# Linux Tutorial

Version 1.21

*Jon Wakelin, Liam Gretton, Gary Gilchrist, Teri Forey, University of Leicester.*

*Adapted from Michael Stonebank's original course*
*'UNIX Tutorial for beginners'*

This tutorial has been adapted to make use of the University of Leicester HPC facilities SPECTRE and ALICE. If you use either of these facilities for research work which results in a publication you should acknowledge this with one of the following statements:

*This research used the ALICE High Performance Computing Facility at the University of Leicester*

or

*This research used the SPECTRE High Performance Computing Facility at the University of Leicester*

# Tutorial One

## 1.1 Listing files and directories (ls)

When you first login, your current working directory is your home directory. Your home directory has the same name as your user-name, for example, *nye1*, and it is where your personal files and subdirectories are saved.

To find out what is in your home directory type

```
ls
```

The ls command lists the contents of your current working directory.

However, it does not cause all the files in your home directory to be listed, but only those ones whose name does not begin with a dot (.) Files beginning with a dot (.) are known as hidden files and usually contain important program configuration information. They are hidden because you should not change them unless you are familiar with Linux.

To list all files in your home directory including those whose names begin with a dot, type

```
ls -a
```

ls is an example of a command which can take options: -a is an example of an option. The options change the behaviour of the command. There are online manual pages that tell you what options a particular command can take, and how each option modifies the behaviour of the command. The online manual command is covered in tutorial 4.3.

```
ls -l
ls -lt
ls -ls
ls -lrS
ls -lrt
```

## 1.2 Making Directories (mkdir)

We will now make a subdirectory in your home directory to hold the files you will be creating and using in the course of this tutorial. To make a subdirectory called unixstuff in your current working directory type

```
mkdir unixstuff
```

To see the directory you have just created, type

```
ls
```

## 1.3 Changing to a different directory (cd)

The command cd directory means change the current working directory to 'directory'. The current working directory may be thought of as the directory you are in, i.e. your current position in the file-system tree.

To change to the directory you have just made, type

```
cd unixstuff
```

Type ls to see the contents (which should be empty)

### Exercise 1a

Make another directory inside the **unixstuff** directory called **backups**

## 1.4 The directories . and ..

Still in the **unixstuff** directory, type

```
ls -a
```

As you can see, in the **unixstuff** directory (and in all other directories), there are two special directories called **.** and **..**

In Linux **.** means the current directory, so typing

```
cd .
```

There is a space between cd and the dot. There is normally always a space between the command and the argument.

This may not seem very useful at first, but using (**.**) as the name of the current directory will save a lot of typing, as we shall see later in the tutorial. (**..**) means the parent of the current directory, so typing

```
cd ..
```

will take you one directory up the hierarchy (back to your home directory). Try it now.

Typing cd with no argument always returns you to your home directory. This is very useful if you are lost in the file system.

## 1.5 Pathnames (pwd)

Pathnames enable you to work out where you are in relation to the whole file-system. For example, to find out the absolute pathname of your home-directory, type cd to get back to your home-directory and then type

```
pwd
/home/n/nye1
```

### Exercise 1b

Use the commands ls, pwd and cd to explore the file system.

(Remember, if you get lost, type cd by itself to return to your home-directory)

## 1.6 More about home directories and pathnames

### Understanding pathnames

First type cd to get back to your home-directory, then type

```
ls unixstuff
```

to list the contents of your unixstuff directory.  Now type

```
ls backups
backups: No such file or directory
```

This is simply because you have not created a directory called backups.

Now, create a sub-directory of unixstuff named backups:

```
cd unixstuff/
mkdir backups
ls backups/
```

Note that it is not necessary to be in the unixstuff directory to create a subdirectory of it. A quicker alternative would be:

```
mkdir unixstuff/backups
ls unixstuff/backups
```

### ~ (your home directory)

Home directories can also be referred to by the tilde ~ character. It can be used to specify paths starting at your home directory. So typing

```
ls ~/unixstuff
```

will list the contents of your unixstuff directory, no matter where you currently are in the file system.

What do you think the following would list?

```
ls ~
```

What do you think the following would list?

```
ls ~/..
```

## 1.7 Shell Shortcuts for bash

```
Ctrl-A (jump to start of line)
Ctrl-E (jump to end of line)
Ctrl-K (delete (kill) everything from the cursor onwards
Ctrl-W (delete the previous word only)
Ctrl-Y (paste whatever was just deleted)
Ctrl-C (kill/exit a running process)
Ctrl-L (clear the screen)
Ctrl-R (search for previously executed commands)
Tab    (auto-complete command or file/directory name)
↑ / ↓  (scroll back / forwards through previously entered commands)
```

## Summary

| | |
|---|---|
| ls | list files and directories |
| ls -a | list all files and directories |
| mkdir | make a directory |
| cd *directory* | change to named directory |
| cd | change to home-directory |
| cd ~ | change to home-directory |
| cd .. | change to parent directory |
| pwd | display the path of the current directory |

# Tutorial Two

## 2.1 Copying Files and Directories (cp)

cp *file1 file2* is the command which makes a copy of **file1** in the current working directory and calls it **file2**.

What we are going to do now is to take a file stored in an open access area of the file system, and use the cp command to copy it to your *unixstuff* directory.

First, change to your *unixstuff* directory.

```
cd ~/unixstuff
```

Then at the shell prompt type:

```
cp /cm/shared/training/tutorial/science.txt .
```

Don't forget the dot (.) at the end. Remember, in UNIX, the dot means the current directory. The above command means copy the file **science.txt** to the current directory, keeping the name the same.

