Introduction to unix/linux

A familiarity with unix or linux and particularly the command-line interface is important in computational science. Even if you use graphical tools in your day-to-day work, at some point it is likely that you will have to access a compute server remotely or find that you need the flexibility provided by the command-line. The first set of sessions in the Mathematics: Computational Classes are to introduce you to working on the command-line and the C++ language.

There are many programs which do the same thing. You are allowed to use whatever ones you like best though command-line tools are *strongly* encouraged. The notes provided do not cover all possibilities. In particular, I have chosen to use bash (Bourne Again Shell) as the unix shell as it is widely used as the default shell. Other possibilities include csh, tcsh and zsh: feel free to try them. Some commands are the same between shells, some are different.

Similarly there is a variety of terminals you can use to run the shell in: they have more similarities than differences. OSX has its own terminal and xterm (supplied by X11), which is the same as that found in linux. Linux distributions tend to have several: the different desktops all have their own versions as well as the ubiquitous xterm and others such as aterm. I will switch between different ones.

The choice of text editor (more on this later) is particularly personal and is one we leave to you.

These sessions are aimed at introducing you to a command-line interface and so most of the time will be spent trying things out. If you wish to do this elsewhere (e.g. at a CMTH workstation), that is fine. I will be here to answer any questions. Experimentation is encouraged!

Contact

James Spencer

email: j.spencer@imperial.ac.uk

website: http://www.cmth.ph.ic.ac.uk/people/j.spencer

Unix philosophy

A key idea behind all unix systems is to have many small programs, each of which perform a specific task, rather than large "jack-of-all-trades" programs. To accomplish complicated tasks, tools can be connected together. This flexibility makes it possible to combine utilities to achieve what we want without having to write the utilities ourselves; for example data can be sorted without having to write a sorting program ourselves.

Additional resources

GNU/Linux Command-Line Tools Summary by Gareth Anderson

A little dated but an excellent and comprehensive introduction to the Linux command line. http://tldp.org/LDP/GNU-Linux-Tools-Summary/html/GNU-Linux-Tools-Summary.html

Advanced Bash-Scripting Guide by Mendel Cooper

A thorough exploration of scripting and unix utilities but starts without assuming any previous knowledge and gives lots of examples. Invaluable. http://tldp.org/LDP/abs/html/

I have borrowed from both of these in places.

Bash reference manual

The ultimate reference guide. http://www.gnu.org/software/bash/manual/bashref.html

Launching the terminal

The MTMCC course require command line tools and a C/C++ compiler: the gcc suite (http://gnu.gcc.org) is free and includes what is widely regarded as the standard C/C++ compiler.

Linux

A terminal can be launched in most Linux distributions from the main menu: it's typically in the Accessories sub-menu. Some distributions can launch the terminal from the desktop menu (right click on the desktop). Some other (free) programs will be needed: these will be mentioned as needed.

Mac OSX

Enter terminal into the spotlight search box and select the Terminal application. The iTerm2 application (a free download) is another terminal application which might be nicer to use.

Windows

Cygwin (http://www.cygwin.com) provides a Unix-like environment for Windows, including gcc.

Command format

Commands are typically in the format:

```
command [options] argument_1 argument_2 ...
```

Throughout the notes the above format will be used to indicate bash commands. The output from the commands is deliberately not given in general: the given commands should be executed to see what they do. A vital part of learning about a *nix environment is to explore and try things.

Commands referred to in the main text are formatted like *this* (at least the first time they're mentioned).

Commands can take optional arguments (where the position doesn't matter) and positional arguments (which may be compulsory and where the specific position of each argument matters). Options are usually signified with either a single dash ('-') or a double dash ('--'). A single dash is used fort short options (i.e. a single letter) whereas the double dash is used for options which are words. Short options can usually be grouped together, for example the following are equivalent:

```
ls -l -h
ls -lh
```

Short options often have more verbose equivalents. Both long and short options can take arguments (which can stop short options being grouped together).

