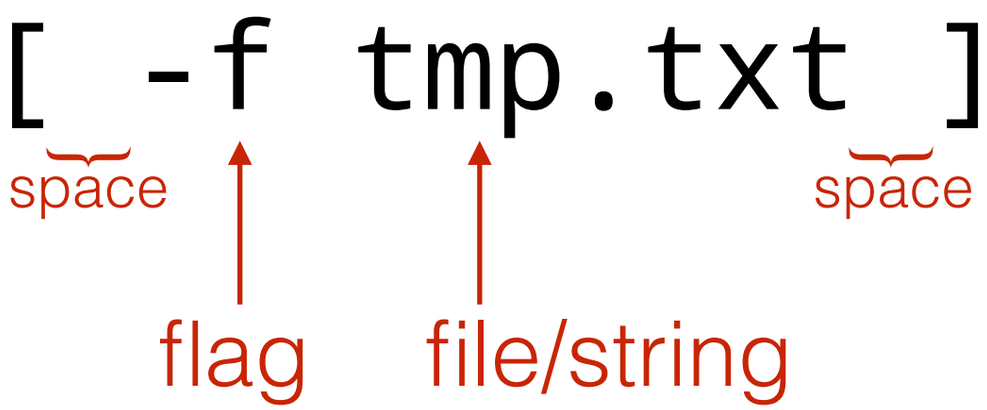
- **if** statements are similar to those found in C.
- They allow conditional execution of code, allowing decisions to be made about what to execute
- For instance **if** a file exists, **then** remove that file, **else** log an error.
- There are three general forms for an in **if** statement in Bash: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



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

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

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

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

3. gareth@brutha: ~/Lec04 (ssh)

```
#!/bin/bash

# Get the user running this script
MYID=$(whoami)

# Create an empty file
touch temp_file

# Test to see if the file exits
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
```

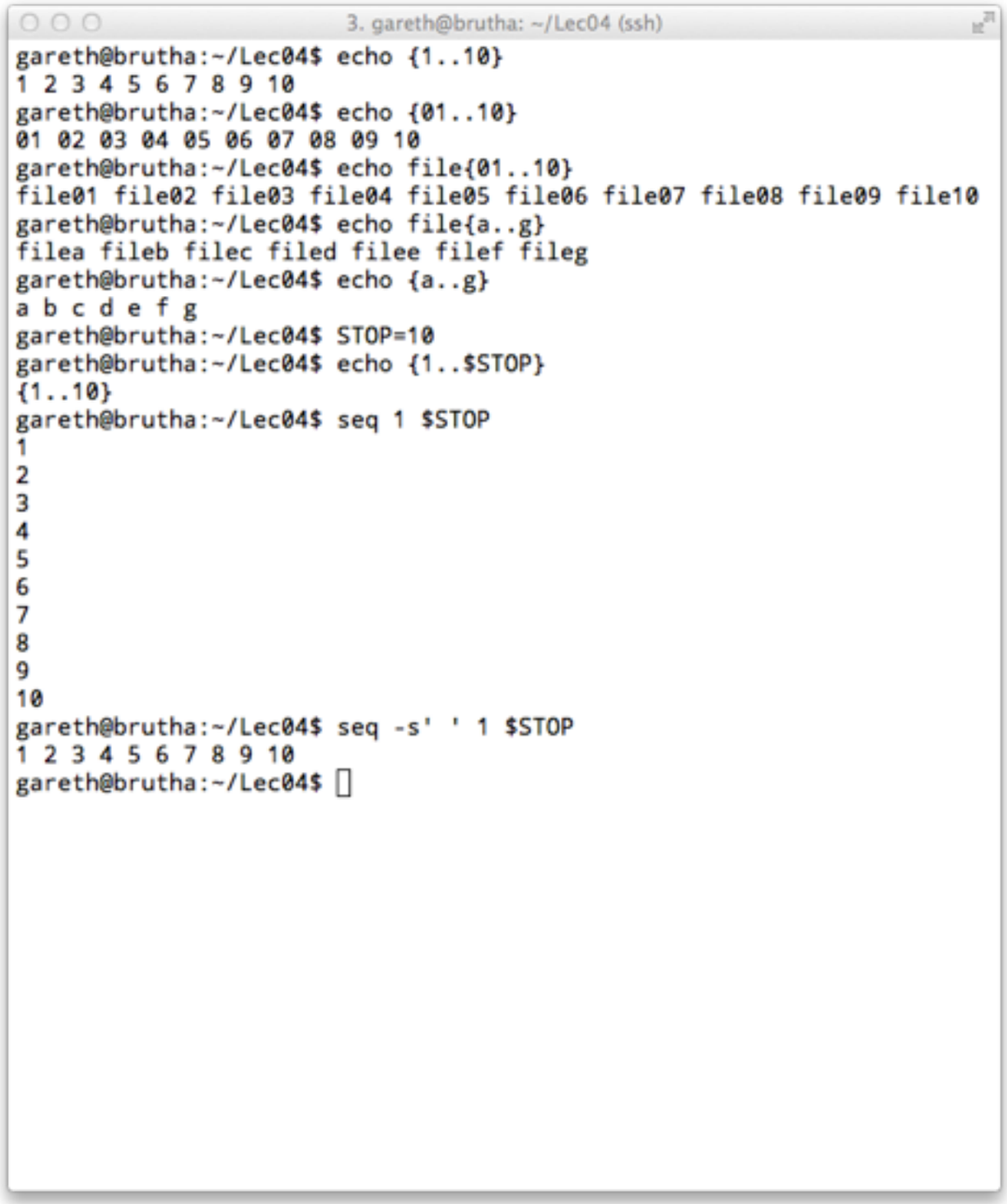
4. gareth@brutha: ~/Lec04 (ssh)

```
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

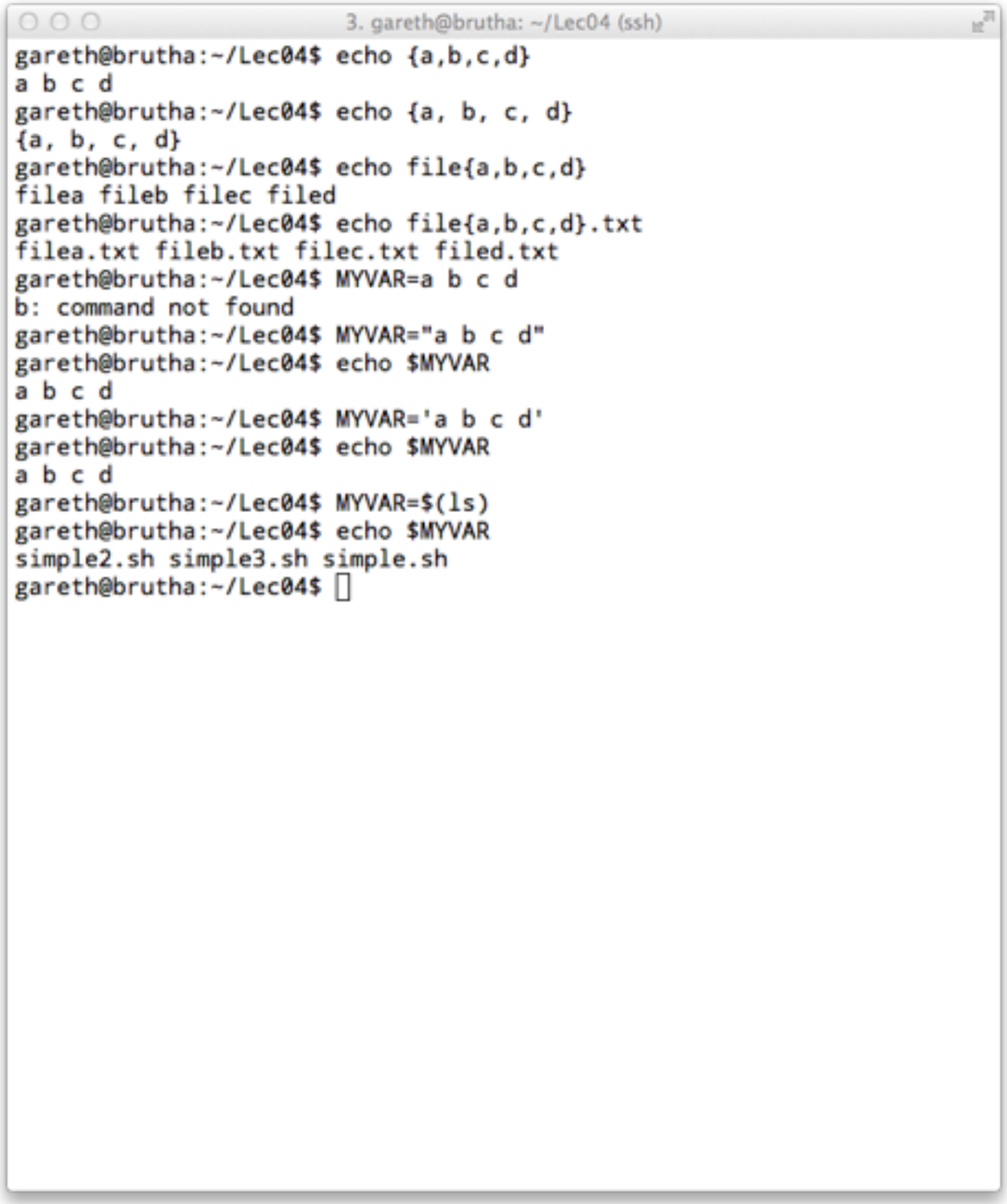
Ranges

- 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.



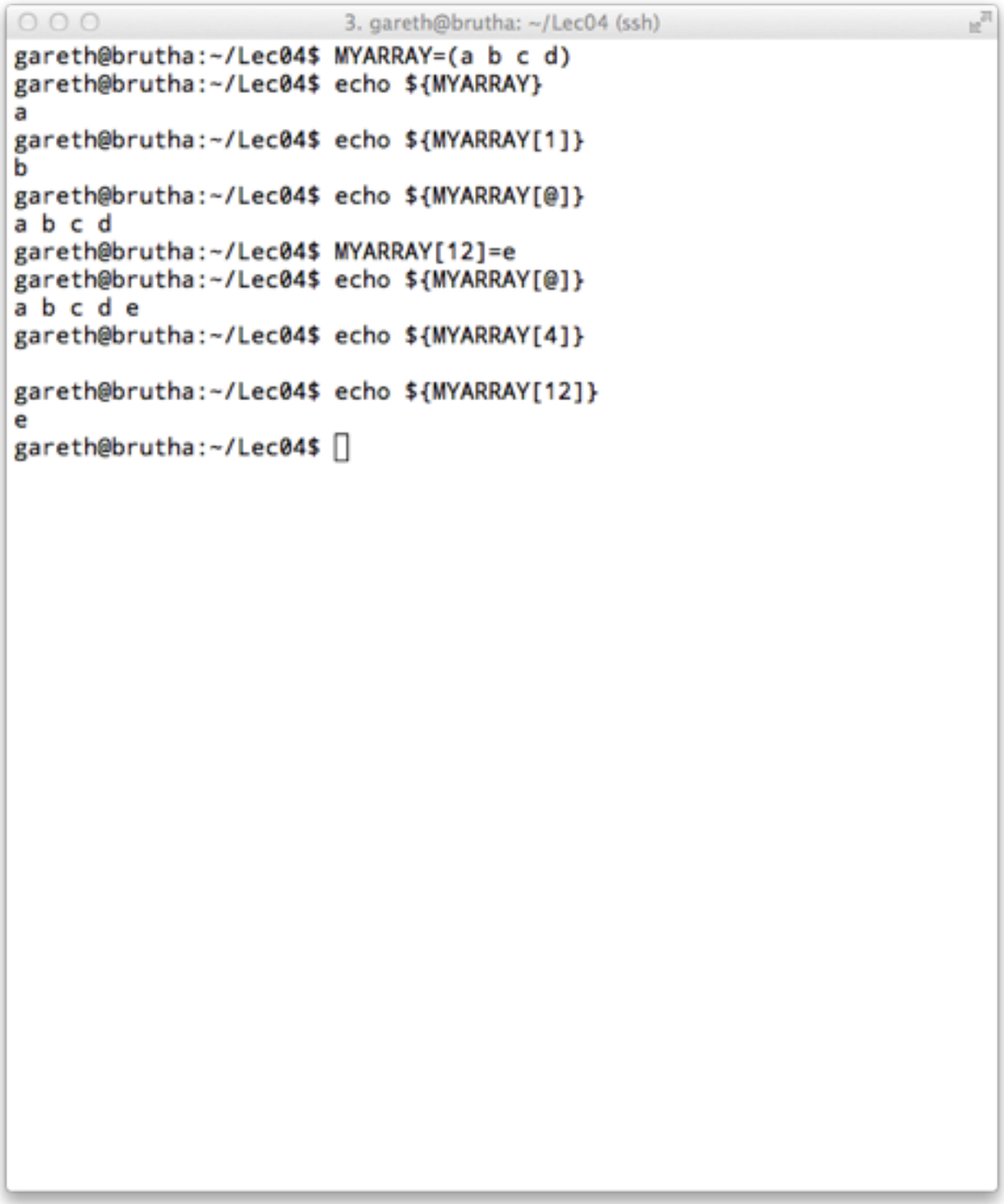
Lists

- 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”**



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**



- 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 an 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)