Directories can also be copied with the cp command, but it's necessary to add the option **–R** to do so. This option means 'recursive' and will copy the contents of the directory as well as the directory itself, ***for example***:

```
cp -R directory1 directory2
```

### Try running

```
cp -R /cm/shared/training/tutorial ~/unixstuff
```

### Exercise 2a

Create a backup of your **science.txt** file by copying it to a file called **science.bak**

## 2.2 Moving files and Directories (mv)

The move command has a variety of similar but subtly different uses. It can be used to move a file to a different location (i.e. a different directory). It can also be used to move *multiple* files to a different directory. It can also be used to rename a file or a directory. ***For example***:

```
mv file1 directory1/
```

This would move *file1* from the current directory into *directory1*.

```
mv file1 file2 file3 directory1/
```

This would move *file1*, *file2* and *file3* from the current directory into *directory1*.

```
mv file1 file2
```

This would rename *file1* as *file2*.

```
mv directory1/ directory2/
```

This would rename a directory. Finally,

```
mv file1 directory/file2
```

This would move *and* rename a file in one step.

We are now going to move the file *science.bak* to your backup directory. First, change directories to your *unixstuff* directory (can you remember how?). Then, inside the **unixstuff** directory, type

```
mv science.bak backups/
```

To see if it worked type

```
ls
ls backups
```

## 2.3 Removing Files (rm) and Directories (rmdir)

To delete (remove) a file, use the rm command. As an example, we are going to create a copy of the **science.txt** file then delete it.

Inside your **unixstuff** directory, type

```
cp science.txt tempfile.txt
ls
rm tempfile.txt
ls
```

In order to delete an empty directory you can use the command

```
rmdir directory
```

However this won't remove directories that already have files in them, instead you can use

```
rm -r directory
```

to recursively delete files in directory (use sparingly - **there is no Recycle bin**!)

You can use the *rmdir* command to remove a directory (make sure it is empty first). Try to remove the **backups** directory. You will not be able to since Linux will not let you remove a non-empty directory.

### Exercise 2b

Create a directory called **tempstuff** using mkdir, then remove it using the rmdir command.

## 2.4 Displaying the contents of a file on the screen

### clear (clear screen)

Before you start the next section, you may like to clear the terminal window of the previous commands so the output of the following commands can be clearly understood.

At the prompt, type

```
clear
```

This will clear all text and leave you with the prompt at the top of the window.

### cat (concatenate)

The command *cat* can be used to display the contents of a file on the screen. Type:

```
cat science.txt
```

As you can see, the file is longer than than the size of the window, so it scrolls past making it unreadable.

### less

The command *less* writes the contents of a file onto the screen a page at a time. Type

```
less science.txt
```

Press the space bar if you want to see another page, type *q* if you want to quit reading. As you can see, *less* is used in preference to *cat* for long files.

### head

The *head* command writes the first ten lines of a file to the screen. First clear the screen then type

```
head science.txt
```

Then type

```
head -5 science.txt
```

What difference did the -5 do to the *head* command?

### tail

The *tail* command writes the last ten lines of a file to the screen. Clear the screen and type

```
tail science.txt
```

How can you view the last 15 lines of the file?

## 2.5 Searching the contents of a file

### Simple searching using less

Using *less*, you can search though a text file for a keyword (pattern). For example, to search through **science.txt** for the word 'science', type

```
less science.txt
```

then, still in *less* (i.e. don't press **q** to quit), type a forward slash (**/**) followed by the word to search for, e.g.

```
/science
```

As you can see, *less* finds and highlights the keyword. Type **n** to search for the next occurrence of the word.

### grep

*grep* is one of many standard UNIX utilities. It searches files for specified words or patterns. First clear the screen, then type

```
grep science science.txt
```

As you can see, *grep* has printed out each line that contains the word science. Or has it?

Try typing

```
grep Science science.txt
```

The *grep* command is case sensitive; it distinguishes between *Science* and *science*.

To ignore upper/lower case distinctions, use the -i option, i.e. type

```
grep -i science science.txt
```

Often when there is a lot of text it is useful to highlight the matches (this is a default setting on ALICE / SPECTRE now but may not be on other systems)

```
grep --color -i science science.txt
```

To search for a phrase or pattern, you must enclose it in single quotes (the apostrophe symbol). For example to search for the phrase *spinning top*, type

```
grep -i 'spinning top' science.txt
```

Some of the other options of *grep* are:

-v display those lines that do NOT match
-n precede each matching line with the line number
-c print only the total count of matched lines

Try some of them and see the different results. Don't forget, you can use more than one option at a time, for example, the number of lines without the words *science* or *Science* is

```
grep -ivc science science.txt
```

## wc (word count)

A handy little utility is the *wc* command, short for word count. To do a word count on **science.txt**, type

```
wc -w science.txt
```

To find out how many lines the file has, type

```
wc -l science.txt
```

To find out how many characters the file has, type

```
wc -m science.txt
```

## Summary

| | |
|---|---|
| cp *file1 file2* | copy file1 and call it file2 |
| mv *file1 file2* | move or rename file1 to file2 |
| rm *file* | remove a file |
| rmdir *directory* | remove a directory |
| cat *file* | Display or concatenate a file |
| less file | display a file a page at a time |
| head *file* | display the first few lines of a file |
| tail *file* | display the last few lines of a file |
| grep 'keyword' *file* | search a file for keywords |
| wc *file* | count number of lines/words/characters in file |

# Tutorial Three

## 3.1 Redirection

It is extremely common for processes initiated by Linux commands write to the standard output (that is, they write to the terminal screen), and many take their input from the standard input (that is, they read it from the keyboard). There is also the standard error, where processes write their error messages, by default, to the terminal screen.