Help

Most commands come with a help option, which briefly explains the common options and how to run the command:

```
open -h
open --help
```

The *open* command is only really useful under OSX, where it is used to open files and directories. In fact, the Linux *open* command is completely different (and doesn't even have a help option) and is used only in C/C++ programming. Try also *ls --help*.

Much more information can be obtained from the *man page* associated with the command:

```
man ls
man man
```

The man page can be scrolled forwards and back and searched by typing / followed by the search string.

But what if you don't know what the relevant command is in the first place? You can search for it either using google or on the command line using *man*:

```
man -k "copy file"
```

man -k searches a short description of the command for the string provided (in the above example, "copy file"). *man -K* searches the entire man page but this is much slower. Sometimes *man -k* lists many commands (try *man -k copy*!) but at least it provides a starting point.

There are quite a lot of differences in the options for BSD commands (used in OSX) and those in Linux. For this reason I will rarely give options to commands: please look at the help or man pages for these. Some commands are used in examples but not explained: these should be tried and their functions looked up using the man pages.

Files and directories

The *ls* command lists files and directories. With no path specified it lists the contents of the current directory:

```
ls
ls /tmp
```

The string after *ls* is used to list only select files or directories. You can use the ? and * wildcards, which match a single letter and any (or no) letters respectively:

```
ls /t?p
ls D*
ls Documents Downloads
```

Directories are separated using a forward slash. The filesystem starts from the top-level or "root" directory, denoted by /. A path to a directory or file can be specified either as an absolute path (i.e. specified starting from the root) or as a relative path (specified relative to the current working directory). A single dot, '.', represents the current directory and a double dot, '..', represents the parent directory. *pwd* prints the current directory you're in.

The home directory is the directory which contains your files, directories and settings. Generally you only have the necessary permission to create, delete and modify files and directories within your home directory unless another user has granted you permission to do so with their files. This directory is usually `/home/<username>` under Linux and `/Users/<username>` under OS X, where `<username>` is your username. `~` is shorthand for the full path to your home directory.

Task

1. Running

   ```
   ls ~/
   ```

doesn't show any files beginning with a dot. Why not? How can you see them?

You can move around the filesystem using the *cd* command:

```
cd /tmp
pwd
cd
pwd
```

Running *cd* without specifying a directory returns you to your home directory.

Task

2. What does running *cd* - do?

mkdir and *rmdir* create and delete (empty) directories:

```
mkdir my_dir
rmdir my_dir
```

Tasks

3. Run the command

   ```
   mkdir ~/dir1/dir2
   ```

   How can the error be fixed?
4. Create a directory and some file(s) in the directory. Try deleting the directory using *rmdir*. What happens? How can you delete it? (Hint: look at the *rm* man page.)

cp copies files and directories and *rm* deletes files and directories:

```
echo "test" > test_file
cp test_file test_file_2
rm test_file*
```

mv is used to move or rename files and directories:

```
mv file1 file2       # Renames file1 to file2
mv file1 dir1        # Moves file1 to directory dir1
mv file1 file2 dir1  # Moves file1 and file2 to directory dir1
```

cp, *mv* and *rm* have many options governing their behaviour (especially for working with directories). Refer to the man pages for more details.

It is quite useful to create new files (especially for experimenting with) or to update the last modification time of an existing file. The *touch* command does this:

```
touch new_file
ls -l
sleep 5
touch new_file
ls -l
```

The contents of a file (or files) can be viewed using *cat*:

```
echo "test" > test_file
cat test_file
```

cat dumps the whole file to screen which is unhelpful for large files. The *more* command can be used to print the file to the terminal one screen-full at a time. Space moves forwards in the file and search is the same as with man pages. *more* is quite limited: you can only move forwards. *less* (which is actually more than more in this case) allows files to be scrolled through and searched in both directions.

Because bash separates commands on spaces, it is inconvenient to use spaces in the names of files and directories despite the fact that this is common in other operating systems.

