Shell

The Unix Command Line

- This course uses the unix command line *shell* a lot and it's the de-facto interface to high performance computers
- We'll use 'ssh' to access a shell on a remote supercomputer
- You can choose you favourite shell, all our examples use *bash*
- The next magic cell sets up the environment and directories etc in a directory `training/tmp` in your home directory. It will be totally wiped in the process: if you run this outside the training accounts of this course, please be careful.

```
cd
rm -rf ${HOME}/training/tmp
mkdir -p ${HOME}/training/tmp
cd ${HOME}/training/tmp
for x in 0 1 2 3 4 5 6 7 8 9
do
  touch examplefile$x
done
for x in 0 1 2 3
do
  mkdir exampledirectory$x
done
touch exampledirectory1/examplefileA .hiddenexample1
echo "Text in a file" > examplefile3
echo DONE.
```

Shell, part 1: The First Commands

Listing files and directories

- produce a simple, locale sorted list of files

```
ls
```

- list "hidden" files, too

```
ls -a
```

- give detailed information (we'll learn to interpret it later)

```
ls -l
```

- list only `examplefile1` and the contents of `exampledirectory1` directory

```
ls -l examplefile1 exampledirectory1
```

- a full usage help; works with almost all commands

```
ls --help
```

- When you need to operate on lots of files, use *glob patterns* like this

```
echo example*
```

```
echo examplefile[0-8]
```

- Do yourself a favour and avoid “special” characters in filenames: TODO!!! `()[]{} &*$?\|#'"‘`
- Yes, that is a space in there!
- All of these work if you *escape* them correctly, but it is complicated:

```
touch horriblen\ame1 horriblen\\ame2 horrible\\name3 horri\\blename4 horrible\ example5
```

- and it gets even worse if you try to write a script and process this programmatically
- besides `\` can give you nasty surprises:

```
ls horr*
```

- oops…

```
rm horrible\name3 horrible example5
```

- but only `/` is really forbidden in file names: you just cannot have it in a file name

Making Directories

- directory is just a special type file
  - initially a directory is like an empty file
  - creating one is equally simple

```
mkdir exampledirectory10
```

- there are other kinds of special files, too: *sockets*, *device nodes*, *symbolic links*, etc

Changing to a different Directory

Go to directory called `exampledirectory1`

```
cd exampledirectory1
```

- and back to where you were

```
cd ..
```

- that `..` refers to the directory containing current directory
- current directory is referred to with `.`

  - hence `..` and `./..` are the same thing

Shell, part 2: Managing Directories and Files

the directory tree

- image of dirtree TODO!!!
- where are we in the tree?

```
pwd
```

Copying Files

- copy `examplefile1` to `examplefile11`

```
cp examplefile1 examplefile11
```

- and `examplefile1` to `exampledirectory2/` with its original name

```
cp examplefile1 exampledirectory2/
```

- or with a new name

```
cp examplefile1 exampledirectory2/newname
```

- this is equivalent to move followed by copy or vice versa (but has different semantics)
- a more sophisticated copying tool is called *rsync*

```
rsync -a exampledirectory2/ exampledirectory12/
```

- limitation: the above can only create one directory level, i.e. `rsync -a exampledirectory2/ exampledirectory12/exampledirectory13/` will fail

Moving Files

- just like copying

```
mv examplefile11 exampledirectory3/
```

- let's move it back to current directory but with a new name

```
mv exampledirectory3/examplefile11 ./newname
```

- remember, directories are files, so can `cp`, `rsync`, `mv` directories just as well as files
- but be careful: `cp|mv|rsync directory dest` behaves differently depending on whether `dest` exists or not

- be extra careful: all these commands overwrite destinations without warning

Removing Files and directories

- remove a file

```
rm examplefile9
```

- but directories cannot be removed unless they are empty

```
rmdir exampledirectory1
```

- so remove the contents first

```
rm exampledirectory1/*
rmdir exampledirectory1
```

- there is a way to do this with one command, but people have removed all their files with it by accident…

