

## 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 <i>directory</i>	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 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 through 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

<code>cp file1 file2</code>	copy file1 and call it file2
<code>mv file1 file2</code>	move or rename file1 to file2
<code>rm file</code>	remove a file
<code>rmdir directory</code>	remove a directory
<code>cat file</code>	Display or concatenate a file
<code>less file</code>	display a file a page at a time
<code>head file</code>	display the first few lines of a file
<code>tail file</code>	display the last few lines of a file
<code>grep 'keyword' file</code>	search a file for keywords
<code>wc file</code>	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 (concatenate) 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 reads input from STDIN (the keyboard) and produces an alphabetically or numerically sorted 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

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

<i>who</i>	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 give 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

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



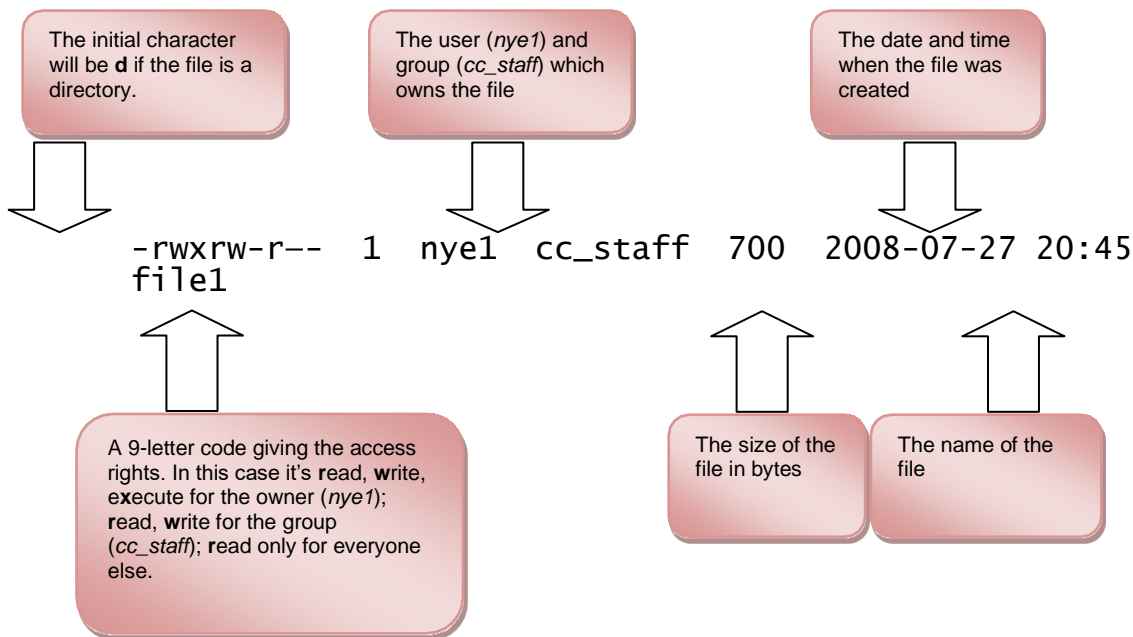
## Tutorial Five

### 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
```

## 5.1 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

commandprompt 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 jobnumber and PID, is returned by the machine. The user is 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 Controlkey 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*.

## 5.3 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.

## 5.4 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 refuse 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 to 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 logout. 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.