- Standard Input (STDIN) - Usually the keyboard
- Standard Output (STDOUT) - Usually the Terminal
- Standard Error (STDERR) - Usually the Terminal

## 3.2 Redirecting Standard Output

We use the **>** symbol to redirect the output of a command. Many of the commands we have seen so far write their output to the terminal (for instance *cat*, *ls*, *grep*, *tail*, *head* and *wc* all write to STDOUT). However we can redirect the output of any of these commands to a file instead (the file can have any name you chose. If the file does not exist it will be created, if it does exist it will be replaced.

The command *echo* prints its arguments to standard output. Compare these two commands

```
echo "Hello World"
```

and

```
echo "Hello World" > output.txt
```

You can view the contents of your new file using

```
less output.txt
```

### Exercise 3a

Create a file called **list1** using a suitable text editor (see appendices

*A.3 Opening a* text editor (PuTTY/SSH) and *A.4 Opening a text editor (NX)* for more information) containing the following items one per line, *orange*, *plum*, *mango*, *grapefruit*. Save and close your file. Now create a second file called **list2** that contains the following items: *apple*, *peach*, *grape*, *orange*. Again save and close your file. You can view your files using a command such as *cat*, *more* or *less*, for example

```
more list1
more list2
```

You should now have two files. We will use the *cat* command to join (con*cat*enate) these files into a new file called **biglist**. Type

```
cat list1 list2 > biglist
```

this command reads the contents of **list1** and **list2** in turn, and then writes the text to the file **biglist**.

## 3.3 Appending data to an existing file

It was mentioned above that the redirection operator, **>**, will create a new file if one does not exist, but it will overwrite the contents of a file if the file already exists. If we want to add/append data to an existing file, rather than overwrite it, we need to use the **>>** operator instead

For example, to append a *kiwi* to the file **biglist** we would type:

```
echo "kiwi" >> biglist
cat biglist
```

You will see that a kiwi was added to the list. Now repeat this using a single **>** operator.

```
echo "Avocado" > biglist
cat biglist
```

You will see that all of the original content of the file has been lost and replaced with the word *Avocado*

## 3.4 Redirecting Standard Error

Standard error and standard output are very similar. Both are generally written to the terminal and it is not always obvious what is STDOUT and what is STDERR. However, STDOUT can be easily differentiated from STDERR using redirection. We redirect Standard Error to a file using the operator **2>**

## 3.5 Redirecting Standard Input

Similarly we can use the **<** operator to redirect STDIN. For example, the **sort** command read input from STDIN (the keyboard) and produces an alphabetically or numerically sort list. Type

```
sort
```

Then type in the names of some vegetables. Press Return after each one, and hit control-d after the last entry to return to the shell.

```
carrot
beetroot
artichoke
^d (control-d to stop)
```

The output will be

```
artichoke
beetroot
carrot
```

Instead of generating STDIN using the keyboard, we can use the **<** operator to redirect the contents of a file to STDIN. For example, to sort your list of fruit, first re-create biglist:

```
cat list1 list2 > biglist
```

then to sort it type:

```
sort < biglist
```

and the sorted list will be output to the screen.

Putting it all together: It is possible to redirect input, output and errors all in one go for example,

```
sort < biglist > sorted_list 2> errors.txt
```

In which case input is read from the file **biglist** (rather than the keyboard)**,** output is sent to the file **sorted_list** (rather than to the terminal) and any error messages are sent to the file **errors.txt** (rather than the terminal).

## 3.6 Pipes

To see who is on the system with you, type

```
who
```

One method to get a sorted list of names is to type,

```
who > names.txt
sort < names.txt
```

This is a bit slow and you have to remember to remove the temporary file called *names.txt* when you have finished. What you really want to do is connect the output of the *who* command directly to the input of the *sort* command. This is exactly what pipes do. The symbol for a pipe is the vertical bar |

The pipe / vertical bar character is usually typed with 'shift' and the key to the left of 'z' on the keyboard.

For example, typing

```
who | sort
```

will give the same result as above, but quicker and cleaner. To find out how many users are logged on, type

```
who | wc -l
```

How would you find out how many login sessions *you* have running? Hint: you will need to use *grep* from Tutorial 2.5

## Summary

| | |
|---|---|
| *command > file* | redirect standard output to a file |
| *command 2> file* | redirect standard error to a file |
| *command >> file* | append standard output to a file |
| *command < file* | redirect standard input from a file |
| *command1* \| *command2* | pipe the output of command1 to the input of command2 |
| cat *file1 file2 > file0* | concatenate file1 and file2 to file0 |
| sort | sort data |

| | |
|---|---|
| who | list users currently logged in |

# Tutorial Four

## 4.1 Wildcards

### The characters * and ?

The character * is called a wildcard, and will match against none or more character(s) in a file (or directory) name. For example, in your **unixstuff** directory, type

```
ls list*
```

This will list all files in the current directory starting with **list....**

Try typing

```
ls *list
```

This will list all files in the current directory ending with **....list**

The character **?** will match exactly one character. So *ls ?ouse* will match files like **house** and **mouse**, but not **grouse**. Try typing

```
ls ?list
ls list?
```

If you need to match a limit number of patterns you can use *{pattern1,pattern2,etc}*

```
ls list{1,2}
```

This can be used with most commands:

```
mkdir newdir{1,2,3,4,5}
```

The previous command would create 5 new directories

## 4.2 Filename conventions

We should note here that a directory is merely a special type of file. So the rules and conventions for naming files apply also to directories.