Input, output and redirection

Input, output and error information are treated separately in a unix system.

standard input

Standard input is the input from the user. Normally refers to the keyboard.

standard output

Standard output is the output from a program and is printed (by default) to the screen (or more specifically, to the terminal from which the program is run).

standard error

Standard output contains any error messages from a program. This is also printed by default to then screen and so output and error messages appear to be mixed. The key difference between standard error and standard output is that standard output can be buffered and standard error is not. This means that error messages can appear earlier than output if the output buffer is not full. (We shall discuss this distinction a little more in the C++ sessions.)

All three of these can be redirected:

>

Send standard output elsewhere.

2>

Send standard error elsewhere. For example:

```
echo hello world > hi
cat hi hi2 2> err
more err
cat hi hi2 > out 2> err
more out err
```

&>

Sends standard output and standard error to the same place.

<

Takes information from somewhere else (normally a text file) and sends it to standard input as if you had typed it in yourself. For example:

```
tr '[a-z]' '[A-Z]'  < hi
```

Note that using > to redirect output or error to a file will cause that file to be overwritten. Use >>, 2>> and &>> to append to files if they exist.

Sometimes you don't want to see any output and/or error messages. There's a "black hole" which things can be redirected to in this case called /dev/null:

```
cat hi hi2 2> /dev/null
```

Task

5. The operators <, > and >> were described above. There's also (unsurprisingly) a << operator. Find out what it does.

Text editors

Input to and output from command line programs is commonly in plain (ASCII) text. For this reason we must use an editor which does not add formatting information or saves files a proprietary format. This also allows unix tools to interact easily. A text editor (rather than word processor) is used to edit files. Most text editors also have options such as syntax highlighting and automatic indentation which are useful when writing programs.

Some text editors are:

vi* and *vim

A standard unix editor (also available for windows). *vim* is an improved editor based upon *vi*: many systems now alias *vi* to *vim*.

emacs

emacs requires much more resources than vim but is far more than just an editor. Is almost an operating in its own right. See emacs and xemacs under Linux and AquaMacs under OSX.

gedit* and *kate

GUI text editors included with the GNOME and KDE environments respectively. *gedit* is also available for OSX.

TextEdit

OSX. Launch from spotlight.

notepad

Windows only. Very basic.

XCode

OSX. An integrated development environment rather than just a text editor. Launch from spotlight.

The choice of editor is very personal and it's worth spending some time looking at the options and finding one which suits you as they can make a huge difference to your productivity. However some familiarity with vi/vim is recommended as it is available on almost all unix-based systems.

In this course I will use *vim*, but you are welcome to use an editor of your choice. The default behaviour of *vim* can be controlled by options given in the ~/.vimrc file.

Task

6. Familiarise yourself with the *vim* text editor (or another editor of your choice). The vim tutor has its own command:

   ```
   vimtutor
   ```

   The emacs tutor can be run from within emacs using the key combination CTRL-c t. Many other text editors also have a tutorial.

Multiple commands

So far we have only run one command at a time. Multiple commands can be entered on a single line by using the semi-colon to separate the different commands:

```
echo "hello" > hi; more hi
```

Subsequent commands will be run no matter what happens in the previous command. This isn't always appropriate. Sometimes we want to run a command only if the previous one succeeds or fails. This can be accomplished using the operators && and || respectively:

```
cd /tmp  && echo "directory exists" || echo "oops"
cd /tmp1 && echo "directory exists" || echo "oops"
```

compared to:

```
cd /tmp; echo "directory exists"; echo "oops"
```

Commands can also be chained together by sending the standard output from one command to the standard input of another calculation using the pipe "|" operator. This is a main feature of unix systems and is what enables complicated operations to be performed by combining small specialised utilities. For example

```
ls ~/ | tee home_dir_contents
```

sends the output of ls to the input of *tee*. *tee* sends its input to standard output and the filename specified. This is useful as it allows the output of a command to be monitored and also saved for future reference.