Shell, part 3: Working with Files from the Command Line

Displaying the contents of a file on the screen

- for small files

```
cat examplefile3
```

- but this is not useful for big files as they'll scroll off the screen, better one is `less examplefile3` or `more examplefile3` if `less` is unavailable

Searching the contents of a file

- find "This" from a `examplefile3`

```
grep -E "This" examplefile3
```

- or use a *regular expression* or *regex* to match any string with capital "T" followed after any number (including zero) characters by "s"

```
grep -E "T.*s" examplefile3
```

- or "T" followed … by "x"

```
grep -E "T.*x" examplefile3
```

- `man grep` for more details on what a `regex` is

STDIO and friends

- It is often useful to capture the output of a program or send input programmatically to a program: redirection!
- all programs have three non-seekable files open: standard input where user types in, standard output where program writes normal output, and standard error where program is supposed to write error messages
- normally called *stdin*, *stdout* and *stderr*
- redirect stdout with ">"

```
ls > examplefile12
```

- no output: it went to `examplefile2`:

```
cat examplefile12
```

- can also redirect stdin to a file using redirection: this provides input to `grep example` from a file

```
grep example < examplefile12
grep directory < examplefile12
```

- Can also combine these without going via files: *pipes*; note that the following only "pipes" stdout

```
ls | grep example
```

- A more complicated case with stderr ("2>") redirected to `/dev/null` (a black hole):

```
ls i_do_not_exist examplefile1 2> /dev/null | grep example
```

- now errors go to where stdout goes ("&1" means "same as stdout")

```
ls i_do_not_exist examplefile1 2>&1 | grep file
```

- can also swap them around: now stderr is redirected to stdout (2>&1) but stdout is then redirected to `/dev/null` ("1>/dev/null"), so pipe ("|") only gets stderr now

```
ls i_do_not_exist examplefile1 2>&1 1>/dev/null | grep file
```

- order matters: this sends everything to `/dev/null`

```
ls i_do_not_exist examplefile1 1>/dev/null 2>&1 | grep file
```

Shell, part 4: Permissions, Processes, and the Environment

Securing your files

- Basic permissions are for *owner*, *group*, *other*.
- `r` means read, `w` write, `x` execute (or "change into" for directories)

```
ls -la
```

- Careful! Prmissions on directory control new file creation and deletion, so can "steal" files! (Just demonstrating the sequence, the original file is already owned by the training user.)

```
mv examplefile3 3elifelpmaxe
cat 3elifelpmaxe > examplefile3
rm 3elifelpmaxe
```

- For shared directories, use `getfacl` and `setfacl` but they have limitations: only files originally created in the directory inherit the ACL, files moved there from elsewhere will need further action.
- ACLs are the only practical way of setting up shared directories
- Give group `users` read access and user `z300` read-write access to `exampledirectory3` and make sure subsequent files and directories created there have similar permissions:

```
setfacl --default --modify u::rw exampledirectory3
setfacl --default --modify g::r exampledirectory3
setfacl --modify u:z300:rw exampledirectory3
setfacl --modify g:users:r exampledirectory3
```

Managing processes

- list your own processes controlled by current (pseudo) terminal

```
ps
```

- or list all processes and threads

```
ps -elfyL
```

- or processes in a parent-child tree

```
ps -eflyH
```

- another way to print the tree; fancy, but not very useful compared to above

```
pstree
```

- two interactive views of processes, including their CPU utilisation

```
top -b -n1
```

- there is also `htop` on most modern machines
- You can execute processes "in the background"

```
sleep 7 &
sleep 5 & kill -SIGSTOP $!
sleep 3
echo 'in a real terminal you could stop a process with C-z and then check what you have in t
```

```
jobs -l
echo 'transfer a process to background'
bg 2
echo 'check it'
jobs -l
echo 'and move one back to the foreground'
fg 1
echo 'normally you get rid of the foreground process with C-c but now we just waited'
```

- Primitive communication between processes is done using *signals*

```
jobs -l
wait
sleep 72 &
sleep 36 &
sleep 3 &
echo 'Send SIGSTOP to the second one'
kill -SIGSTOP %2
echo 'Check what happened.'
jobs -l
echo 'Send SIGTERM to the first process'
kill -SIGTERM %1
echo 'Check'
jobs -l
echo 'Ok, so now it had terminated. Send SIGKILL to the second process'
kill -SIGKILL %2
echo 'Check'
echo 'Wait for %3 to finish'
jobs -l
wait
```