```
#!/bin/bash

# A C style for loop
for (( c=1; c<=3; c++ ))
do
        echo "I am a C style iteration ${c}!"
done

# A Bash style range for loop
for c in {a..d}
do
        echo "I am Range style iteration ${c}!"
done

# A more useful for loop, return the type of each file in a dir.
# Instead of * we could use `ls` or $(ls)
for FILE in *
do
        file -i ${FILE}
done

# A more complicted example, lets loop over the files
# check their types and if it is a shellscript write a message
for FILE in *
do
        TYPE=`file -i ${FILE} |awk '{print $2}'`
        if [ ${TYPE} == "text/x-shellscript;" ]
        then
                echo "${FILE} is a shellscript"
        fi
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$
```

All

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 exits at which point we **break** out of the loop and exit.

3. gareth@brutha: ~/Lec04 (ssh)

```
#!/bin/bash

SECONDS=''

# Wait for seconds to reach 12s
while [ ! ${SECONDS} -eq 12 ]
do
        SECONDS=`date +%S`
        echo ${SECONDS}
        sleep 1
done
echo "Done waiting, its $(date)"

# Wait until a lock file exits then exit
while true
do
        if [ -e lock ]; then
                break
        fi
done

echo "Exiting..."
exit 0
```

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).

3. gareth@brutha: ~/Lec04 (ssh)

```
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
```

4. gareth@brutha: ~/Lec04 (ssh)

```
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$
```

All

- 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 by 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. gareth@brutha: ~/Lec05 (ssh)

```
#!/bin/bash

# read can be used to get input from the user and store
# it in a variable
echo "Type a word: "
read WORD

# test to see if we catually got a word.
if [ -z ${WORD} ]; then
        echo "No really, you need to type in a word!"
        exit 1
else
        echo "Your word was - ${WORD}."
fi

# specifying -s means any characters typed won't be echoed
# to the screen.
echo "Type a secret word!!!!: "
read -s SECRET
echo "Your secret was - ${SECRET}."


# If we don't specify a variable read defaults to $REPLY
# -p allows us to passs a string as a prompt for input.
while true
do
        read -p "Do you want to exit? [y/n]: "
        if [ ${REPLY} == "y" ]; then
                break
        elif [ ${REPLY} == "n" ]; then
                echo "Continuing on..."
        else
                echo "${REPLY} not an understood answer."
        fi

done
```

All

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.

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
~
~
```

2. gareth@brutha: ~/Lec05 (ssh)

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

```
 1 #!/bin/bash
 2
 3 # Function definitions happen before the 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
~
```

2. gareth@brutha: ~/Lec05 (ssh)

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

All

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 funciont.
- **$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.

1. gareth@brutha: ~/Lec05 (ssh)

```
#!/bin/bash

# Arguments can be accessed using the same parameters as
# the Bash script itself, i.e. $1 represents the first
# argument passed to the function
function test_args {
        echo "I was called with $# arguments"
        echo "The arguments were $@"
        echo "The first argument was $1"
        echo
}

# Functions are called in the same way as commands
# Arguments are listed after the functin name
test_args a b c d

# We can access all the arguments by using a shift function
# shift effectively removes the first element and "shifts"
# the other elements left.
function shift_args {
        until [ -z $1 ]
        do
                echo "There are $# args - $@"
                echo $1
                shift
        done
}

shift_args a b c d
```

2. gareth@brutha: ~/Lec05 (ssh)

```
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$
```

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.

1. gareth@brutha: ~/Lec05 (ssh)

```
#!/bin/bash

# By default all variables declared in a Bash script have file
# scope, or in other words are declared a GLOBAL Variables
GLOBALVAR="This is a GLOBALVAR"

# We can specify a variable should be local to a function by
# using the local keyword
function variables {

        MYGLOBALVAR="This is also a GLOBALVAR"
        local MYVAR="This is a local VAR"

        echo "Function: ${GLOBALVAR}"
        echo "Function: ${MYGLOBALVAR}"
        echo "Function: ${MYVAR}"
}


echo "Main before: ${GLOBALVAR}"
echo "Main before: ${MYGLOBALVAR}"
echo "Main before: ${MYVAR}"

variables

echo "Main after: ${GLOBALVAR}"
echo "Main after: ${MYGLOBALVAR}"
echo "Main after: ${MYVAR}"

exit 0
```

2. gareth@brutha: ~/Lec05 (ssh)

```
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$
```

All

- 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.

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 that **set -x**

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

2. gareth@brutha: ~/Lec05 (ssh)

```
#!/bin/bash

# Debugging support:
export PS4='+$BASH_SOURCE:$LINENO:$FUNCNAME: '
set -x

# A single or multi-line comment telling the person
# reading the code what the script is supposed to do.

function usage {
        echo "This script prints out a given file line by line"
        echo "It requires a single argument whcih is the file"
        echo "to read."
        echo
        echo "simple6.sh <filename>"
}

# Test to make sure that an argument has been supplied.
if [ -z "$1" ]; then
        usage
        exit 1
fi

# Loop over the file and print each line
# Prepend each line with the filename.
while read LINE; do
        echo "$1: ${LINE}"
done < $1

exit 0
```

3. gareth@brutha: ~/Lec05 (ssh)

```
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**

2. gareth@brutha: ~/Lec05/C (ssh)

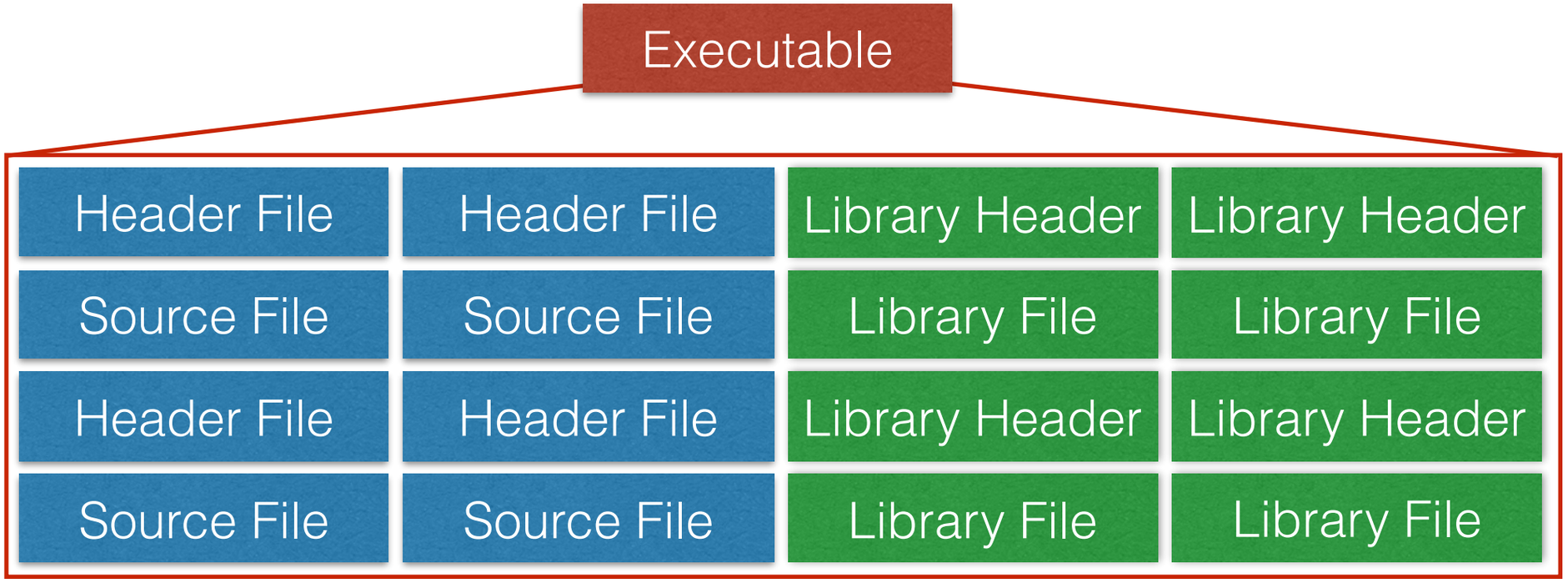
```c
#include <stdio.h>

/*
 * A small program that reads arguments from stdin,
 * loops over the arguments and prints them out one
 * by one.
 *
 */

/* Main */
int main(int argc, char *argv[]) {
    int i,sum,average;

/* Ensure we actually have some arguments */
    if (1 == argc) {
       printf("Arghhhhh\n");
       return 1;
    }

    /* Loop over each Argument and print to screen */
    for (i=1; i < argc; i++) {
        printf("Arg %d - %s\n",i,argv[i]);
    }
    return 0;
}
```

3. gareth@brutha: ~/Lec05/C (ssh)

```
gareth@brutha:~/Lec05/C$ gcc -o loop loop.c
gareth@brutha:~/Lec05/C$ ./loop
Arghhhhh
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).

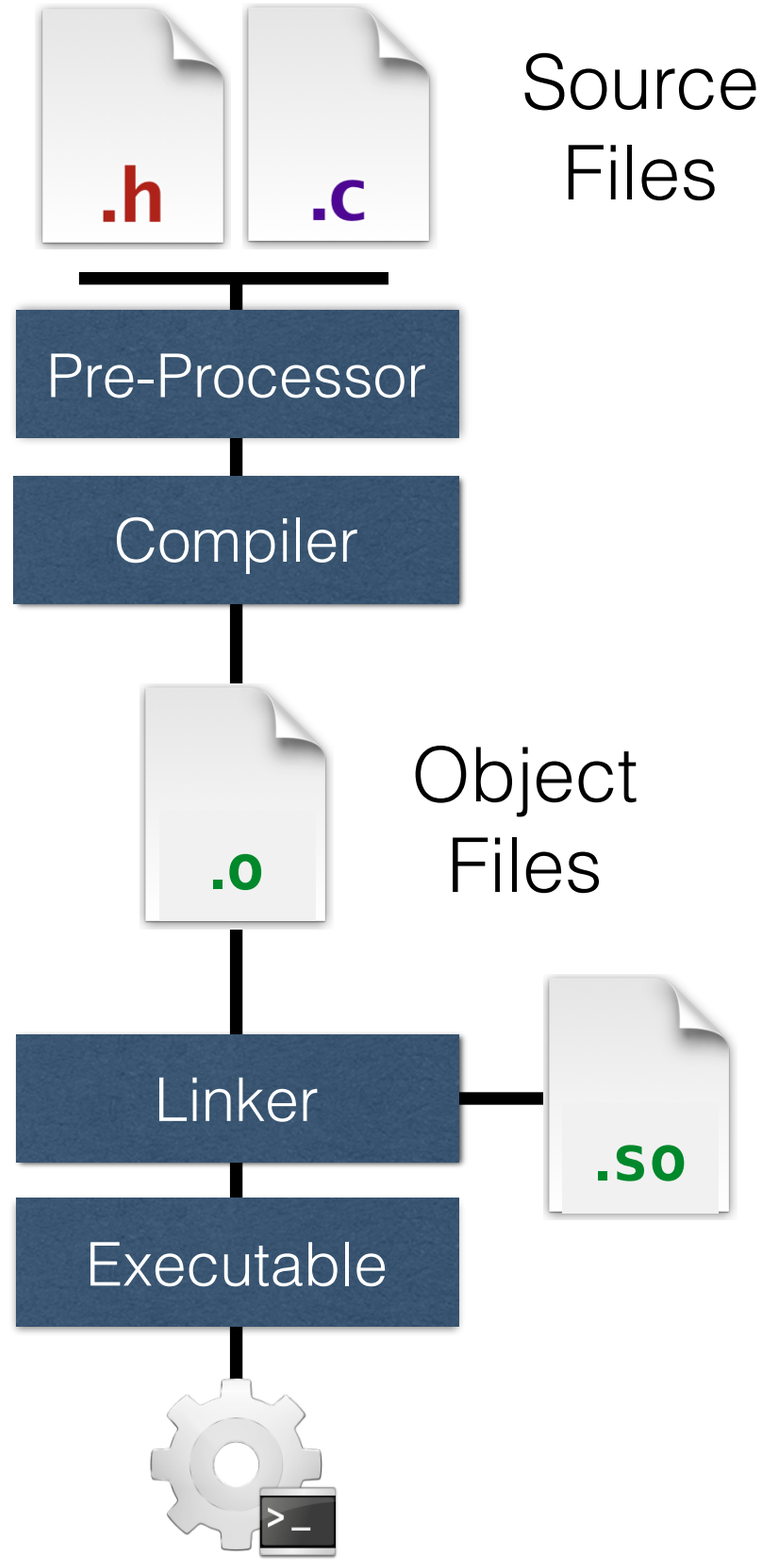
Compilation

- Compilation is actual three separate stages:

- **Pre-processor** - takes source and header files and combines the two, while expanding all #defines etc (see C lectures).