In naming files, characters with special meanings such as **/ * & %** , should be avoided. Also, avoid using spaces within names. The safest way to name a file is to use only alphanumeric characters, that is, letters and numbers, together with _ (underscore) and . (dot).

File names conventionally start with a lower-case letter, and may end with a dot followed by a group of letters indicating the contents of the file. For example, all files consisting of C code may be named with the ending .c, for example, *prog1.c*. Then in order to list all files containing C code in your home directory, you need only type *ls *.c* in that directory.

## 4.3 Getting Help

### On-line Manuals

There are on-line manuals which gives information about most commands. The manual pages tell you which options a particular command can take, and how each option modifies the behaviour of the command. Type *man command* to read the manual page for a particular command.

For example, to find out more about the *wc* (word count) command, type

```
man wc
```

Alternatively

```
whatis wc
```

gives a one-line description of the command, but omits any information about options etc.

When you are not sure of the exact name of a command,

```
man -k keyword
```

will give you the commands with keyword in their manual page header. For example, try typing

```
man -k list
```

## Summary

| | |
|---|---|
| * | match any number of characters |
| ? | match one character |
| man command | read the online manual page for a command |
| whatis command | brief description of a command |
| apropos keyword | match commands with keyword in their man pages |

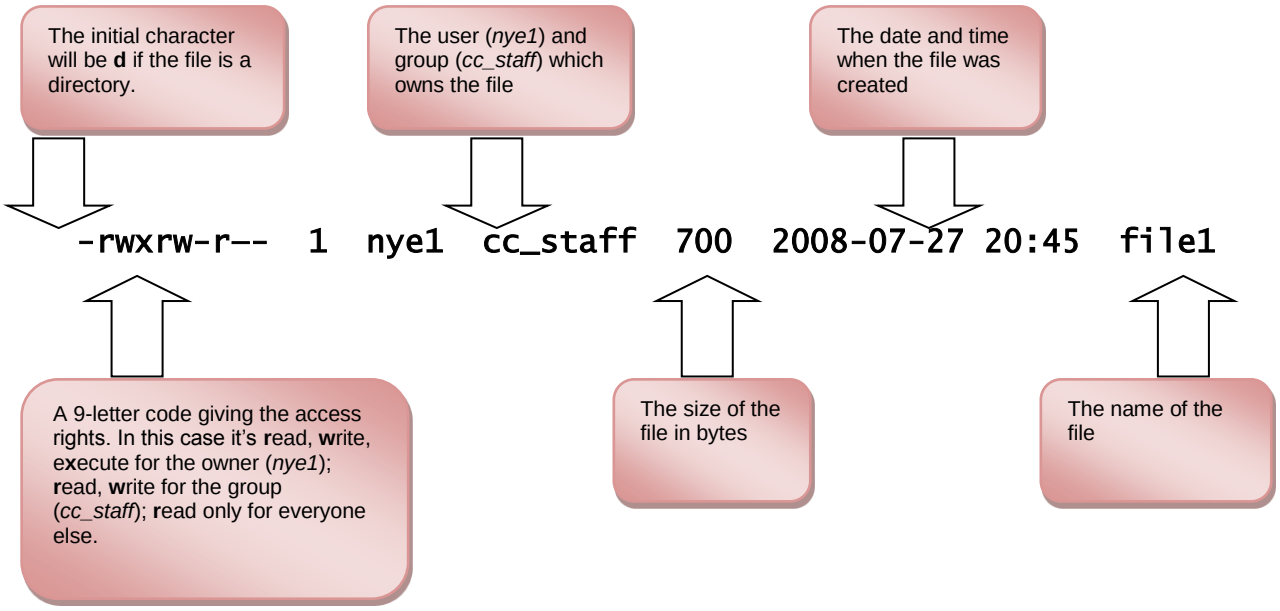
# Tutorial Five

## 5.1 Viewing file and directory permissions

In your unixstuff directory, type

```
ls -l
```

You will see that you now get lots of details about the contents of your directory, similar to the example below.




Each file (and directory) has associated access rights, which may be found by typing ls -l.

In the left-hand column is a 10-symbol string consisting of the symbols d, r, w, x, -, and, occasionally, s or S. If **d** is present, it will be at the left hand end of the string, and indicates a directory: otherwise **-** will usually be the starting symbol of the string, indicating a normal file.

The 9 remaining symbols indicate the permissions, or access rights, and are taken as three groups of 3.

- The left group of 3 gives the file permissions for the user that owns the file (or directory) (*nye1* in the above example);
- The middle group gives the permissions for the group of people to whom the file (or directory) belongs (*cc_staff* in the above example);

- The rightmost group gives the permissions for everyone else.

The symbols r, w, etc., have slightly different meanings depending on whether they refer to a simple file or to a directory.

### Access rights on files

- **r** (or **-**), indicates read permission (or otherwise), that is, the presence or absence of permission to read and copy the file
- **w** (or **-**), indicates write permission (or otherwise), that is, the permission (or otherwise) to change a file
- **x** (or **-**), indicates execution permission (or otherwise), that is, the permission to execute a file, where appropriate

### Access rights on directories

- **r** allows users to list files in the directory;
- **w** means that users may delete files from the directory or move files into it;
- **x** means the right to access files in the directory. This implies that you may read files in the directory provided you have read permission on the individual files.