- the notorius segmentation fault or segmentation violation or segfault for short causes the kernel to send the `SIGSEGV` signal to the offending process
- some batch job systems on supercomputers will send `SIGUSR1`, `SIGUSR2`, `SIGXCPU` or `SIGTERM` when your job is about to run out of its allocated time slot
- COSMOS will send `SIGTERM` first, followed by `SIGKILL` if you don't quit peacefully

Shell startup and environment

- When you log in, `bash` will execute several *script* files; basically lists of commands
  - system startup files
  - personal ones in `~/.bash_profile` (login sessions) or `~/.bashrc` (other sessions)

- edit as you see fit, but be careful: mistakes can lead to inability to log in!

- *Environment variables* control how `bash` behaves; most important ones are

  `PATH` list of colon-separated directory names to look, in order, for commands typed in the prompt

  `HOME` your home directory, also available as — under certain conditions

- Useful UNIX commands
  - check the value of a variable PATH

```
echo ${PATH}
```

- check where (in `PATH`) command `ls` lives

```
which ls
```

- easiest way to give a long list of parameters to a program

```
find exampledirectory12 -name 'example*1' -print0 |xargs -0 ls -l
```

Examples / Practicals / Exercises (these go to cookbook, too)

- Remove file called `foo bar`.
- Remove file called `-rf`.
- Remove file called `nasty $SHELL`,
- Remove LaTeX compilation by-products (i.e. files ending in `.log` and `.aux`) in a directory hierarchy which is 10 levels deep (hint: `find`).
- List all executable files in the current directory, including "hidden" ones.
- List of directories in the current directory.
- Write a shell script which outputs "`filename` is older" if `filename` is older than your `~/.bashrc` and "`filename` is newer" it it is not older.
- Create yourself an ssh private-public-keypair and set up key based authentication with your training account on `microcosm.damtp.cam.ac.uk`.
- You should pick one and learn the tricks of at least one text-editor to make your life easier on the terminal. This course does not cover that but popular choices are `emacs` and `vim`; emacs has a good built-in tutorial which you can easily access the first time you start it.

Further resources

here for example

Version Control and Git

- When collaborating on a project, or when producing any substansive code independantly version control is normally invaluable. Version control software saves the changes you make to code meaning if you accidently delete something or introduce a fatal bug the code can be reverted to a previous state prior to this occuring. Good (specifically modern) version control software also makes it easy to have multiple versions of a code, so that several people can work on the same program and even the same file. The version control software makes it easier to merge in the different code that people have produced and provides tools to deal with sitations when two versions of the code conflict (although if your using svn you might as well give up because it's horrible).

- There are many programs to do version control (in fact google docs/Of**e 360 have a primitive forms of it) the one covered here is `git` as it's probably the most useful/common one. Another important version control software worth mentioning is `svn` as a lot of scientific software is distributed using it. `svn` is an older version control software and is good at version control, however it is fairly hopeless at dealing with collaborative projects (apperently modern svn is a bit better, but that's rarely what people use).
- Code in git is organised into repositories, which are self contained projects/ programs. Within a repository there can be many branches, which are copies of the code with different histories. Typically a repository has a master branch which contains the code that is used and development/hotfix branches which are used to add new features/fix bugs without breaking the code in the master branch.
- When dealing with collabortive projects there are various philosopies of how the project ought to be structured, but we won't go into it here.
- To obtain a copy of a repositroy (say the repository this course is stored in) use:

```
git clone <project url>
```

- for this course specifically

```
git clone https://github.com/juhaj/topics-python-in-research
```

- From now on we will be using a clone of the repository of this course to demostrate how `git` works.
- While you can safely edit the repository you have just obtained a copy of, it's good practice to create a new branch for your changes in case you do something stupid,

```
git checkout -b 'dev-newbranch'
```