- **Compilation** - take 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** - take 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 by 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)

```
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         return 0;
 16 }
 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 you **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.

```
2. gareth@brutha: ~/Lec05/C/multi (ssh)
gareth@brutha:~/Lec05/C/multi$ clear
gareth@brutha:~/Lec05/C/multi$ ldd myprog
        linux-vdso.so.1 =>  (0x00007fff37800000)
        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$
```

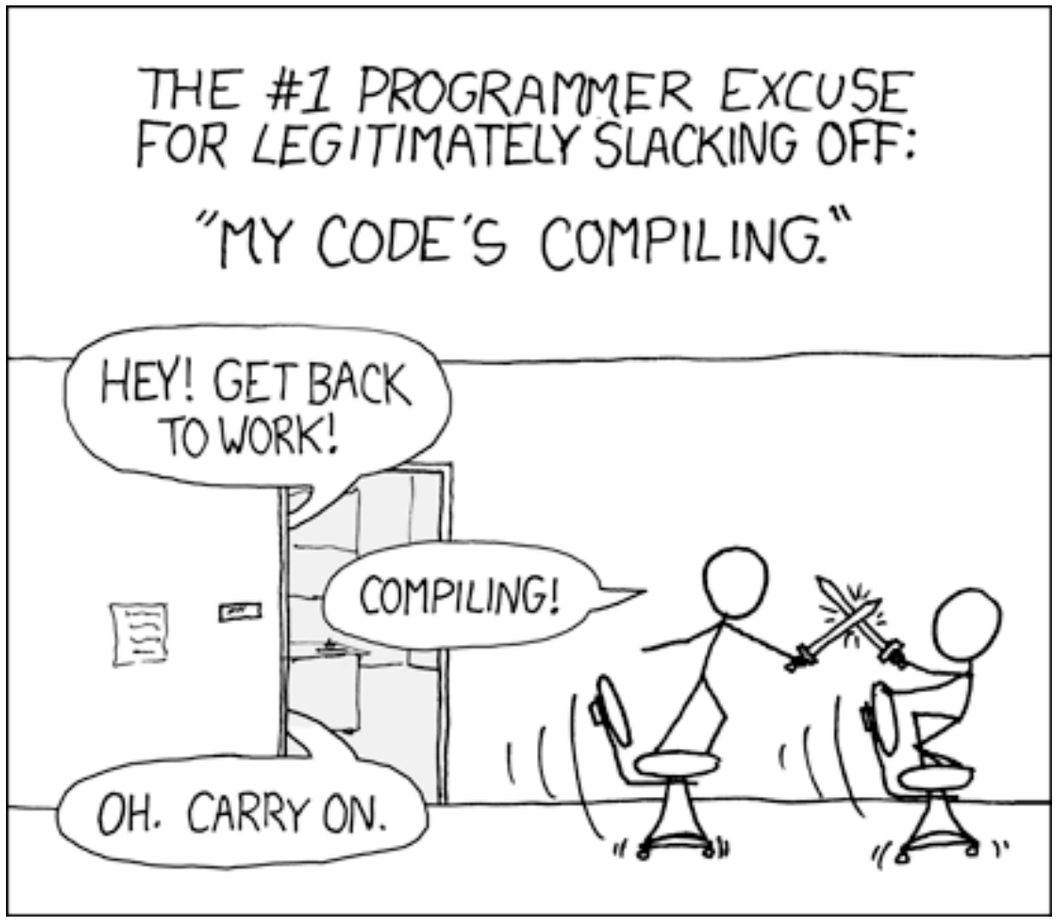
- Input (User & File)
- Functions
- Good Practice
- An Aside - Compilation

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.

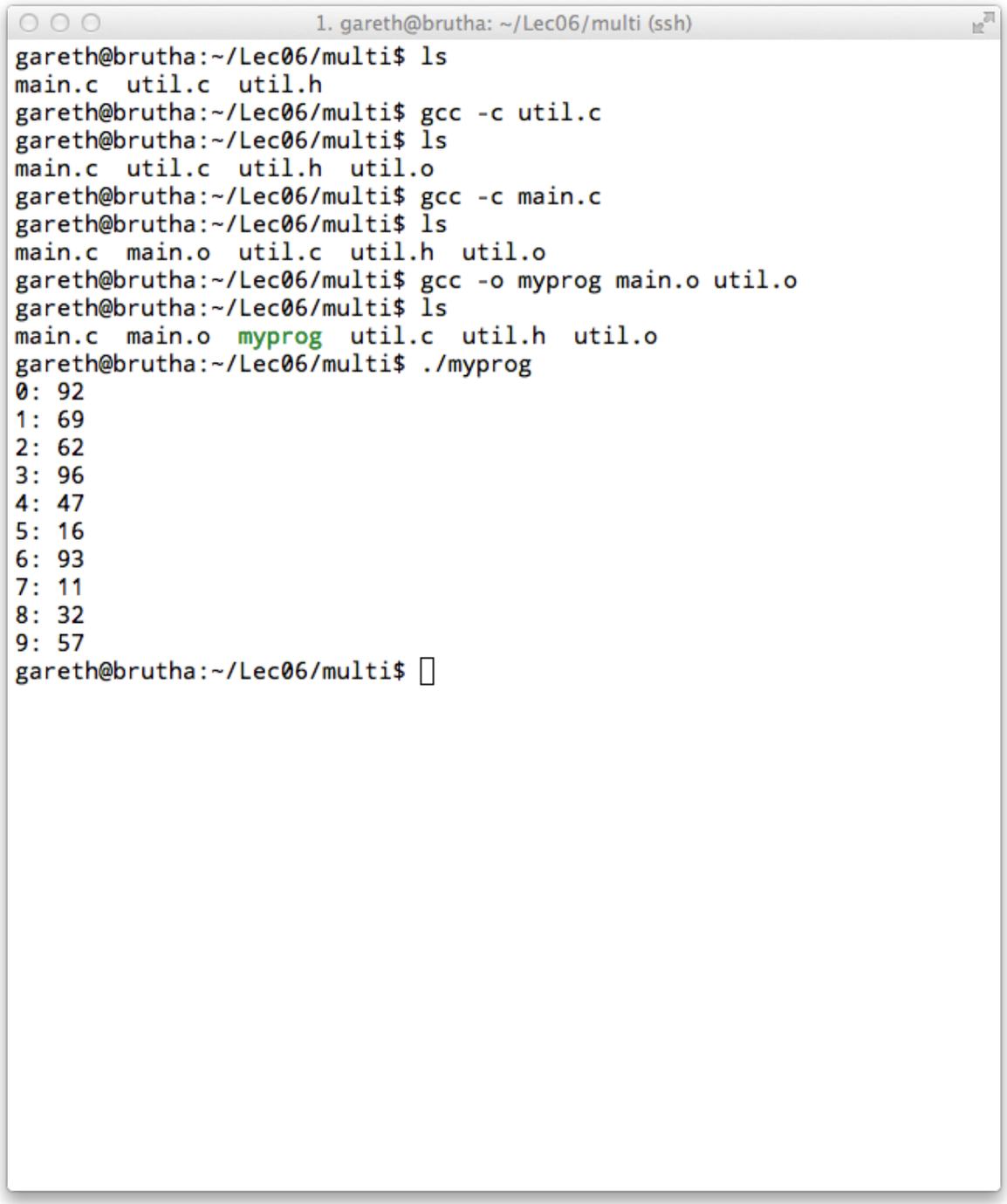
  - **util.c** - has a function that takes an array and fills it with a series of random numbers.

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                                          1,1           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         return 0;
 16 }
 17
~
main.c [+]                                      17,1          All
-- INSERT --
```

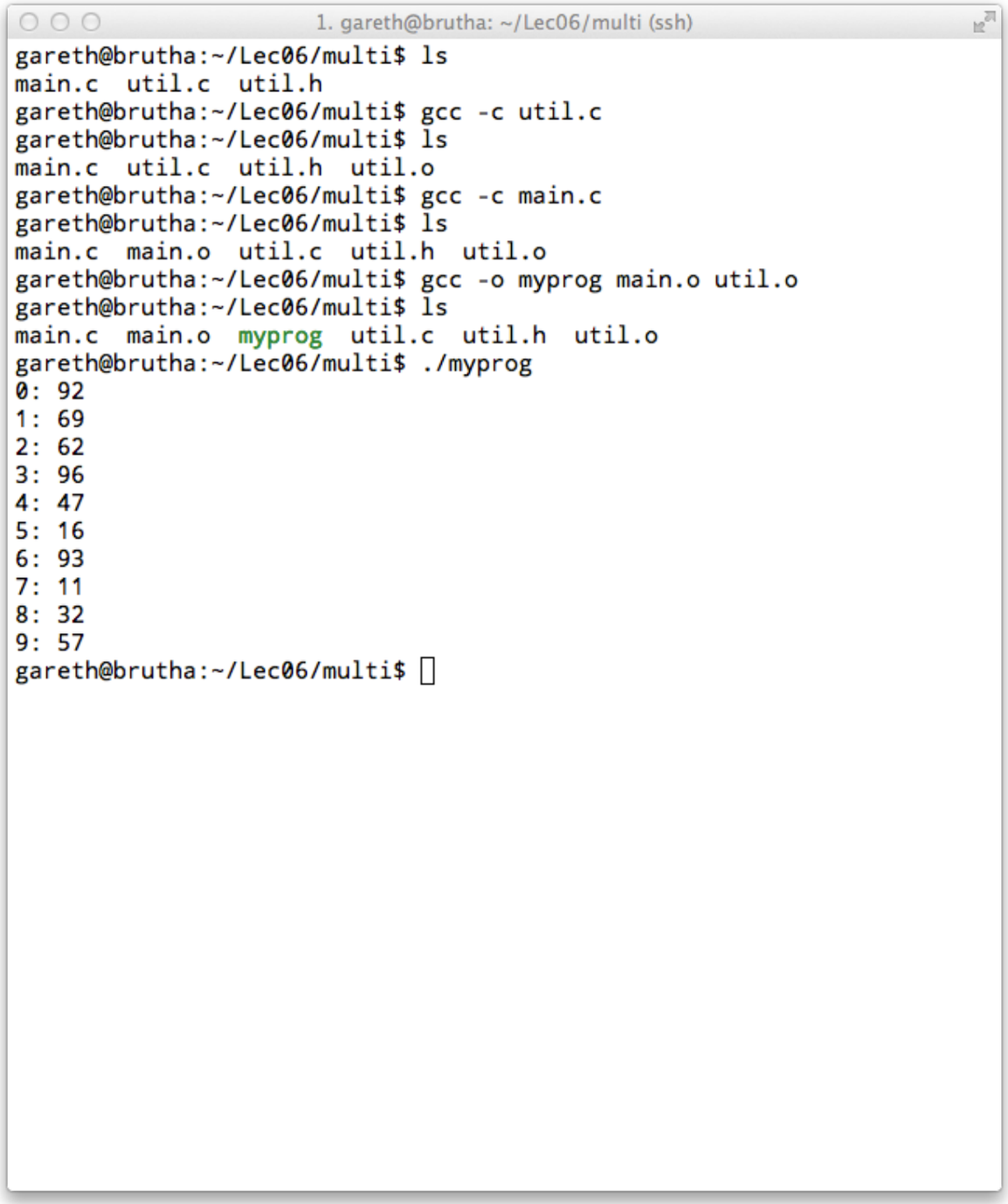
Compiling the Example

- 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.