Task

7. Work out what each piece of this command does:

   ```
   cut -f1 -d" " ~/.bash_history | sort | uniq -c | sort -nr | head -10
   ```

History

Using the history of recent commands in bash can save a lot of time and typing.

The up and down arrow keys scroll through recent history. You can either repeat the same command or edit it first. The *history* command shows the contents of bash's history file. Normally we are just interested in a fairly recent command. A number following the *history* command causes that number of the most recent commands to be printed out:

```
history
history 10
```

Each command has a number before it and this can be used to repeat that command:

```
!n        # repeats n-th command
!!        # repeats the previous command
!-n       # repeats the n-th previous command (so !-1 is equivalent to !!)
```

You can also use a string to repeat a previous command that involved that string:

```
!string   # repeats the last command starting with string
!?string  # repeats the last command containing string
```

for example:

```
cd /tmp
cd
!?tmp
```

Care should be taken with using ! to re-run a command: it is easy to delete or overwrite files...

There are also special variables that refer to the arguments of the previous command: !* refers to all arguments of the previous command and !$ refers to the last argument of the previous command. These are useful for doing different things to the same file or if there's a typo in the command:

```
echo hello world
echo !*
echo !$
sl /home # /Users on OSX.
ls !$
```

The history can be searched using CTRL-R. Press CTRL-R and start typing. The most recent command matching the entered string will appear. Repeated presses of CTRL-R will cycle through earlier commands. Pressing ENTER will run that exact command again and pressing TAB or the left or right arrow key will allow you to edit the command before running it.

The number of commands stored in the history file (~/.bash_history) is determined by the environment variables HISTSIZE and HISTFILESIZE.

General tips

aliases

An alias can be used to make a command do something else or automatically add certain options. For example:

```
touch test_file
cp test_file test_file2
cp test_file test_file2
rm test_file2
alias cp='cp -iv'
cp test_file test_file2
cp test_file test_file2
```

The alias means cp will (for the rest of the session) print out what it's doing and require confirmation before overwriting an existing file.

auto-completion

Using the TAB key causes bash to attempt to automatically complete the command for you. If there are multiple options then pressing TAB twice will print the possibilities:

```
ec<TAB>
ma<TAB><TAB>
```

echo

The *echo* command repeats anything you type and is useful (amongst other things) to check to see how wildcards will be expanded without running the command:

```
echo "hello word"
echo cp -r D* /tmp
```

exit

The *exit* command closes the terminal. Also try CTRL-d (linux) or CMD-w (OSX).

reset

The *reset* command reinitialises the terminal. This is useful if the text from the terminal becomes nonsensical (which can happen if a program crashes particularly badly).

CTRL-C

CTRL-C stops the current command and returns to a new prompt.

(backslash)

bash treats some characters such as spaces and wildcards as special characters. This is a problem when, for instance, a filename contains a space. Backslash is used to escape the special characters to stop bash from expanding them. Tab completion automatically escapes any spaces in a filename.

~ (tilde)

~ is a shortcut to your home directory. Instead of typing:

```
cd /home/fred/Documents  #/Users/fred/Documents on OSX
```

fred could type:

```
cd ~/Documents
```

~ can also be used to as a shortcut to other users' home directories by typing ~username.

#

Everything in a line following # is treated as a comment and not executed by the shell. This is useful if you've typed a long command but then realise you want to do something else first: you can comment out the command and then come back to it later using the shell history. # is entered by

pressing alt-3 on Macs (this is annoying).

{}

Brace expansion is useful when you have two long but similar filenames and can save a lot of typing. The command separated items in the list are expanded by spaces and prefixed and suffixed by the items directly before and after the braces:

```
echo test{1,2,3}
touch this_is_a_test_file
cp this_is_a{,nother}_test_file
```

Configuration

Bash can be configured in many ways to suit your own preferences: for instance by using aliases or adjusting environment variables.

Environment variables are either for information or control the behaviour of certain programs (such as HISTSIZE). Some useful environment variables are:

SHELL

stores the type of shell being used.