- which creates and switches to a new branch, in this case called `dev-newbranch` (although you can call yours what you like so long as you don't put bloody spaces in it). The = -b= is there to create a new branch and isn't required to checkout an existing one.
- Now open a file editor and write a new file in the main directory (alternately if you are lazy do `touch newfile.txt`). How do we add this file to version control?

```
git add newfile.txt
```

- tells git that `newfile.txt` should be staged for commit. This has to be done each time you change the file as git will not commit changes if you haven't added them yet. We could continue making changes and adding them as above until we are ready to preform a commit. Each commit is a snapshot of the branch at a given moment, and you can go back to a previous commit if you make a mistake. They are also important if two people are working on the same project. To commit the changes do,

```
git commit -m 'added newfile.txt as test'
```

- The = -m 'added newfile.txt as test' = is unnessicary, it's just a message that goes with the commit to explain the changes made.
- Say you broke the code and can't figure out how to fix it or deleted something important, to go back to a previous, first look at a list of the previous commit with,

```
git log --oneline
```

- you should see the commit you have just made with the message you just wrote, there will also be a sequence of numbers and letters which is the revision id. There will be a list of revisions with commit messages which preceded your change. Copy the id of the commit prior to yours and do,

```
git checkout <previous-revision-id>
```

- This will checkout the previous commit before you made your changes. To revert the changes, first go back to the current commit (=git checkout 'dev-newbranch'=) then call:

```
git revert <previous-revision-id>
```

The Compiler

- While some code (e.g. the shell commands used today and python introduced tomorrow) can be run line by line, others must be compiled first. A compiler takes the code you write and rewrites it in machine code, a set of machine readable commands which are what the computer actually carries out.

- For shell/python this is done on the fly with each line of code read, converted to machine code and executed in turn, such languages called interpeted languages.

- The other class of languages are the compiled languages such as c/c++/Fortran. Code written in a compiled language cannot be run, instead the code must be compiled to an executable with an appropriate compiler. The executable is a binary file which has been written from the original file and can now be run.

Language	Compiler
c	`gcc`
c++	`g++`
fortran	`gfortran`
java	`javac`

- There are multiple compilers for a given language `gcc` , `g++` and `gfortran` are GNU compilers and are most commonly used on Linux.
- Code can be compiled to different types of binary files which have different purposes. There is often different types of code file in a language to deal with class/function decleration and writting libraries of common functions/classes.
- Object files are the result of compiling a single executable. Multiple object files can be combined to produced executables or shared libraries.
- Executables are files which can be run, i.e. they are the programs actually used by the user.
- Shared Libraries (Which go under various different names dependent on how the seperate objects are linked) are files which combine a number of object files and enable these to be imported into a program. Most programs (and indeed many libraries) make use of shared libraries so that existing functionality isn't duplicated. For instance a program written to solve computational fluid dynamics will normally need to include a solver for systems of linear equations. Instead of rewritting this functionality from sratch the code will make use of a shared library where the solver has been implimented (In this case BLAS/ LAPACK).
- For c code the above files can be produced in the following way with `gcc`

```
gcc -c code.c
```

- produces an object

```
gcc -shared -o libcode.so code1.o code2.o
```

- produces a shared library from `code1.o` and `code2.o`. Note this is not the only type of library for instance files ending in `.dylib` and `.a` are also types of library but are used in different ways.

```
gcc -o program code.c
```

- produces an executable `program` from `code.c`. This executable can be run by typing `./program` into the directory containing the executable and presumably does useful things (prints hello world, solves a maths problem, breaks your computer etc).
- Fortran and c++ can be compiled in a similar way using `gfortran` and `g++`.

- Compiled languages need to know the type of variables/functions used in the code. So when code from a shared library or object is used in a file, that file needs to declare the functions/class etc used. These are done in seperate files which contain only the declerations of the functions/classes in the linked library/object and crucially should not contain implimentation. The compiler appends these to the start of the file before compilation, hence in c/c++ these are termed header files and have the suffix `.h` / `.hpp` . Fortran has a related file called a module file, however these do contain the implimentation and are also compiled to probuce objects.
- In general a program consisting of multiple objects/headers is,

```
g++ -I /headerfile/dir -L /library/dir -llibrary -o outputfilename
```

What is the Compiler Doing?