Compiling the Example

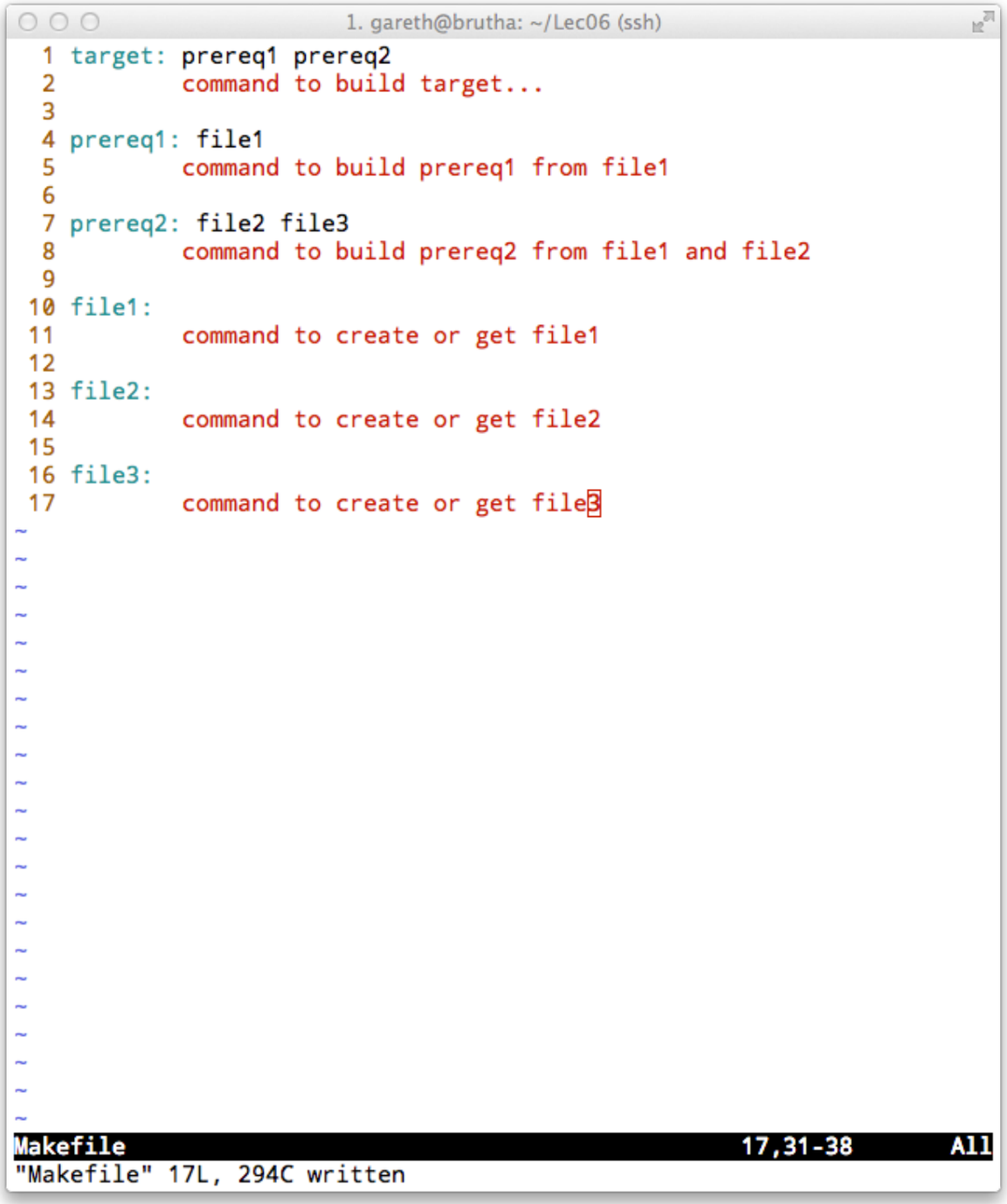
- The downside of incremental builds is the programmer is required to keep track of all of the files that change. When code bases grow to 1000's of files this can be a problem
- Importantly we need to ensure no stale object files are linked into the main program.
- Manually writing compile lines can 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.



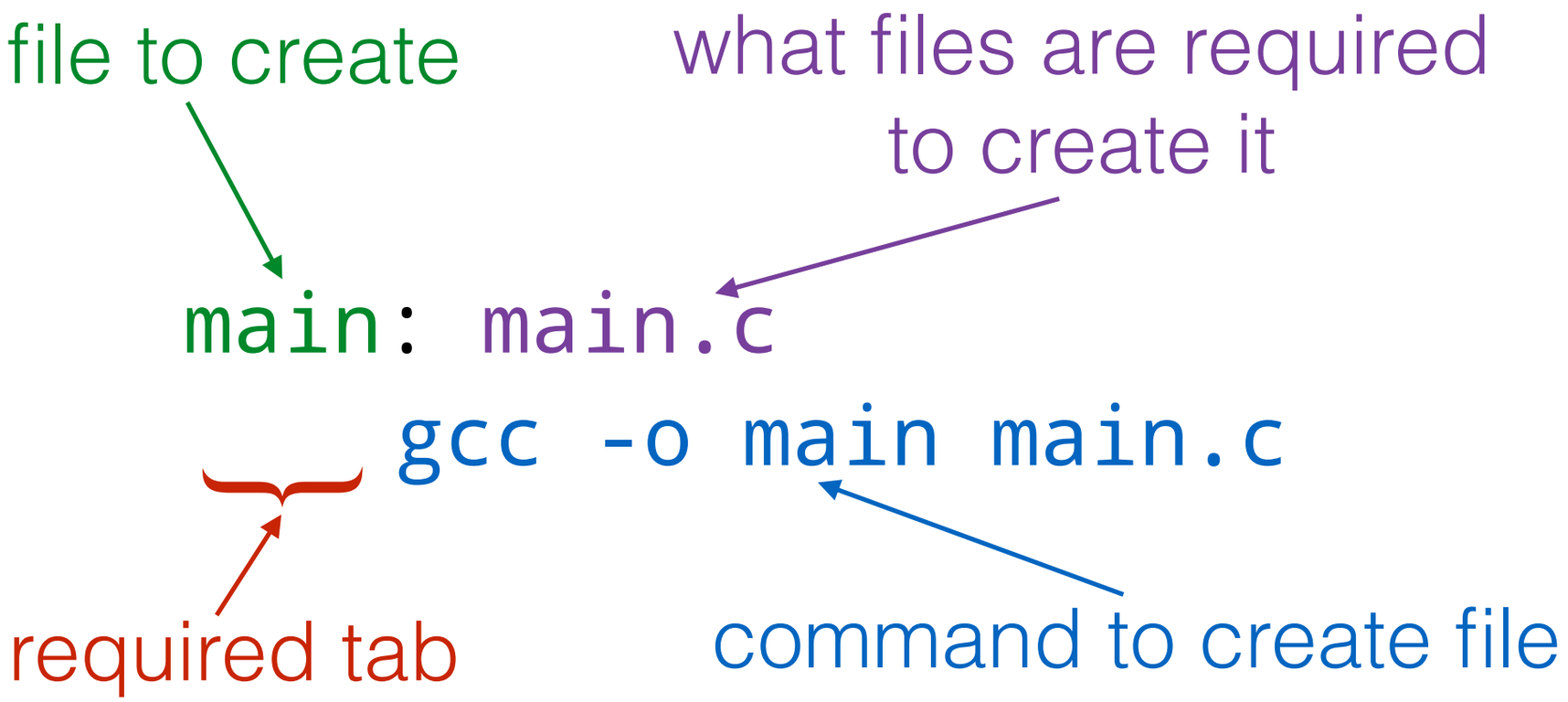
- Compilation
- Makefiles
- Revision Control
  - Git

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.



Make Rules

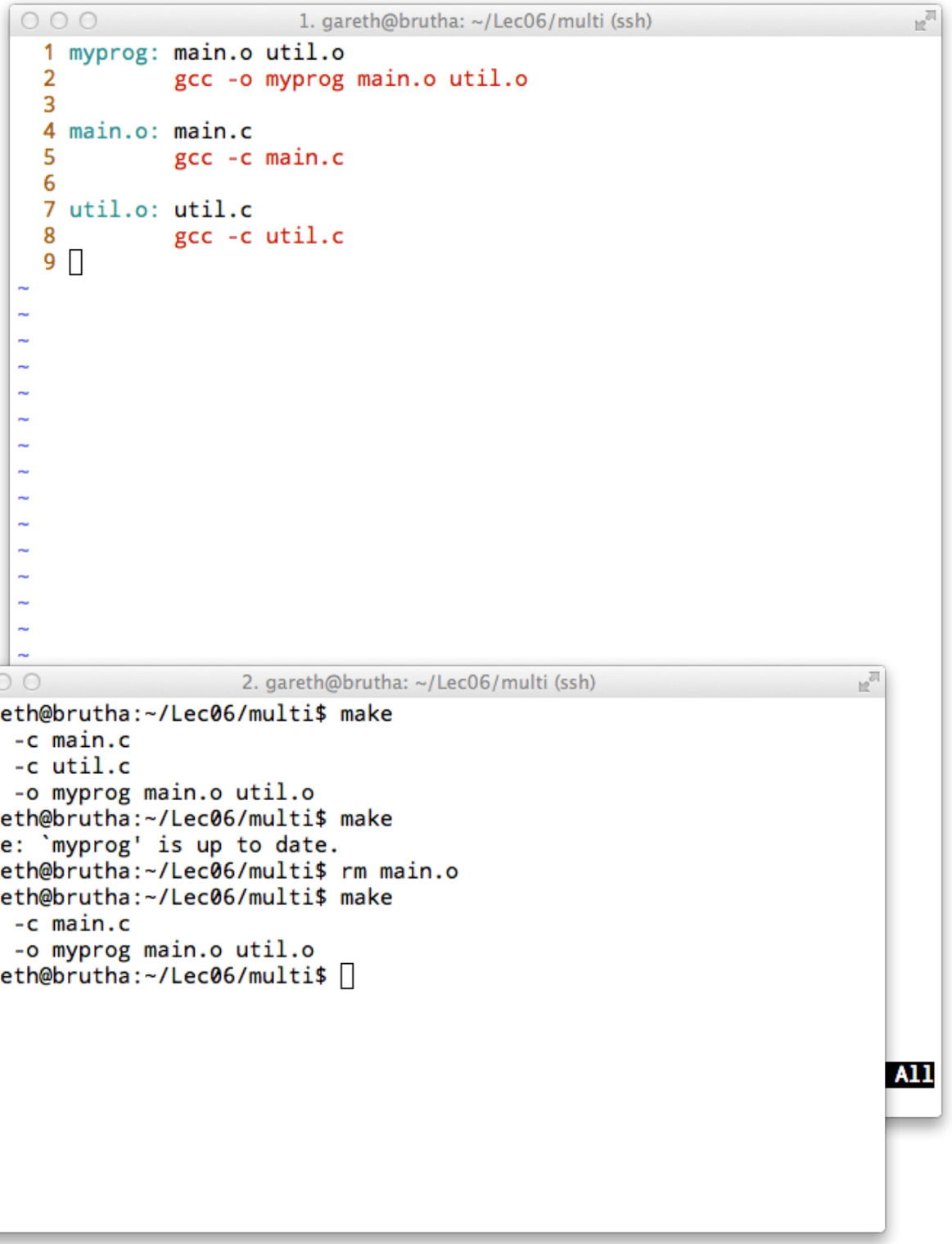


- Makefile rules require a **target**, it's **dependencies** and the **command** needed to produce the target from it's dependencies. In this instance `main` is produced from `main.c` by running `gcc`.
- Makefiles can be used for a number of things, not just compiling. For instance the following rule 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.



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**
- Referencing the variable again is similar to Bash:
  - **${VAR}**
  - **$(VAR)**
- 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**)

1. gareth@brutha: ~/Lec06/multi/vars (ssh)

```
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
~
~
~
~
~
```

2. gareth@brutha: ~/Lec06/multi/vars (ssh)

```
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.

1. gareth@brutha: ~/Lec06/multi/vars (ssh)

```
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)
~
```

2. gareth@brutha: ~/Lec06/multi/vars (ssh)

```
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$
```

All

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...

1. gareth@brutha: ~/Lec06/multi/autovars (ssh)

```
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)
~
```

2. gareth@brutha: ~/Lec06/multi/autovars (ssh)

```
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.