So, in order to read a file, you must have execute permission on the directory containing that file, and hence on any directory containing that directory as a subdirectory, and so on, up the tree.

### Some examples

-rwxrwxrwx a file that everyone can read, write and execute (and delete).

-rw------- a file that only the owner can read and write - no-one else can read or write and no-one has execution rights.

## 5.2 Changing access rights (chmod)

Only the owner of a file can use chmod to change the permissions of a file. The options of *chmod* are as follows

| Symbol | Meaning |
|---|---|
| u | user |
| g | group |
| o | other |
| a | all |
| r | read |
| w | write (and delete) |
| x | execute (and access directory) |
| + | add permission |
| - | take away permission |

For example, to remove read write and execute permissions on the file **biglist** for the group and others, type

```
chmod go-rwx biglist
```

This will leave the other permissions unaffected.

To give read and write permissions on the file **biglist** to all,

```
chmod a+rw biglist
```

### Exercise 5a

Try changing access permissions on the file **science.txt** and on the directory **backups**

To check that the permissions have changed, use:

```
ls -l
```

## 5.3 Processes and Jobs

A process is an executing program identified by a unique PID (process identifier). To see information about your processes, with their associated PID and status, type

```
ps
```

A process may be in the foreground, in the background, or be suspended. In general the shell does not return the UNIX prompt until the current process has finished executing.

Some processes take a long time to run and hold up the terminal. Backgrounding a long process has the effect that the UNIX prompt is returned immediately, and other tasks can be carried out while the original process continues executing.

### Running background processes

To background a process, type an **&** at the end of the command line. For example, the command *sleep* waits a given number of seconds before continuing. Type

```
sleep 10
```

This will wait 10 seconds before returning the command prompt. Until the command prompt is returned, you can do nothing except wait.

To run *sleep* in the background, type

```
sleep 10 &
[1] 6259
```

The **&** runs the job in the background and returns the prompt straight away, allowing you to run other programs while waiting for that one to finish.

The first line in the above example is typed in by the user; the next line, indicating job number and PID, is returned by the machine. The user is be notified of a job number (numbered from 1) enclosed in square brackets, together with a PID and is notified when a background process is finished. Backgrounding is useful for jobs which will take a long time to complete.

### Backgrounding a current foreground process

At the prompt, type

```
sleep 100
```

You can suspend the process running in the foreground by holding down the Control key and typing **z** (written as **^z**) Then to put it in the background, type

```
bg
```

Note: do not background programs that require user interaction e.g. *nano*.

## 5.4 Listing suspended and background processes

When a process is running, backgrounded or suspended, it will be entered onto a list along with a job number. To examine this list, type

```
jobs
```

An example of a job list could be

```
 [1] Suspended sleep 100
 [2] Running firefox
 [3] Running nedit
```

To restart (foreground) a suspended processes, type

```
fg %jobnumber
```

For example, to restart *sleep 100*, type

```
fg %1
```

Typing *fg* with no job number foregrounds the last suspended process.

## 5.5 Killing a process

### kill (terminate or signal a process)

It is sometimes necessary to kill a process (for example, when an executing program is in an infinite loop)

To kill a job running in the foreground, type **^c** (control-c). For example, run

```
sleep 100
  ^c
```

To kill a suspended or background process, type

```
kill %jobnumber
```

For example, run

```
sleep 100 &
jobs
```

If it is job number 4, type

```
kill %4
```

To check whether this has worked, examine the job list again to see if the process has been removed.

### ps (process status)

Alternatively, processes can be killed by finding their process numbers (PIDs) and using kill PID_number

```
sleep 100 &
ps
    PID TT S TIME COMMAND
  20077 pts/5 S 0:05 sleep 100
  21563 pts/5 T 0:00 netscape
  21873 pts/5 S 0:25 nedit
```

To kill off the process *sleep 100*, type

```
kill 20077
```

and then type ps again to see if it has been removed from the list. If a process refuses to be killed, uses the **-9** option, i.e. type

```
kill -9 20077
```

Note: It is not possible to kill off other users' processes!

## Summary

| | |
|---|---|
| ls -lag | list access rights for all files |
| chmod [options] file | change access rights for named file |
| command & | run command in background |
| ^C | kill the job running in the foreground |
| ^Z | suspend the job running in the foreground |
| bg | background the suspended job |
| jobs | list current jobs |
| fg %1 | foreground job number 1 |
| kill %1 | kill job number 1 |
| ps | list current processes |
| kill 26152 | kill process number 26152 |

# Tutorial Six

## Other useful UNIX commands

### quota

On SPECTRE / ALICE all accounts are allocated a certain amount of disk space on the file system for personal files, up to 20GB. If you go over your quota, you cannot create any more files.

To check your current quota and how much of it you have used, type

```
quotacheck
```

### df

The *df* command reports on the space left on the file system. For example, to find out how much space is left on the fileserver, type

```
df .
Filesystem                    1K-blocks         Used   Available Use% Mounted
on
panfs://172.16.3.1:global 933294615568 846657542400 86637073168  91% /panfs

df -h .
Filesystem                   Size  Used Avail Use% Mounted on
panfs://172.16.3.1:global  870T  789T   81T  91% /panfs
```

### du (disk usage)

The *du* command outputs the number of kilobytes used by each subdirectory. This is useful if you have gone over quota and can no longer log in using NX and you want to find out which directory has the most files (or alternatively, you can use the '*homeusage*' command). In your home-directory, type

```
du *
du –s *
du –sh *
```

### homeusage

The *homeusage* command will do the same as running *du –sh* * in your home directory but will output in ascending order of size to make it easy to see where you are using space. You do not need to be in your home directory to run this command.