PWD

stores the current directory.

HOME

stores the location of the home directory.

To tell bash to access a variable, a $ is prefixed to the variables name (otherwise it tries to interpret the string):

```
echo SHELL
echo $SHELL
```

Setting an alias or environment variable only persists for the current terminal. Changes can be made permanent by having the commands run when the terminal is initialised.

When bash is invoked as an interactive login shell (i.e. you log into the system from the terminal---typically this means either remote access or without using a graphical OS to launch the terminal), then it loads the user settings from ~/.bash_profile.

When bash is invoked as an interactive non-login shell (i.e. the terminal is launched after you've logged into the computer using a graphical OS), then bash loads the user settings from ~/.bashrc.

(Unfortunately it seems OSX does not follow this rule: the Terminal is launched as an interactive login shell.)

The standard way to reconcile .bashrc and .bash_profile is to include the lines

```
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
```

in ~/.bash_profile so that your user settings in ~/.bashrc are always loaded (along with login-specific settings when relevant).

.bashrc and .bash_profile just contain a list of normal commands (one per line) which are executed during the initialisation of bash.

Task

8. Add an alias to your configuration file so that running ls (with no additional options) gives a coloured output.

Tarballs

Tarballs are a way of collecting several files into a single file to make distributing or archiving the set of files easier. Creation and extraction of tarballs are controlled using options to the *tar* command:

```
touch test_file_{1,2,3}
tar -cvf test.tar test_file_*
tar -tf test.tar
rm test_file_*
ls
tar -xvf test.tar
ls
```

Task

9. Create a directory containing several files. Produce a tarball of that directory. Find out how to compress the tarball both at the same time the tarball is created and after an uncompressed tarball has been created.

Handling data

grep is used to search for a string (more specifically: a *regular expression*) from some input. By default it prints out any line which contains that string. *grep* can take input either from standard input (i.e. from a pipe) or from a file:

```
echo -e "hello\nworld" > hi
cat hi | grep 'hello'
grep 'hello' hi
```

grep has many options which enable it to do things such as print out lines of context before/after the matching line, search with case insensitive, search recursively through directories and so on.

The first and last block of lines in a file can be obtained using the *head* and *tail* commands respectively. By default *head* prints out the first 10 lines in a file and prints out the last 10 lines. If a number is given as an option, then that number of lines is printed out:

```
head file_name
head -2 file_name
tail file_name
tail -2 file_name
```

Another useful option to *tail* is *-f*: this causes *tail* to print the last set of lines in a file and print more output as the file is added to, enabling the progress of a command to be *followed*:

```
some_command > out_file &
tail -f out_file
```

awk is a complete programming language designed for manipulation of text. *sed* (stream editor) is a programming language for applying transformations to texts. There isn't room to describe the full set of functionality provided by them: just one example of each will be shown. In particular *awk* is useful for extracting certain columns and *sed* for doing simple replacements on text:

```
cat <<END > data_file
11 12 13MB
22 22 23MB
33 32 33MB
END
cat data_file
awk '{print $1, $3}' data_file | sed -e 's/MB/ MB/'
```

Both awk and sed are commonly used in compact one-line commands rather than long scripts (though they can be used for that if you wish...).

Note the *s/search_string/replacement_string/* syntax used in the sed command. This should be familiar from the text editor tutorial you took in the previous session and is another example of using regular expressions. The implementation of regular expressions is quite similar across many languages and it is very useful to know at least some simple uses. I quite like the introductions to regular expressions given at http://analyser.oli.tudelft.nl/regex/ and the perlre man page.

Finally often we want to plot data after extracting it. There are many plotting programs available which are more convenient (and scriptable) than a spreadsheet. *gnuplot* and *xmgrace* are two such (free) unix programs and make it easy to plot functions and/or tabular data files quickly. A good gnuplot tutorial can be found http://www.duke.edu/~hpgavin/gnuplot.html. gnuplot and xmgrace are available under OSX from the MacPorts system.