1. gareth@brutha: ~/Lec06/multi/advanced (ssh)

```
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)
```

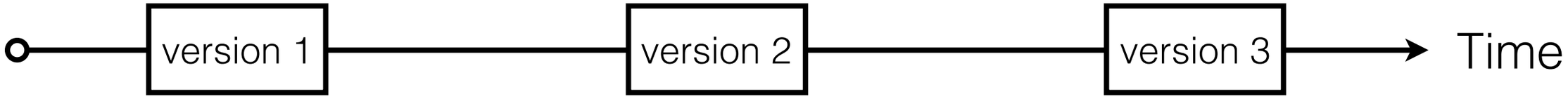
2. gareth@brutha: ~/Lec06/multi/advanced (ssh)

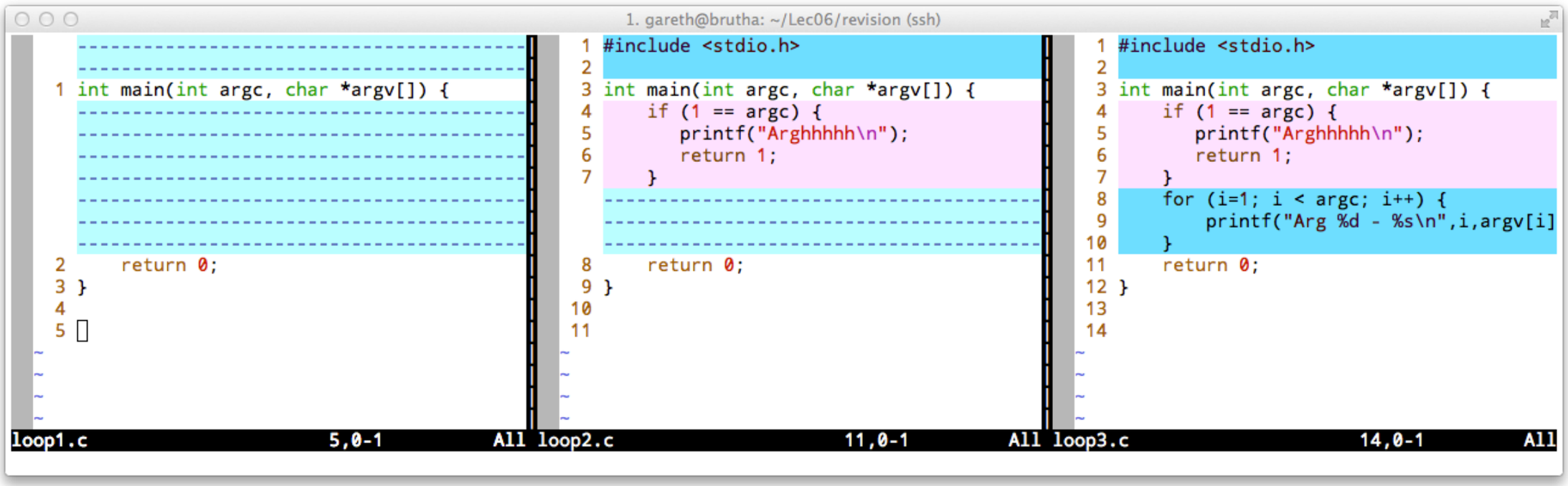
```
gareth@brutha:~/Lec06/multi/advanced$ make debug
gcc -Wall -g -c main.c
gcc -Wall -g -c util.c
gcc -lm -o myprog main.o util.o
gareth@brutha:~/Lec06/multi/advanced$ make clean
rm *.o
rm myprog
gareth@brutha:~/Lec06/multi/advanced$ make opt
gcc -Wall -O2 -c main.c
gcc -Wall -O2 -c util.c
gcc -lm -o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

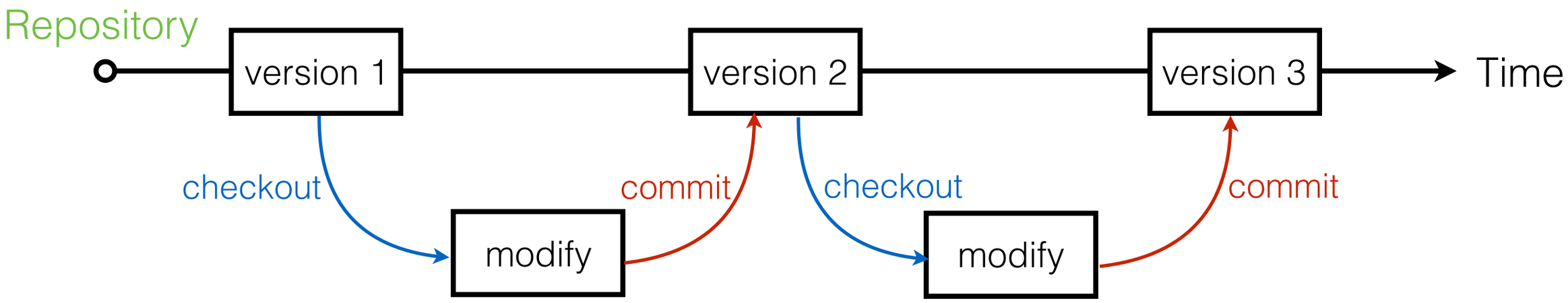
What is Revision Control?

- 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 know as Version control, Change control and Source Code management) is a method for managing the changes to a file, and maintaing a history of the changes that have taken place.

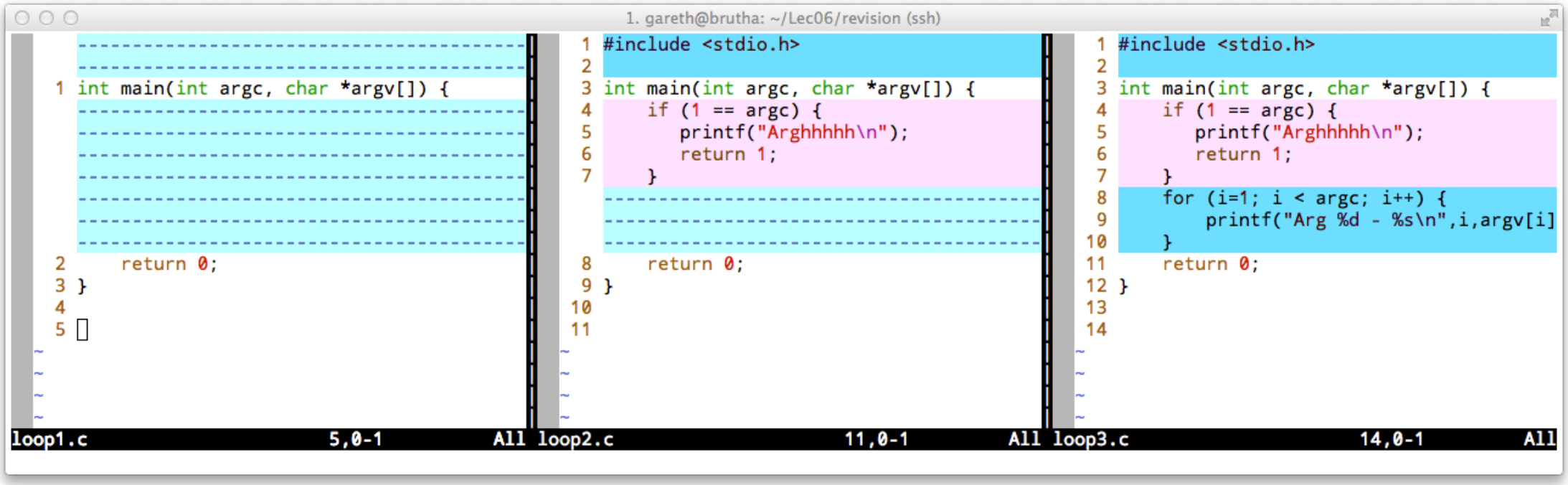




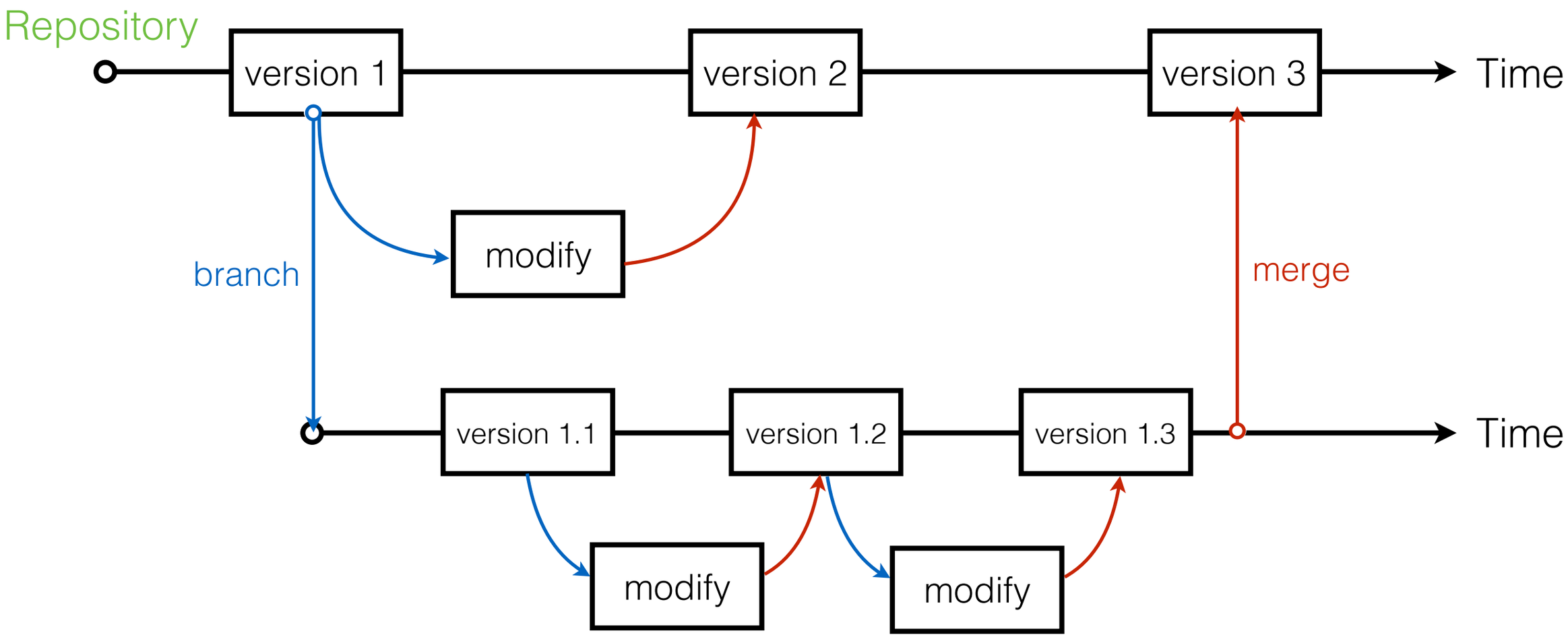
The Basics of Revision Control



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)



Branch & Merge

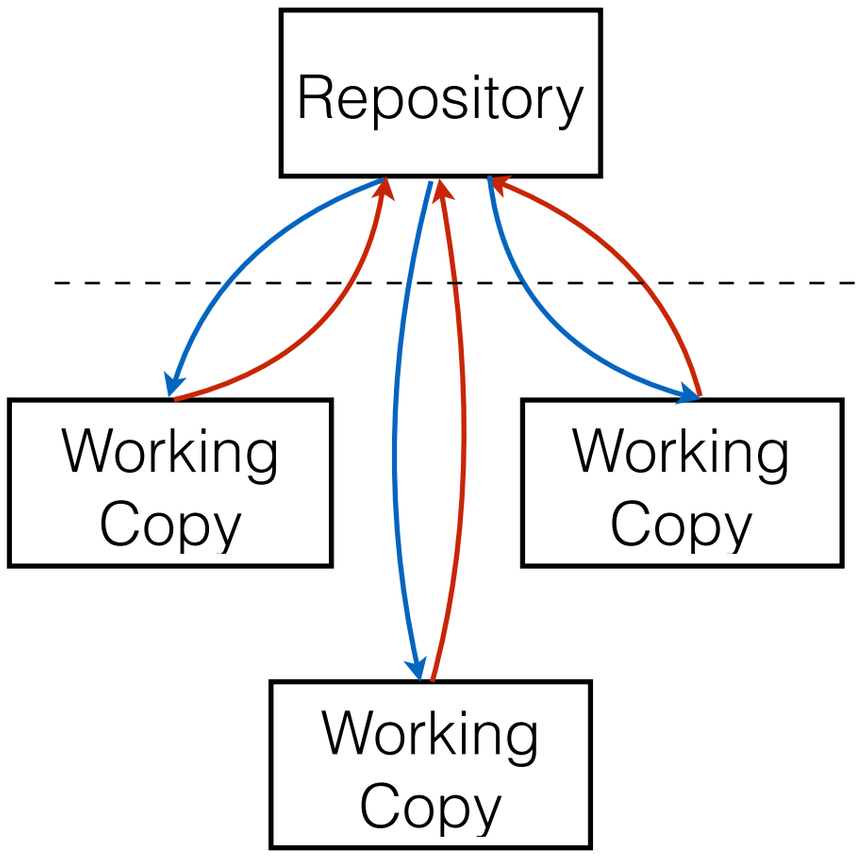


Another key concept is branching and merging.

Branch: Create a complete copy of the code with it's own revision history.

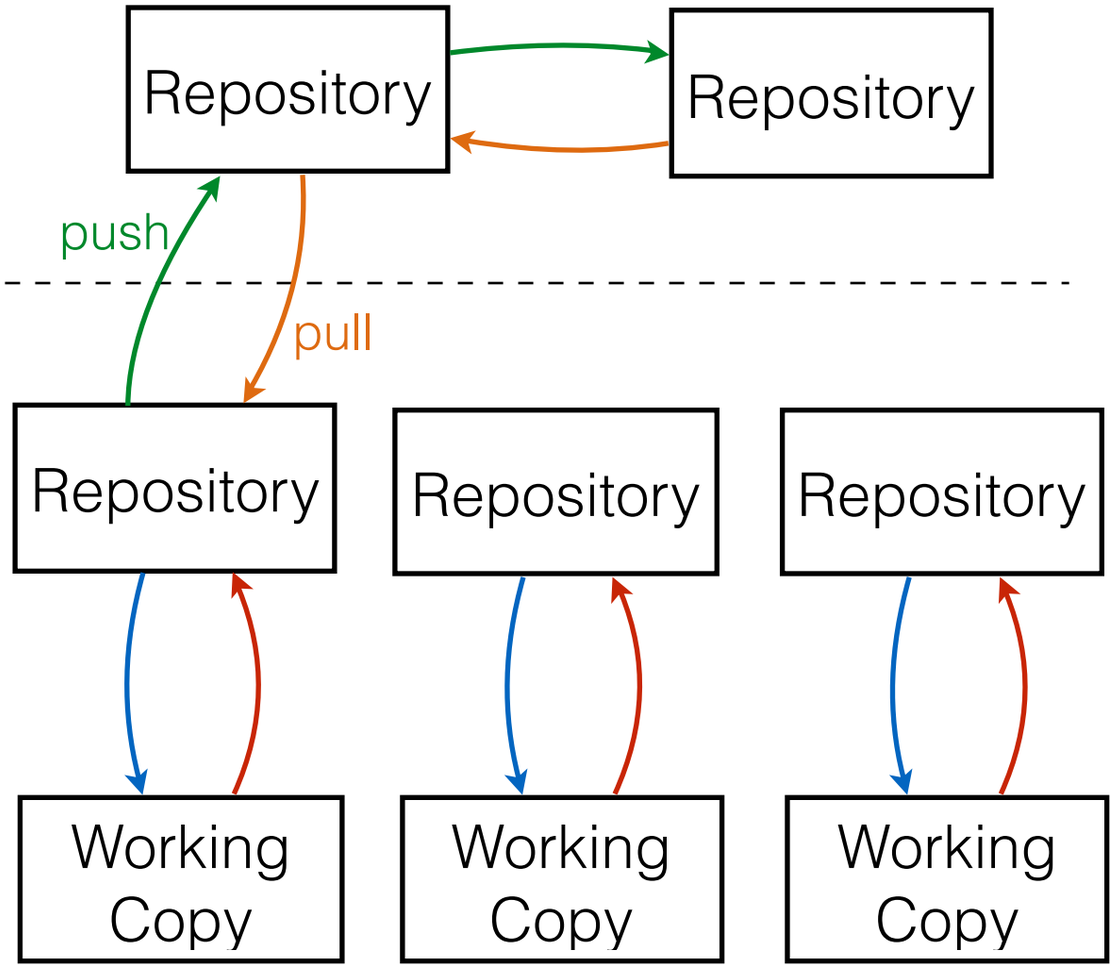
Merge: Take all the changes that have occurred within a branch and merge them (including the history).

Distributed vs. Centralised



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.

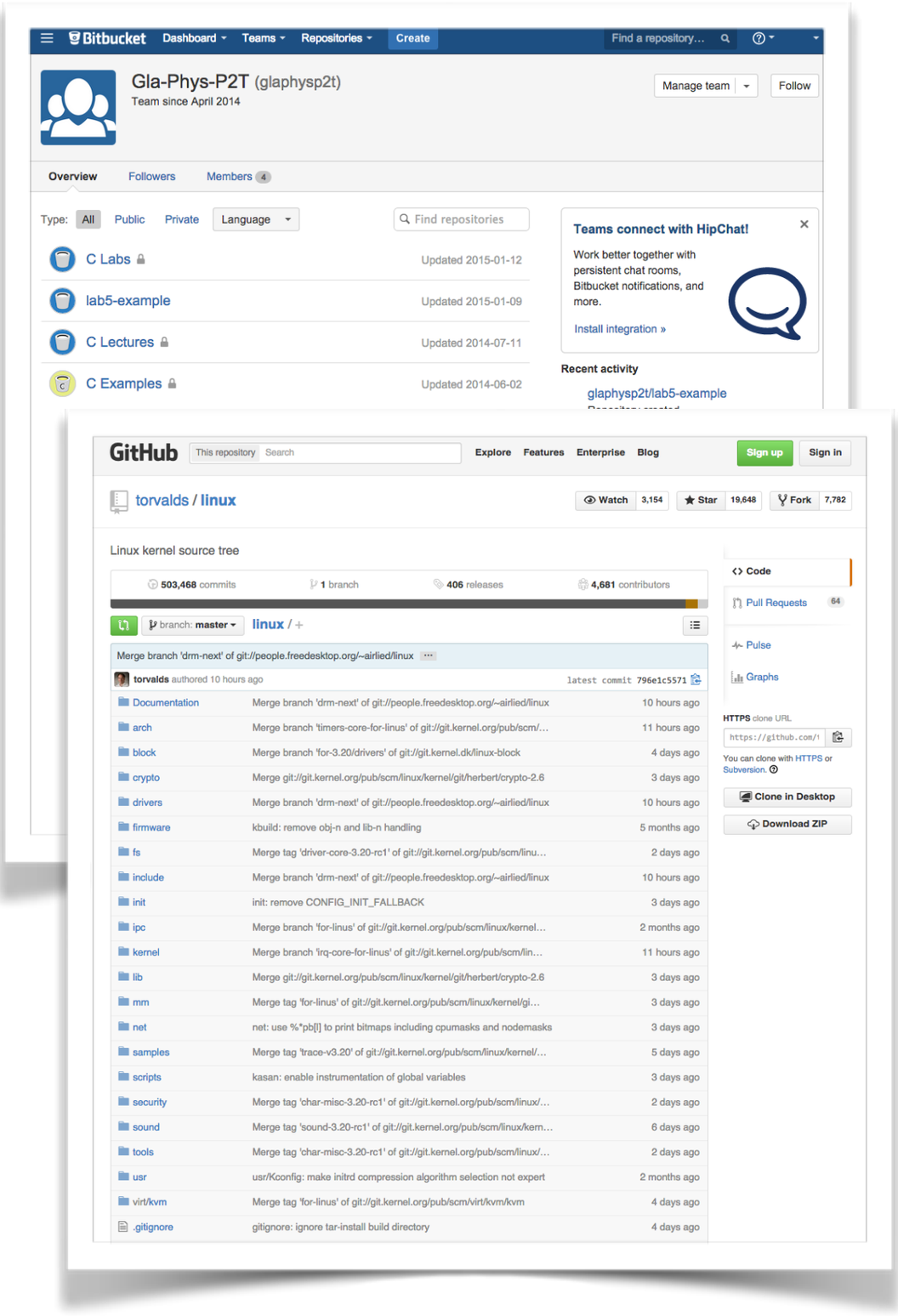
- Lots of online resources are available as well as free online hosting of repositories.

- 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!



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 delete.
- 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 dir
ectory)
#
#       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 dir
ectory)
#
#       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 "Commited and added"
[master c10555f] Commited 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
8fbd35f Refactored to move utility code out of the main file
e532f4f Added randomly generated data
6dcdfe5 Refactored code so that the drawing routines are in their own file, added to the mak
6b4875c Added a simple makefile to build histogram, included a make test which runs the prog
8b2c746 Added a function that calculates the maximum of a list of data and returns it's valu
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 d
d71c39b Added an initial README to the project
gareth@brutha:~/Lec06/git/lab5-example$
```

