Unix Tutorial

Release 1.0

Simon Mutch and Greg Poole

February 17, 2013

CONTENTS

1 Basic Unix **3**
- 1.1 What is Unix? . . . 3
- 1.2 Part 1 . . . 4
- 1.3 Part 2 . . . 7
- 1.4 Part 3 . . . 11
- 1.5 Part 4 . . . 14

2 Advanced Unix **17**
- 2.1 Some data to work with . . . 17
- 2.2 Part 1 . . . 17
- 2.3 Part 2 . . . 21
- 2.4 Part 3 . . . 23
- 2.5 Part 4 . . . 24

These are the course notes for the **Basic and Advanced Unix** sessions of the 2013 ASA/ANITA Astroinformatics Summer School.

For an idea of what is included in each of the **basic** and **advanced** sections, see the contents listings below. A pdf version of this tutorial can be found here.

These notes are a shameless (in some places barely altered) reworking of the Unix Tutorial by M. Stonebank (2001; The University of Surrey).

As such, this work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

CHAPTER

ONE

BASIC UNIX

1.1 What is Unix?

Unix is an operating system. Operating systems control the computer and are responsible for opening files, dealing with the memory, allocating tasks to the processor and creating the underlying interface between the user and the machine.

Unix is made up of three basic components: the *kernel*, the *shell* and the *files and processes*.

1.1.1 The Kernel

The kernel is the underlying core of the computer and carries out all of the basic tasks including:

- The allocation of hardware and resources
- File system access
- Network access
- Execution and management of other programs
- Management of output (display and sound)
- Management of input (keystrokes, mouse clicks etc.)

1.1.2 The Shell

The shell, or command-line interpreter (CLI), is a program which provides the user interface to the kernel. The user can type commands into the shell which are then executed. Example commands include:

- Opening files
- Mounting external drives
- Starting programs

Commands can be run either in the foreground (the shell waits until their completion before accepting new commands) or in the background (the user may continue to issue new commands while the original one is still running).

There are a number of available Unix shells. The most common and earliest is the Bourne SHell (*sh*) and is found on almost every Unix system. Others include:

- The C SHell (csh) and associated TC SHell (tcsh)
- The Bourne Again SHell (bash) - *the default on Mac OS X*
- The Z SHell (zsh) - *my favorite!*

Warning: Most commands work identically on all shells, however, the syntax of configuration and batch scripts often varies which can make it difficult to change between different shell incarnations once you have set up one to your liking.

1.1.3 Files and Processes

Everything in Unix is either a file or a process.

A **process** is an executing program identified by a unique PID (process identifier).

A **file** is a collection of data. They are created by users using text editors, running compilers etc.

Examples of files include:

- a document (report, essay etc.)
- the text of a program written in some high-level programming language
- instructions comprehensible directly to the machine and incomprehensible to a casual user, for example, a collection of binary digits (an executable or binary file);
- a directory, containing information about its contents, which may be a mixture of other directories (subdirectories) and ordinary files.

1.2 Part 1

1.2.1 Listing files and directories

`ls` (list)

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

To find out what is in your home directory, type:

```
% ls
```

The `ls` (short for list) command lists the contents of your current working directory.

There may be no files visible in your home directory, in which case, the Unix prompt will be returned. Alternatively, there may already be some files created by the operating system or your System Administrator when your account was created.

`ls` does not, in fact, 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 generally not change them unless you are familiar with Unix.

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 *flags*: `-a` is an example. Flags change the behaviour of a command. There are manual pages that tell you which flags a particular command can take, and how each one modifies the behaviour of the command (see the *getting help* section).

1.2.2 Making Directories

`mkdir` (make directory)

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

```
% mkdir unix_tutorial
```

Here `unix_tutorial` is an *argument* to the `mkdir` command. Some commands, such as `mkdir` require an argument at all times (in this case, the name of the directory we want to create). Other commands, such as `ls` can take arguments, but do not necessarily require them.

To see the directory you have just created, type:

```
% ls
```

Notice the lack of an argument for the `ls` command here. This means "list the contents of my current working directory". If were to instead type:

```
% ls unix_tutorial
```

we would be presented with the contents of our newly created directory (which is currently empty though!).

1.2.3 Changing to a different directory

`cd` (change directory)

The command `cd` changes the current working directory to another one you specify. The current working directory is the directory you are currently in - your current position in the file-system tree.

To change to the directory you have just made, type:

```
% cd unix_tutorial
```

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

Exercise 1a

Make another directory inside the `unix_tutorial` directory called `backups`.

1.2.4 The directories `.` and `..`

Still in the `unix_tutorial` directory, type:

```
% ls -a
```

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

In UNIX, `.` means the current directory, so typing:

```
% cd .
```

means stay where you are (the `unix_tutorial` directory).

Note: The space between `cd` and the dot is necessary. `cd` is the command and `.` an argument which we are passing to `cd`.

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.

