Introduction to unix/linux III

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 this session 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 in the first session go into these topics in far more depth.

Scripts

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.

The first line of the script needs to specify the interpreter (in this case, the bash shell). 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

```

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:

```
#!/bin/bash
# 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:

```
#!/bin/bash
# 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.)

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.

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
```

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.

Tasks

1. Test out the scripts given as examples.
2. 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".
3. 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.)
4. Find out about the chmod command and the unix concept of file-permissions. Use

    ```
    ls -l
    ```

    to show file permissions.

What does the command:

```
chmod 750 args.sh
```

mean?

Why must directories be executable?

---

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.