The command-line utilities are very useful, however bash is limited and for more complicated situations other scripting languages such as perl and python are easier to use.

Task

10. Download this file: http://www.cmth.ph.ic.ac.uk/people/j.spencer/cdt/names.txt.
    a. Use grep to print your name and the names of the people in the lines directly above and below your name.
    b. Use awk and sed to print the file in the format of FORNAME SURNAME rather than SURNAME, FORNAME.

Scripts

Frequently the results from several calculations needs to be combined and analysed and the output is verbose: only a small amount relevant data needs to be extracted. Doing data extraction by hand is time consuming---what if you have tens, hundreds or thousands of data files to analyse?---and (perhaps more importantly) prone to error. Instead we can get the computer to do the work for us.

The aims of the remaining sections are to introduce to various ways to handle data files and how to write short programs to automate these tasks. This is a very brief introduction to this area: the man pages and resources mentioned at the start go into these topics in far more depth.

Rather than repeating a long set of commands, we can put them into a file (a script) and then call that script in a single command.

We need to distinguish between the name of a variable and its value. $ is used to retrieve the value of a variable. If v1 is the name of a variable, then $v1 gives the data it contains. The only (common) times a variable is used without the $ prefix is when it is assigned or exported. Compare:

```
a=hello
echo a
echo $a
```

No spaces are allowed around the = during assignment as bash then interprets it to mean something else. The output of a command can be assigned to a variable using backticks, "``" :

```
a=`ls -l`
echo $a
```

A space separated list can be iterated over using a *for* block:

```
for i in list of items
do
    commands involving $i
done
```

The list of items can be obtained from another command:

```
for f in `ls`; do
    echo $f
done
```

All useful languages can test for a condition and then perform different tasks depending upon whether that condition is met or not. There are several ways to do this: we will consider only one here.

The *if* block consists of:

```
if [[ condition ]]; then
    commands
fi
```

The commands are executed if the condition is true.

We used this syntax in the previous session to load in settings from the ~/.bashrc file only if it exists:

```
if [[ -f ~/.bashrc ]]; then
    . ~/.bashrc
fi
```

The construct *[[-f ~/.bashrc]]* [1] evaluates true if and only if the file ~/.bashrc exists. This then avoids attempting to load ~/.bashrc if it doesn't exist (which would then lead to an error).

The if block can be extended to do certain things if different conditions are met and if none are met:

```
if [[ condition1 ]]; then
    commands if condition1 is true
else if [[ condition2 ]]; then
```

```
    commands if condition2 is true
elif [[ condition3 ]]; then
    commands if condition3 is true
else
    commands if none of the conditions are true
fi
```

A more comprehensive discussion of tests and the various test operators that are available can be found here: http://tldp.org/LDP/abs/html/tests.html.

Arguments are passed to the script by running the script followed by the arguments as a space separated list. The list of arguments is stored in the $* variable; the total number of arguments is stored in the $# variable; an individual argument can be accessed using $n, where n is an integer giving the number of the argument (so the first argument is $1, the second argument is $2 and so on).

As an example a script (saved as welcome.sh) to welcome a user would be:

```
# A script to welcome a user.
# Usage: welcome.sh [user]
# If the user is supplied, welcome that user, otherwise welcome the current
# user.
if [[ $# -eq 0 ]]; then
    echo "Welcome `whoami`"
else
    echo "Welcome $1"
fi
```

and a script which shows information about the arguments provided is:

```
# Print out information about the arguments supplied.
# Usage: args.sh as many arguments as you like 1 2 3
i=0
echo "total number of arguments: $#"
echo "all arguments: $@"
echo "first argument: $1"
for a in $@; do
    i=$(($i+1))
    echo $i: $a
done
```

(One drawback of bash scripts is that it's very tricky to do even simple maths in them. I prefer more sophisticated scripting languages---e.g. perl and python---for such cases.)