## gzip

This command compresses a file. For example, to compress *science.txt*, type

```
gzip science.txt
```

This will compress the file and place it in a file called science.txt.gz. To uncompress the file, use the *gunzip* command.

```
gunzip science.txt.gz
```

## file

*file* classifies the named files according to the type of data they contain, for example ascii (text), pictures, compressed data, etc.. To report on all files in your home directory. It can be useful to determine what sort of data a file contains in cases where the file name doesn't give a hint. Type

```
file filename
```

## history

The shell keeps an ordered list of all the commands that you have entered. Each command is given a number according to the order it was entered.

```
history
```

You can use the exclamation character (!) to recall commands easily.

```
!!      # recall last command
!-3     # recall third most recent command
!5      # recall 5th command in list
!grep   # recall last command starting with grep
```

You can increase the size of the history buffer by typing

```
HISTSIZE=1000
```

## find

*find* is a powerful but rather complicated command for finding files. By default it searches recursively from the directory specified.

The first argument to the file command is the directory to start searching from. In its simplest form the command then needs a name of an object to search for, and this

must be specified as an argument to the **–name** option. The following example will look for any object called *file1*, and will start searching from the current working directory:

```
find . –name file1
```

To find all objects beginning with *file*, a wildcard can be used, but it must be quoted:

```
find . –name "file*"
```

## locate

*locate* is a very quick way of finding files on a large system. It performs a similar role to the *find* command but works in a very different way. *find* looks through the file system until it finds your files (which can be slow but is almost always correct); *locate* on the other hand searches a database in which the locations of files are maintained. This is far quicker but doesn't reflect very recent changes to the file system, because the database is usually only updated once a day.

```
locate filename
locate -i filename
locate -r filename
```

If your home directory is mounted on a shared filesystem such as NFS, then the database which the locate command queries may not include your home directory.

## wget

*wget* is a web client (not a browser). It can be used download files from web and ftp sites:

```
wget URL
wget http://www.ee.surrey.ac.uk/Teaching/Unix/science.txt
wget -O sci.txt http://www.ee.surrey.ac.uk/Teaching/Unix/science.txt
```

# Tutorial Seven

## 7.1 Variables

Variables are a way of passing information from the shell to programs when you run them. Programs look "in the environment" for particular variables and if they are found will use the values stored. Some are set by the system, others by you, yet others by the shell, or any program that loads another program.

## 7.2 Environment Variables

An example of an environment variable is the *OSTYPE* variable. The value of this is the current operating system you are using. When using variables it's necessary to refer to them with a **$** sign at the start so that the shell knows that we are referring to a variable. Type

```
echo $OSTYPE
```

More examples of environment variables are

- $USER - Your login name
- $HOME - Path name of your home directory
- $HOSTNAME - Name of the computer you are using
- $PATH - Directories the shell searches to find commands
- $SHELL – The shell you are using (should be bash!)

Environment variables are displayed using the *env* command. To show all values of these variables, type

```
env | less
```

## 7.3 Setting variables

You can set shell variables using the **=** operator. For example, to change the number of shell commands saved in the history list, you need to set the shell variable HISTSIZE. It is set to 1000 by default on SPECTRE, but you can increase this if you wish.

```
HISTSIZE=2000
```

Check this has worked by typing

```
echo $HISTSIZE
```

However, this will only set the variable for the current shell - it will be lost once you log out. You should also be aware that the new value of the variable will

1. Not be picked up by any forked processes, including sub-shells
2. Not be picked up by any new sessions that you start

To address the first issue, you can export the variable – this means that forked processes and sub-shells will inherit the variables,

```
export HISTSIZE=2000
```

To address the second issue (i.e to make the changes permanent) you will need to add the above command to your ***.bashrc*** file.

First open the *.bashrc* file in *nano* (or another suitable text editor – (see appendices

*A.3 Opening a* text editor (PuTTY/SSH) and *A.4 Opening a text editor (NX)* for more information)). If you have connected using PuTTY/SSH, use *nano*:

```
nano ~/.bashrc
```

Add the following line to your *.bashrc* file (it doesn't matter where within the file as long as it's on a line of its own):

```
export HISTSIZE=2000
```

Save the file and force the shell to reread its *.bashrc* file by using the shell *source* command:

```
source ~/.bashrc
```

Alternatively you could log out and then start a new shell. Finally, check this has worked by typing

```
echo $HISTSIZE
```

## 7.4 Setting the path

When you type a command, your PATH variable defines in which directories the shell will look to find the command you typed. It is a colon-separated list of directories, and on SPECTRE can be very long, reflecting the number of applications and program modules you may have loaded.

```
echo $PATH
```

will show a long list of directories.

If the system returns a message saying *command: command not found*, this indicates that either the command doesn't exist at all on the system or it is simply not in your path.

For example, to run *units* which we will compile and install in Tutorial Eight you either need to directly specify the units path (*~/units174/bin/units*), or you need to have the directory *~/units174/bin* in your path. Come back to the rest of Tutorial 7.4 once you have completed Tutorial Eight.

You can add it to the end of your existing path (the **$PATH** represents this) by issuing the command:

```
 export PATH=$PATH:~/units174/bin
```

Don't forget the colon, which separates the existing list of directories from the one you are adding to the list.