Git - Log

4. git

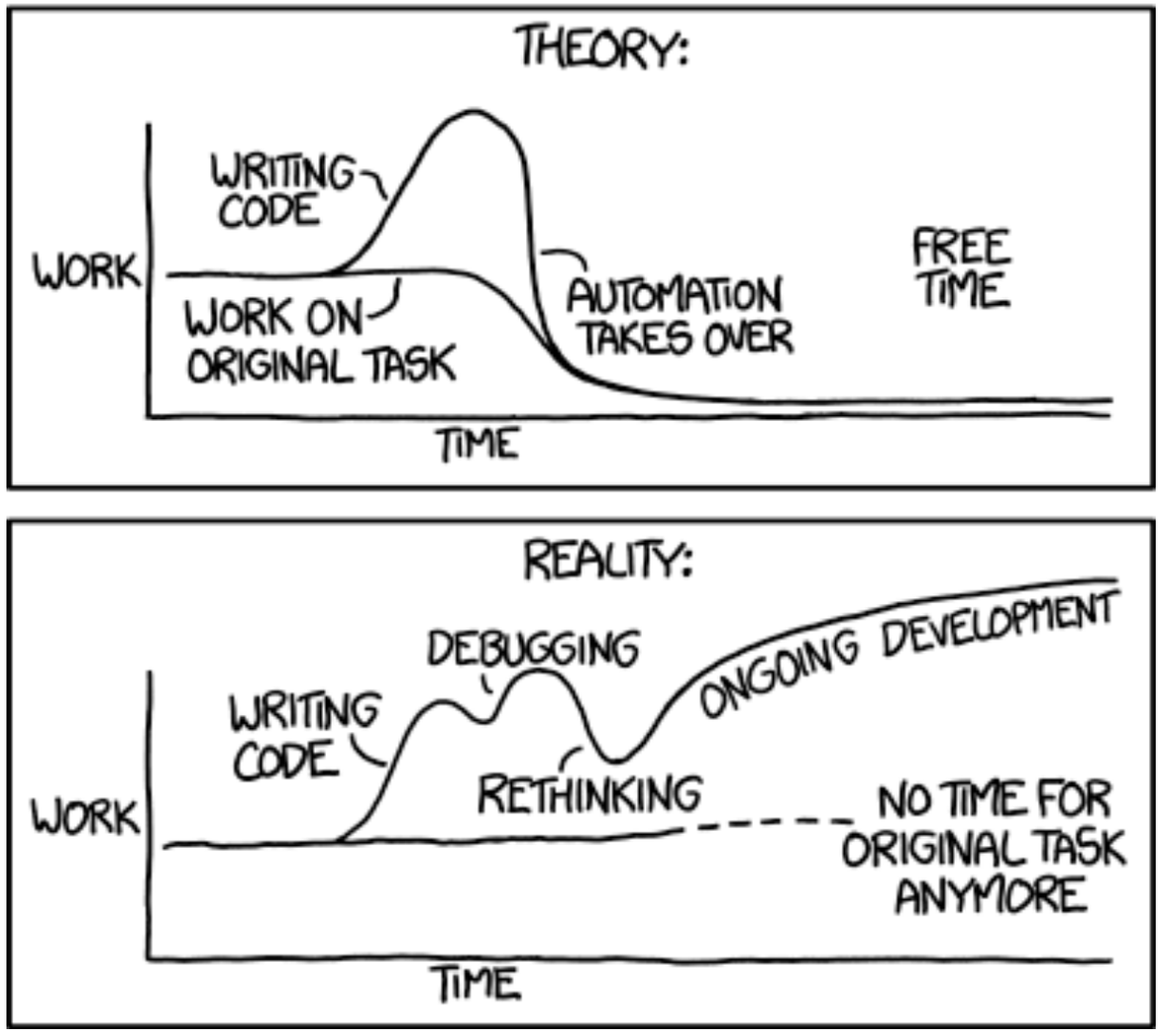
```
Folkvangr:CLabs-Tex gareth$ git log --oneline --graph --decorate
* dfb76d6 (HEAD, origin/master, master) Removed more questions from labs
* 3e037f8 Removed some questions
* 14b61b3 Removed some questions
* d9768f0 Draft changes to labs SCS
* 1170b7a First fixed revision, lab01 SCS
*   e96d985 Merge 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



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.

2. gareth@brutha: ~/Lec05/C/multi (ssh)