Task

11. Modify the welcome script so that it will welcome an arbitrary number of users and print a help message if the first argument is "--help".
12. Download a tarball of data files from http://www.cmth.ph.ic.ac.uk/people/j.spencer/cdt/data_files.tar and extract the files in it. Each directory contains a single data file which contains output of a calculation of the dispersion energy between two 1D wires modelled using Hückel theory at various separations. Write a script which:

    a. Extracts the separation and output from each data file.

b. Sorts the by the amount of separation.

c. Prints out a data table.

d. Save the data table to a file and plot it.

(Hint: start on the command line and figure out how to extract and print the required information from one data file first.)

Running commands

How does the shell know where to find the commands entered in the terminal? You could give the full path to each command, but this becomes tedious very quickly. Instead the shell searches a list of directories given by the environment variable $PATH for the command given (unless the command is a shell builtin).

```
echo $PATH
```

An earlier directory takes precedence over a later one: if a command is in two directories that are in $PATH, the executable in the directory which occurs first is used. Useful commands to locate which executable is being used are:

```
which echo
whereis echo
```

You can extend list of directories searched by adding to $PATH, e.g. to add ~/local/bin to your path:

```
export PATH=$PATH:~/local/bin
```

It is vital to have $PATH on the right-hand side, as otherwise the shell would stop knowing where to find the standard commands! (Fortunately if you do clear your $PATH on the command-line you can close and reopen the terminal, as $PATH only persists if you set it in a bash startup file. If this doesn't make sense, try

```
export PATH=~/local/bin
```

and see what happens...)

The first line of the script needs to specify the interpreter (in this case, the bash shell), which is the program which will interpret and execute the commands contained within the script. The interpreter is signaled by the characters "#!", called the *shebang*, followed by the absolute path to the interpreter. Thus a bash script would start:

```
#!/bin/bash
```

and a tcsh script would start

```
#!/bin/tcsh
```

Finally we need to make the script executable. This is done using the *chmod* command:

```
chmod u+x test.sh
```

The script can now be run, by specifying the absolute path to the script. The path of the current directory is ./ or the full path can be used:

```
./test.sh
/Users/jspencer/test.sh
```

Task

13. Find out about the chmod command and the unix concept of file-permissions.

    ```
    ls -l
    ```

    to show file permissions.

14. What does the command:

    ```
    chmod 750 args.sh
    ```

    mean?

15. Why must directories be executable?

Job control

By default bash waits for the command to complete before returning the prompt--- it runs in the foreground. A process can be run in the background by putting & at the end of the command. Compare:

```
sleep 10
sleep 10 &
```

A command that is running can be interrupted (paused) by pressing CTRL-Z to send the interrupt signal to the process. The command can then be transferred to running in the background by the *bg* command. A command running the the background can be transferred to the foreground using the *fg* command. You can have more than one job paused or running in the background, in which case the *jobs* command lists the jobs and a corresponding job identification number. *bg* and *fg* then take an argument to select which job should be run in the background or foreground.

```
sleep 60
<CTRL-Z>
jobs
bg
jobs
fg
```

jobs only shows the processes launched form the current shell. The *top* command displays as many processes that will fit in the screen ranked by a certain property. You can change it to rank by amount of memory used or amount of the cpu being used: press ? to bring up the help to tell you how to do this. Press q to quit.

The *ps* command also shows the current processes.

```
ps -ax -u username
```

shows the processes associated with the username provided.

If you have a runaway process that can't be killed by CTRL-C, then find the process id (PID) from *ps* or *top* and then use the *kill* command:

```
kill -6 PID
kill -9 PID # if kill -6 doesn't successfully kill the process
```

kill -6 kills the misbehaving process in a cleaner fashion than *kill -9* and so should be tried first.

---

1 The syntax *[condition]* can be used to test simpler conditions (but in general using double brackets is easer to debug). In fact, square brackets aren't even needed if the condition is a command which returns an exit code, as we shall see in the C++ course.