Test that this worked by trying to run *units* in any directory other than where *units* is actually located.

```
cd; units
```

**HINT:** You can run multiple commands on one line by separating them with a semicolon.

To add this path **permanently**, add the following line to your *.bashrc* list of other commands.

```
PATH=$PATH:~/units174/bin
```

## which

The which command shows you the full path to a command (provided that the file is in the path)

```
which command
which wget
```

If there are multiple programs with the same name, you can use:

```
which -a command
```

to list them all. However you should realize that if there are multiple programs with the same name in your path only the one listed first will be executed.

# Tutorial Eight

## 8.1 Compiling software packages

We have many public domain and commercial software packages installed on SPECTRE which are available to all users. However, users are allowed to download and install small software packages in their own home directory, software usually only useful to them personally.

There are a number of steps needed to install the software.

- Locate and download the source code (which is usually compressed)
- Unpack the source code
- Compile the code
- Install the resulting executable
- Set paths to the installation directory

Of the above steps, probably the most difficult is the compilation stage.

### Compiling Source Code

All high-level language code must be converted into a form the computer understands. For example, C language source code is converted into a one or more object files containing low-level machine code. The final stage in compiling a program involves linking the object files to libraries which contain certain built-in functions. This final stage produces an executable program, code which the computer's CPU can execute directly.

To do all these steps by hand is complicated and beyond the capability of the ordinary user. A number of utilities and tools have been developed for programmers and end-users to simplify these steps.

### make and the Makefile

The *make* command allows programmers to manage large programs or groups of programs. It aids in developing large programs by keeping track of which portions of the entire program have been changed, compiling only those parts of the program which have changed since the last compile.

The make program gets its set of compile rules from a text file called **Makefile** which resides in the same directory as the source files. It contains information on how to compile the software, e.g. the optimisation level, whether to include debugging info in the executable. It also contains information on where to install the finished compiled binaries (executables), manual pages, data files, dependent library files, configuration files, etc.

Some packages require you to edit the Makefile by hand to set the final installation directory and any other parameters. However, many packages are now distributed with the GNU configure utility.

### configure

As the number of UNIX variants increased, it became harder to write programs which could run on all variants. Developers frequently did not have access to every system, and the characteristics of some systems changed from version to version. The GNU configure and build system simplifies the building of programs distributed as source code. All programs are built using a simple, standardised, two-step process. The program builder need not install any special tools in order to build the program.

The configure shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a **Makefile** in each directory of the package.

The simplest way to compile a package is:

1. **cd** to the directory containing the package's source code.
2. Type **./configure** to configure the package for your system.
3. Type **make** to compile the package.
4. Optionally, type **make check** to run any self-tests that come with the package.
5. Type **make install** to install the programs and any data files and documentation.
6. Optionally, type **make clean** to remove the program binaries and object files from the source code directory

The *configure* utility supports a wide variety of options. There is usually a help option available to get a list of interesting options for a particular configure script.

```
./configure --help
```

The only generic options you are likely to use are the --prefix and --exec-prefix options. These options are used to specify the installation directories.

The directory named by the --prefix option will hold machine independent files such as documentation, data and configuration files.

The directory named by the --exec-prefix option, (which is normally a subdirectory of the --prefix directory), will hold machine dependent files such as executables.

## 8.2 Downloading source code

For this example, we will download a piece of free software that converts between different units of measurements.

First create a new directory then copy the software and save it to your new directory.

```
mkdir download
cd download
cp /cm/shared/training/tutorial/units-1.74.tar.gz .
```

## 8.3 Extracting the source code

Make sure you are within the **download** directory and list the contents.

```
cd ~/download
ls -l
```

As you can see, the filename ends in **.tar.gz**. This is a common file format for distributing software packages in source form. It comprises a tar file which has been compressed with gzip (tar.gz files are often called *tarballs*). A tar file is a collection of directories and files packaged as a single file with the *tar* command. Sometimes files of this type are named ending with **.tgz**

First uncompress the file using the gunzip command. This will create a .tar file.

```
gunzip units-1.74.tar.gz
```

Then extract the contents of the tar file.

```
tar -xvf units-1.74.tar
```

Alternatively the two steps can combined into a single command:

```
tar -zxvf units-1.74.tar.gz
```

Notice the extra -z flag. This instructs tar to gunzip the file before unpacking the archive. Again, list the contents of the **download** directory, then go to the **units-1.74** sub-directory.

```
cd units-1.74
```

## 8.4 Configuring the package

The first thing to do is carefully read the **README** and **INSTALL** text files (use the *less* command). These contain important information on how to compile and run the software.

The units package uses the GNU configure system to compile the source code. We will need to specify the installation directory, since the default will be the main system area which you will not have write permissions for. We need to create an install directory in your home directory.

```
mkdir ~/units174
```

Then run the configure utility setting the installation path to this.

```
./configure --prefix=$HOME/units174
```

If configure has run correctly, it will have created a Makefile with all necessary options. You can view the Makefile if you wish (use the *less* command), but do not edit the contents of this.

## 8.5 Building the package

Now you can go ahead and build the package by running the make command.

```
make
```

After a minute or two (depending on the speed of the computer), the executables will be created. You can check to see everything compiled successfully by typing

```
make check
```

If everything is okay, you can now install the package.

```
make install
```

This will install the files into the **~/units174** directory you created earlier. It is important to realize that while the

```
./configure
make
make check
make install
```

sequence is extremely common, it is not a standard and there is no absolute guarantee that software will install this way. However, the *make* command is a standard Linux command which reads in a user written file (the makefile) that describes how to build and install software. If you are writing your own software (and in particular if you are distributing your software) you should look to use the make command - although its use is beyond the scope of this course. The configure command is more in-depth still and will create a makefile for a given system from a template file, you will probably not need to create your own *configure* scripts unless you are working on developing a very large software application and intended to distribute it widely.