```
util.h
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        return 0;
16 }
17
~
main.c [+]                                            17,1          All
-- INSERT --
```

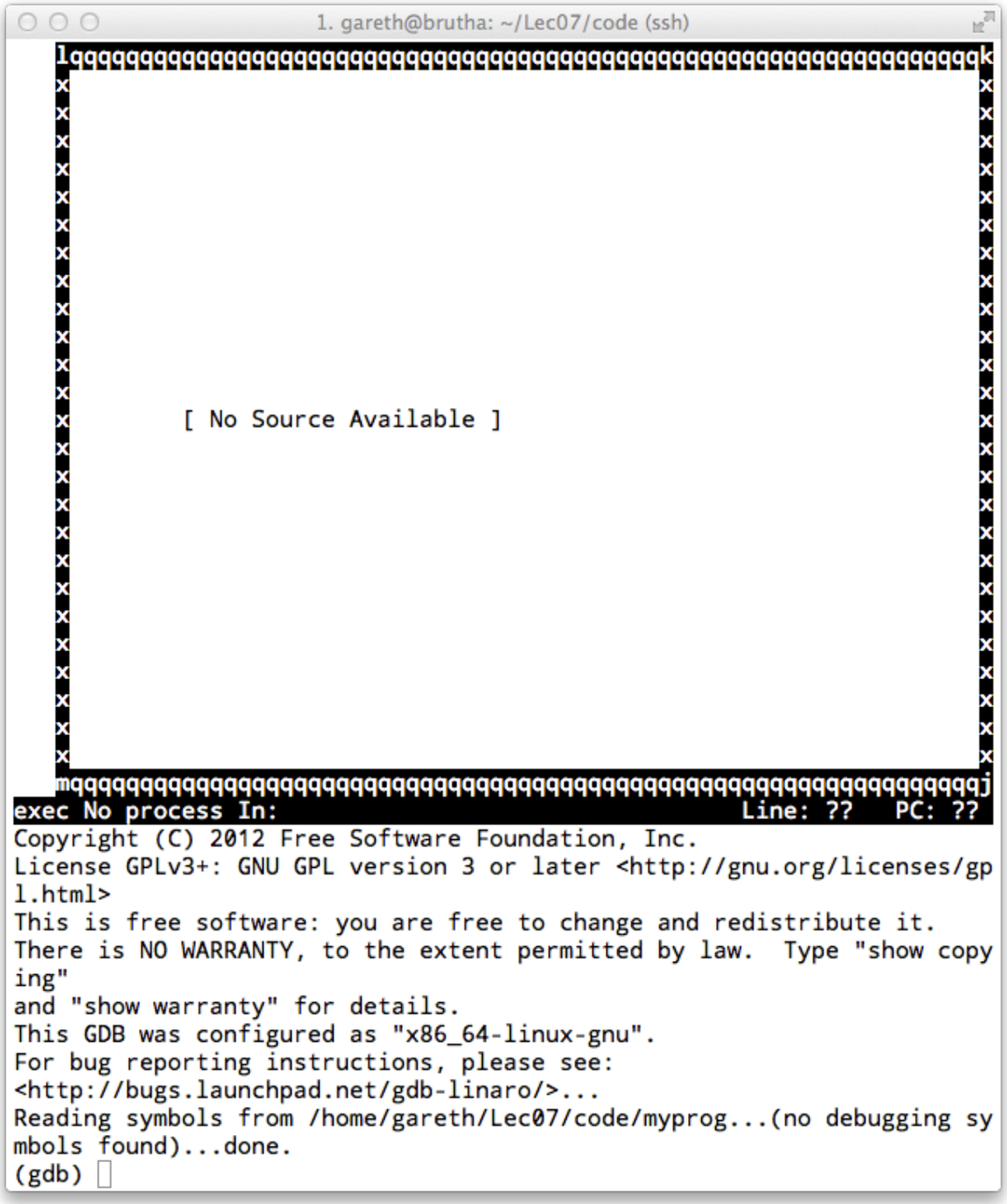
Adding Debug Information

- 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**

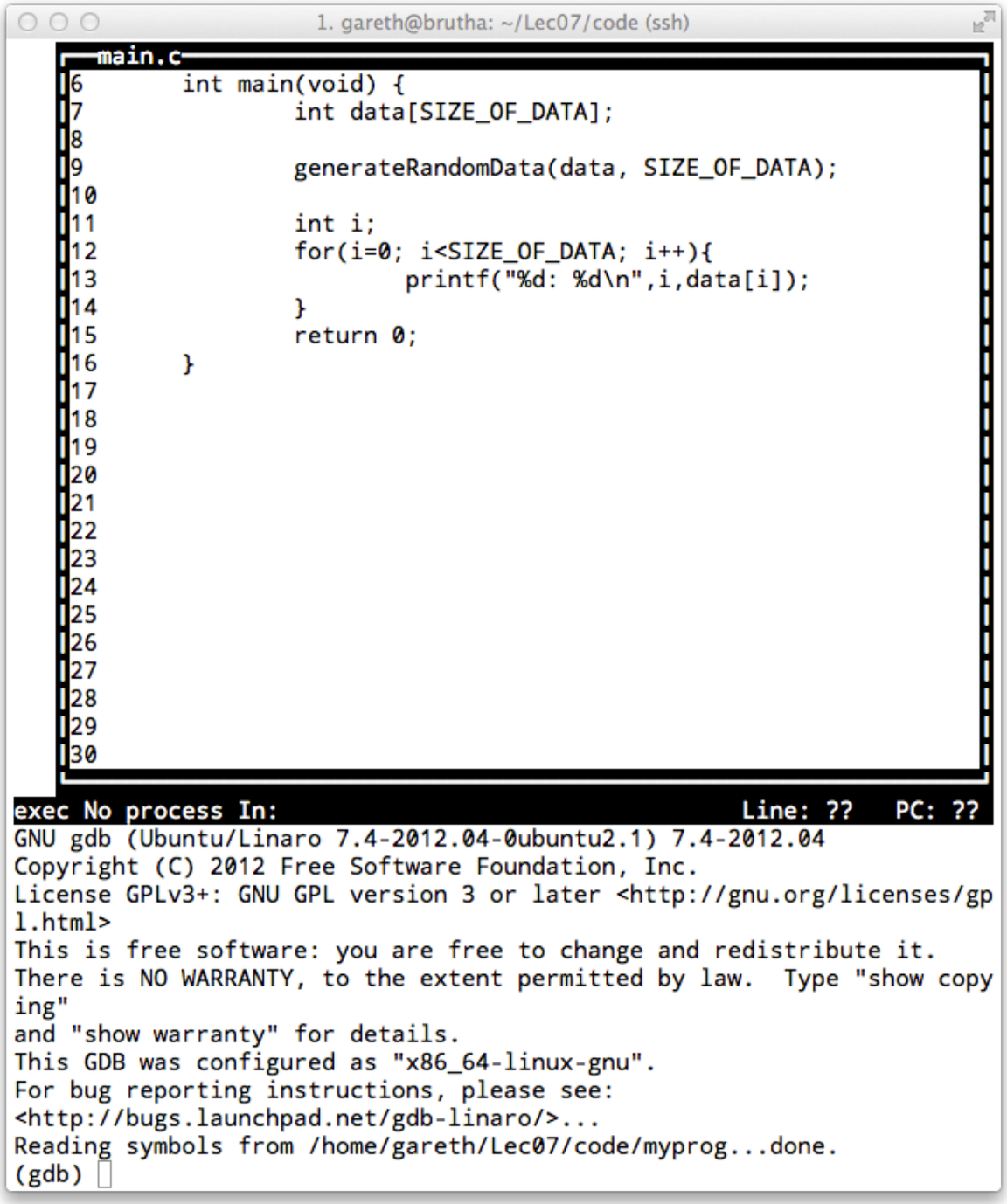
1. gareth@brutha: ~/Lec07/code (ssh)

