Introduction to unix/linux II

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.

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.

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
```

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.

Tasks

1. Add an alias to your configuration file so that running ls (with no additional options) gives a coloured output.
2. Work out what each piece of this command does:

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

3. 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.
4. 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.