## 8.6 Running the software

You are now ready to run the software (assuming everything worked).

```
cd ~/units174
```

If you list the contents of the units directory, you will see a number of subdirectories.

| | |
|---|---|
| bin | The binary executables |
| info | GNU info formatted documentation |
| man | Man pages |
| share | Shared data files |

To run the program, change to the **bin** directory and type

```
./units
You have: 6 feet
You want: metres
* 1.8288
/ 0.54680665
(ctrl-d to exit)
```

If you get the answer 1.8288, congratulations, it worked. To view what units it can convert between, view the data file in the share directory (the list is quite comprehensive). To read the full documentation, change into the **info** directory and type

```
info --file=units.info
```

## 8.7 Stripping unnecessary code

When a piece of software is being developed, it is useful for the programmer to include debugging information into the resulting executable. This way, if there are problems encountered when running the executable, the programmer can load the executable into a debugging software package and track down any software bugs.

This is useful for the programmer, but unnecessary for the user. We can assume that the package, once finished and available for download has already been tested and debugged. However, when we compiled the software above, debugging information was still compiled into the final executable. Since it is unlikely that we are going to need this debugging information, we can strip it out of the final executable. One of the advantages of this is a much smaller executable, which should run slightly faster.

What we are going to do is look at the before and after size of the binary file. First change into the **bin** directory of the *units* installation directory.

```
cd ~/units174/bin
ls -l
```

As you can see, the file is over 100kB in size. You can get more information on the type of file by using the *file* command.

```
file units
units: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), for GNU/Linux
2.6.4, dynamically linked (uses shared libs), not stripped
```

To strip all the debug and line numbering information out of the binary file, use the strip command

```
strip units
ls -l
```

As you can see, the file is now 45 kB – less than half its original size. Half of the binary file was debug code!  Check the file information again.

```
file units
units: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), for GNU/Linux
2.6.4, dynamically linked (uses shared libs), stripped
```

Notice now that the file is **stripped**.

# Tutorial Nine

## Automation of commands using shell scripts

Once you have become used to basic commands and finding your way around a Linux system you may want to automate tasks which are executed frequently. This can be done with shell scripts – lists of commands to run in sequence. This is a simple but powerful form of programming which with loops and conditional statements allows complex sequences of commands to be run, saving much time.

## 9.1 Basic shell script

Create and edit a file called *test.sh*:

```
#!/bin/bash

echo “Hello $USER”
```

The line at the beginning is called a shebang, sha-bang, hashbang, pound-bang or hash-pling and tells the computer that the script should be run in a bash shell. The script knows that everything beginning with a dollar sign ($) is a variable and will use the value of the environment variable $USER (see section 7.2)

Once you have finished editing the file, close the editor and make the file executable with ‘chmod’, which you used in section 5.2

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

**Before you run the script, what do you think the output will be?**

Now run the script:

```
./test.sh
```

## 9.2 Using variables

As well as using environment variables, new variables can created and used within your script. The example below uses a new variable, NUMLOGINS.

Note that you do not need to use the dollar sign when you create the variable, only to refer to it once it has been created. The $( *command* ) construct is used in this instance to pass the output of a command or commands and assign the value to the new variable.

```
#!/bin/bash

echo “Hello $USER”
NUMLOGINS=$(who | grep $USER | wc -l)
echo “You have $NUMLOGINS login sessions”
```

## 9.3 Looping with ‘for’

Rather than simply process a sequence of commands one after the other, loops can perform the same commands for different variables. The example below uses the variable *$user* which is set up at the beginning of the ‘for’ loop. Note that this is not the same as the $USER environment variable as variable names are case sensitive.

```
#!/bin/bash

for user in gg78 ljg2 root
do
  NUMLOGINS=$(who | grep $user | wc –l)
  echo “$user has $NUMLOGINS sessions”
done
```

The script loops over the section between ‘do’ and ‘done’ once for each value of $user specified by the line beginning with ‘for’.

Numbers can be used instead of variables in the loop so that the script can keep track of which number loop it is on:

```
#!/bin/bash

for i in 1 2 3 4 5
do
  echo “loop $i”
done
```

If the script needs to loop a lot of times, this notation can be used:

```
for i in {1..20}
```

This will loop 20 times with the value of $i incrementing by 1 each time. If incrementing by a value other than 1 is required then use the ‘seq’ command, eg:

```
for i in $(seq 0 0.5 4)
```

This will give values for $i from 0 to 4, incrementing by 0.5

## 9.4 Conditional statements

"if / then / else" statements allow conditional branching within a script. This script tests the value of a variable and then decides which branch to take (and therefore what to output), depending on that value.

```
#!/bin/bash

NUMLOGINS=$(who | grep $USER | wc -l)

if [ $NUMLOGINS -gt 1 ]
then
   echo "$USER is logged in with at least 2 sessions"
else
   echo "$USER has less than 2 sessions"
fi
```

The 'if … then … else' statement must be finished with 'fi'. The 'else' option is optional – the script will carry on past 'fi' if you do not use it.

'-gt' is short for 'greater than'. Other tests include:

- -eq is equal to
- -ne is *not* equal to
- -ge is greater than or equal to
- -lt is less than
- -le is less than or equal to

### Exercise

Write a script which will test how many logins you and the 'root' user have and tell you who has the most (or if equal)