```
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                                                    1,1           All
"Makefile" 29L, 410C
```

Adding Debug Information



Without **-g**

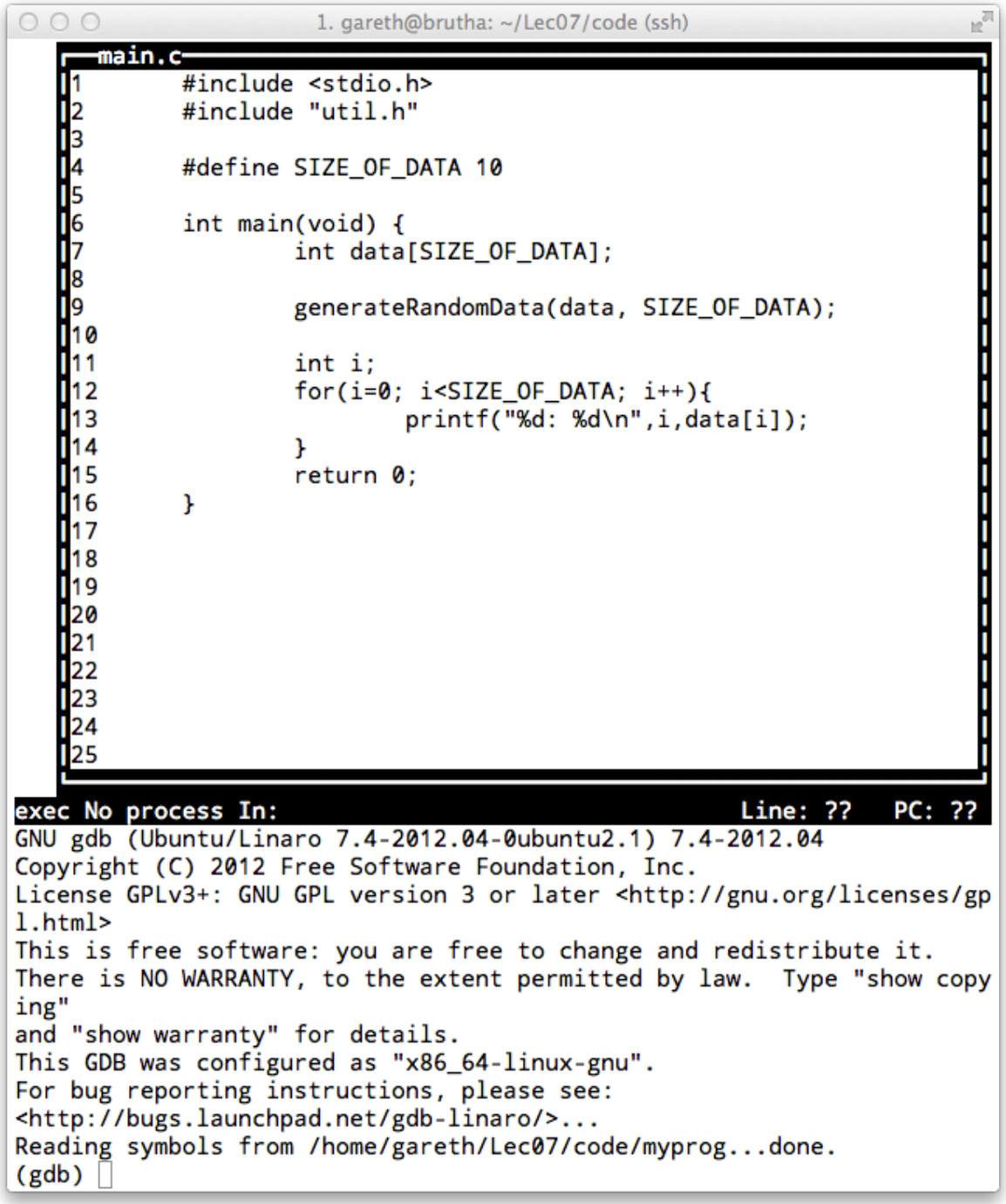


With **-g**

- 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>**



Seeing code in plain GDB

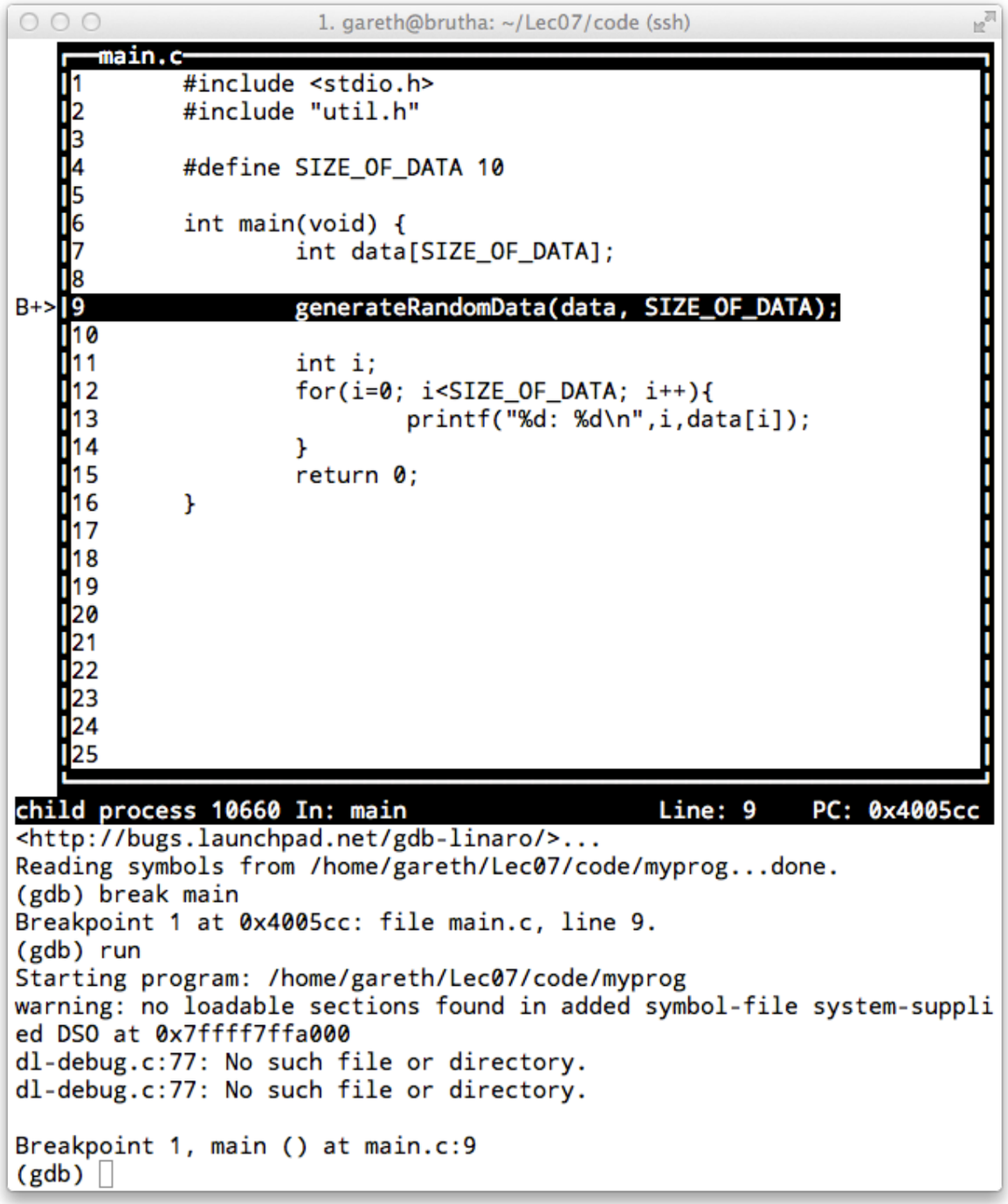
- If we use gdb without the terminal UI, we need to be able to see the code.
- To do this we can use the list command:
  - **list** - by default prints ten lines.
  - **list <line>** - prints the code at the line number specified.
  - **list <function>** - prints the code of the given function
  - **list <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/gp
l.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 copy
ing"
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

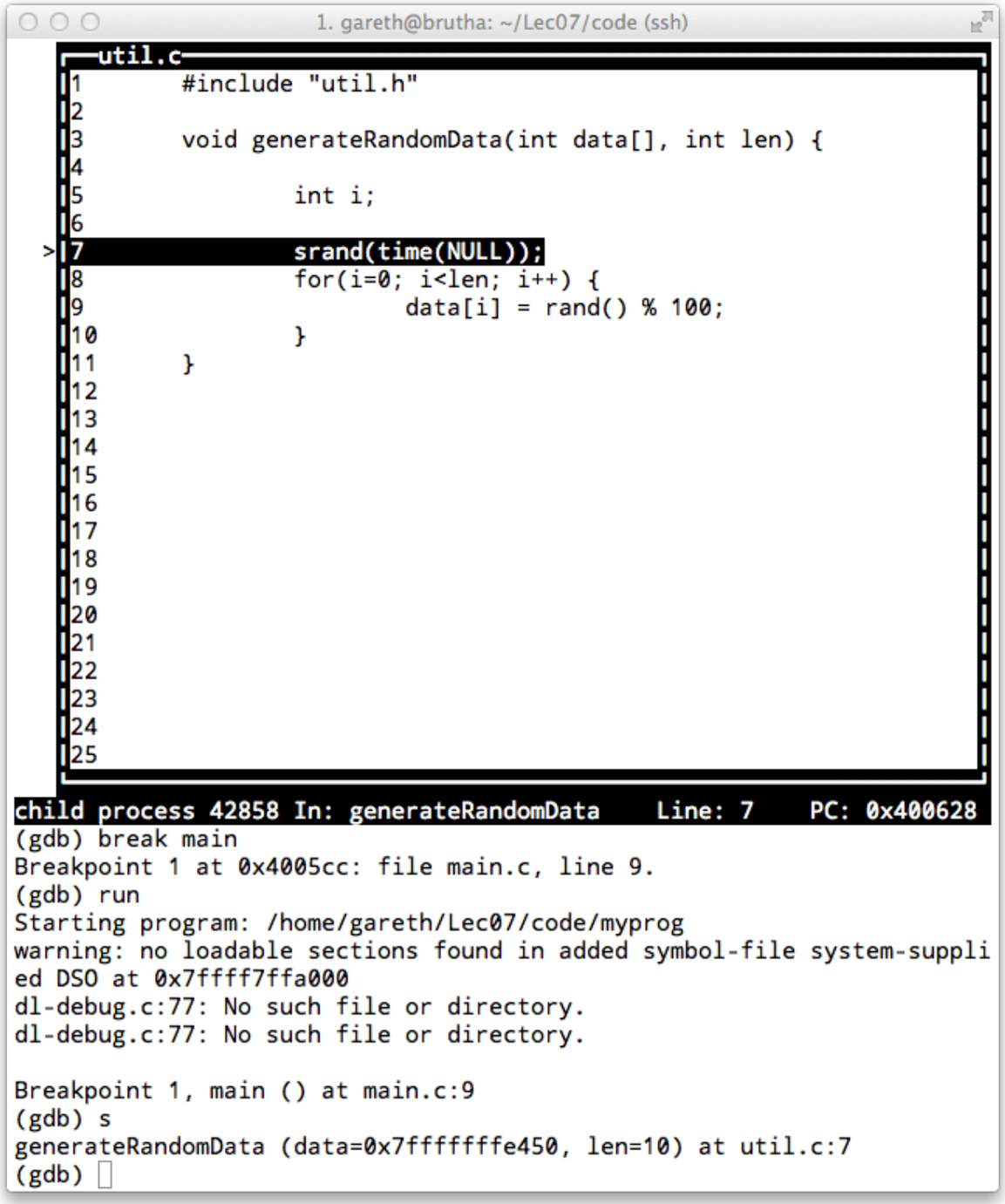
Breakpoints

- 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 you current session by doing
  - **info breakpoints**
- You can use delete to remove set breakpoints:
  - **delete** - deletes all breakpoints
  - **delete <number>** - deletes breakpoint at number



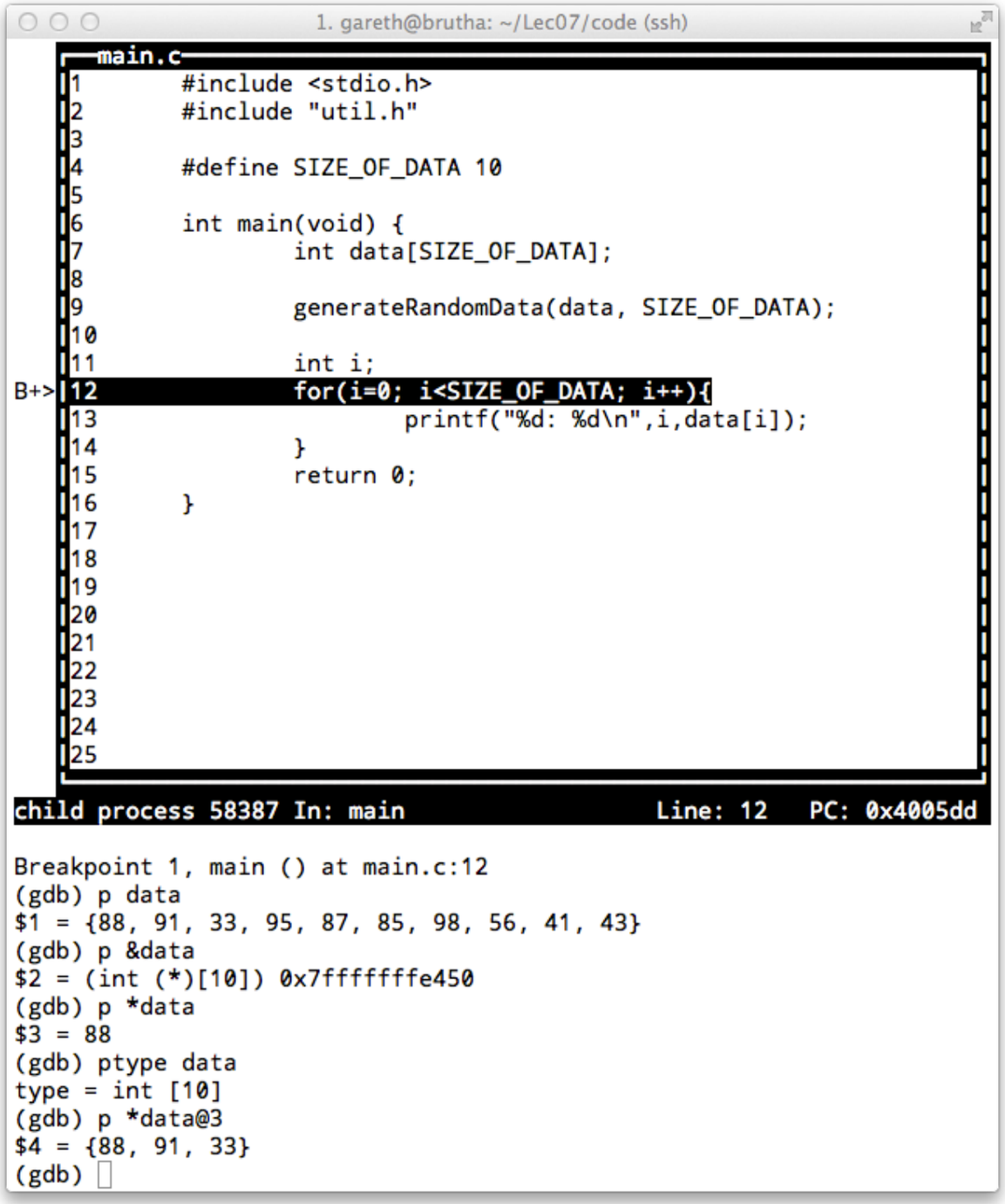
Running the code

- 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.



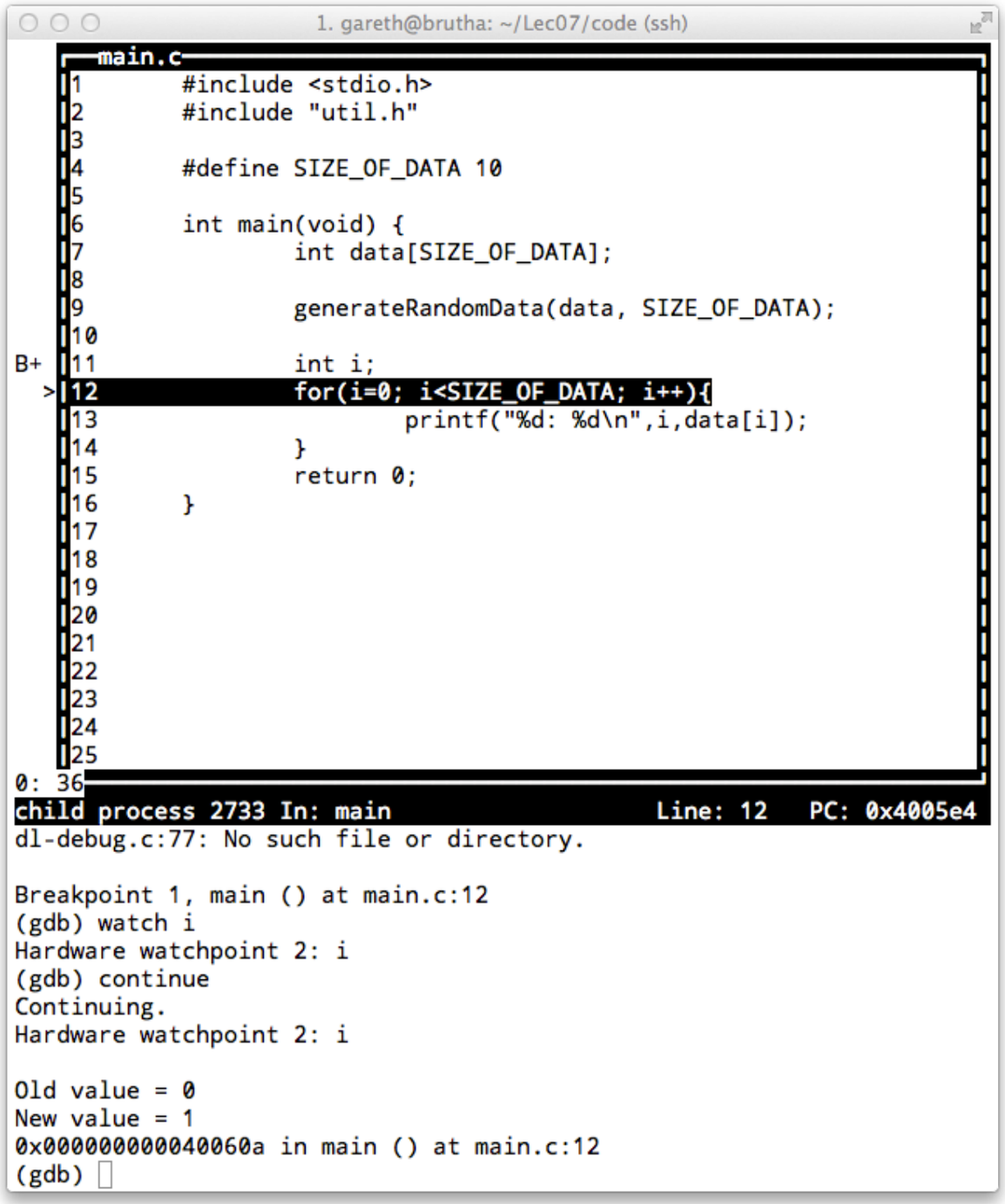
Examining Variables

- 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



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 an 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 if i > 7**



- Getting started
- Running GDB
- Interacting with our Code