- So what exactly does the compiler do? Say we have a simple piece of c code that adds 1 to an integer in a for loop,

```
int i, count, n;
n = 10;
for (i=0;i++;i<n)
{
  count++;
}
```

- This cannot be understood by the computer and must be rewritten by the Computer as,

```
        MOV r0,#0
        MOV r1,#0
        MOV r2,#0
        MOV r2,#10

loop    ADD r0,#1
        ADD r1,#1
        CMP r1,r2
        BNE loop
```

- In fact the above isn't machine code, but Assembly. However these are the commands the computer is actually carrying out and has a one-to-one corrispondance with machine code, which is effectively the above written in hexadecimal. It is worth pointing out that the above is hardware dependent and for a different computer, particularly if the chip is by a different manufacturer, the output of the compiler will be completely different due to the different chip achitecture.

Makefile and Instulation On Unix

- While compiling a single file can be done effectively with just a compiler, compiling an entire program takes a long time if it has to be done in one go. In addition different computers have different architectures and often store common libraries in different places. Thus most programs on linux (That aren't obtain directly as binary files), are installed using a configure script and a makefile.
- A makefile is a sequence of dependent instructions. Say you had a program which was split into two files `main.cpp` which contains the main program and `another_file.cpp` which contains a lot of function definitions which are used by `main.cpp`. Using the method described above this can be compiled by running the following set of commands,

```
g++ -c another_file.cpp
g++ -c main.cpp
g++ -o executable main.o another_file.o
```

- This could be written in a script and run each time `main.cpp` or `another_file.cpp` was modified. However this has the issue that if one of the files is modified both are recompiled, even though the other hasn't changed. For a large progam consising of ~100 files it becomes prohibative to recompile all of them when a small change may have been made to a single file. This is where makefiles come in, written as a makefile the above is,

```
executable: main.o another_file.o
  g++ -o executable main.o another_file.o

main.o: main.cpp
  g++ -c main.cpp

another_file.o: another_file.cpp
  g++ -c another_file.cpp
```

- Running this script causes only the files that have been modified, and any files which depend on them, to be compiled. Thus if `main.cpp` is changed running the makefile will result in the following commands being executed,

```
g++ -c main.cpp
g++ -o executable main.o another_file.o
```

- As can be seen in this instance the script does not recompile `another_file.cpp` as it has not changed in any way.

- When compiling a program the compiler needs to know where the libraries which the program is dependant on are stored. This is done with a configure script which writes the approriate makefile for the operating system, arcitecture, and file system that the program is being installed on.

Many programs on Linux are (or can be) installed from the source code and this is done as follows,

```
./configure
```

- Finds all the programs dependancies and writes the appropriate makefile

```
make
```

- Compiles the code to object files and links them to create a shared library or executable

```
make install
```

- Copies the resultant shared libraries/executables to somewhere appropriate in the file system so they can be easily acessed. For instance executables could be copied to "/bin" where they are in the system path an can be thus called be simply typing the name of the executable from any directory.
- examples, with last one taking forever to compile because the file is so big (but no -ipo as that will take long even with modular code): how to sort that out

Appendix I: Key Commands

`ls`	list the contents of the current directory
`cd`	changes the directory to that specified, i.e. `cd subdir` changes to subdir
`cp` / `mv`	copy/mv a file or directory (needs -r for cp) to specified location
`rm`	removes a file/directory (with -r)
`mkdir`	creates a new directory
`ssh`	log into a remote machine
`scp` / `rsync`	copies files from a remote machine. `rsync` should be prefered
`echo`	prints a variable to screen
`git`	version control software (not the only one)
`chmod`	changes the permisions (who can read/modify/execute) on a file
`sed` / `grep` / `awk`	file search and manipulation commands, (see their man pages)
`man`	brings up the manual for a given command, can often also be done by appending `--help`

For Mac:

`open`	opens a file with whichever program has been set up as the default for that file (e.g. `open a.pdf` oper