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

1.2.5 Pathnames

`pwd` (print working directory)

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
```

The full pathname **may** look something like this:

```
/Users/myname
```

which means that `myname` (your home directory) is in the directory `Users`, which itself is located at the root of file-system.

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)

1.2.6 More about home directories and pathnames

Understanding pathnames

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

```
% ls unix_tutorial
```

to list the contents of your `unix_tutorial` directory.

Now type:

```
% ls backups
```

You will get a message like this:

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

The reason is, `backups` is not in your current working directory. To use a command on a file (or directory) not in the current working directory (the directory you are currently in), you must either `cd` to the correct directory, or specify its full pathname. Therefore, to list the contents of your backups directory, you must type:

```
% ls unix_tutorial/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 ~/unix_tutorial
```

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

1.2.7 Summary of commands

Command	Description
`ls`	list files and directories
`ls -a`	list all files and directories (including hidden)
`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 of current location
`pwd`	display the full path of the current directory

1.3 Part 2

1.3.1 Copying Files

cp (copy)

The command `cp file1 file2` will make a copy of `file1` in the current working directory with the name `file2`.

What we are going to do now is download a file from the internet, save it to our `unix_tutorial` directory and then use the `cp` command to copy that file to our `backups` directory (which we created in *Exercise 1a*).

First point your web browser to this link (my_first_file.txt) and save the file using "*Save as..*" to your `unix_tutorial` directory.

Now, `cd` to your `unix_tutorial` directory.

```
% cd ~/unix_tutorial
```

Then at the Unix prompt, type:

```
% cp my_first_file.txt my_first_file.bu
```

This command means copy the file `my_first_file.txt` to `my_first_file.bu`.

> **Warning:** The `cp` command allows you to use already existing filenames as the target for the copy, hence it can be used to overwrite files!

1.3.2 Moving files

mv (move)

The command `mv file1 file2` moves (or renames) `file1` to `file2`.

To move a file from one place to another, use the `mv` command. This has the effect of moving rather than copying the file, so you end up with only one file rather than two.

It can also be used to rename a file, by moving the file to the same directory, but giving it a different name.

We are now going to move the file `my_first_file.bu` to your backup directory.

First, change directories to your `unix_tutorial` directory (using the `cd` command). Then, inside the `unix_tutorial` directory, type:

```
% mv my_first_file.bu backups/
```

Type `ls` and `ls backups` to see if it has worked.

1.3.3 Removing files and directories

`rm` (remove), `rmdir` (remove directory)

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

Inside your `unix_tutorial` directory, type:

```
% cp my_first_file.txt tempfile.txt
% ls
```

You should now see both `my_first_file.txt` and `tempfile.txt` listed. Next type:

```
% rm tempfile.txt
% ls
```

and the file `tempfile.txt` should be gone!

You can use the `rmdir` command to remove an **empty** directory. Try to remove the `backups` directory. You will not be able to since Unix will not let you remove a non-empty directory.

Exercise 2a

Create a directory called `Tempstuff` using mkdir , then remove it using the rmdir command.

1.3.4 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 my_first_file.txt
```

Unfortunately, the file is longer than than the size of the window, so it scrolls past. On many modern terminals we can just scroll up to see what we have missed, however, on older systems this may not be possible!

less

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

```
% less my_first_file.txt
```

Press the `[space-bar]` if you want to see another page, of press `q` if you want to quit. Typically, `less` is a better option for reading a long file than `cat`. Other useful keys in `less` include:

Key	Action
`g`	Return to top of file
`G`	Scroll to bottom of file
`[up]`	Scroll up one line
`[down]`	Scroll down one line

head

The `head` command writes the first ten lines of a file to the screen.

First clear the screen then type:

```
% head my_first_file.txt
```

Then type:

```
% head -5 my_first_file.txt
```

What difference did the `-5` flag 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 my_first_file.txt
```

Exercise 2b

Try using the tail command to view the last 15 lines of `my_first_file.txt`.

1.3.5 Searching the contents of a file

Simple searching using `less`

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

```
% less my_first_file.txt
```

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

```
/science
```

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

grep (don't ask why it is called 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 my_first_file.txt
```

and grep will display each line containing the word "science"... *Or has it?*

Try:

```
% grep Science my_first_file.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.

```
% grep -i science my_first_file.txt
```

To search for a phrase or pattern, you must enclose it in single quotes (‘). For example to search for "black holes"

```
% grep -i 'black holes' my_first_file.txt
```

Some other useful grep include:

Option	Result
-n	precede each matching line with the line number
-v	display those lines that do **not** match
-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 my_first_file.txt
```

wc (word count)

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

```
% wc -w my_first_file.txt
```

To find out how many lines the file has, type:

```
% wc -l my_first_file.txt
```

1.3.6 Summary of commands

Command	Description
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 a file
more 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

1.4 Part 3

1.4.1 Redirection

Most processes initiated by Unix 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, which is by default again the terminal screen.

We have already seen one use of the `cat` command to write the contents of a file to the screen.

Now type `cat` without specifying a file to read from standard input:

```
% cat
```

Then type a few words on the keyboard and press the `Return` key.

Finally hold the `Ctrl` key down and press `d` (written as `^D` for short) to end the input.

What has happened?

If you run the cat command without specifying a file to read, it reads the standard input (the keyboard), and on receiving the 'end of file' (`^D`), copies it to the standard output (the screen).

In Unix, we can redirect both the input and the output of commands.

1.4.2 Redirecting the Output

We use the > symbol to redirect the output of a command. For example, to create a file called `list1.txt` containing a list of fruit, type:

```
% cat > list1.txt
```

Then type in the names of some fruit. Press `Return` after each one.

```
pear
banana
apple
```

To stop press `^D` (control D).

What happens is the cat command reads the standard input (the keyboard) and the > redirects the output, which normally goes to the screen, into a file called `list1.txt`.

To read the contents of the file, type:

```
% cat list1.txt
```

Exercise 3a

Using the above method, create another file called `list2.txt` containing the following fruit: orange, plum, mango, grapefruit. Read the contents of `list2.txt`.

The form >> appends standard output to a file. So to add more items to the file `list1.txt`, type:

```
% cat >> list1.txt
```

Then type in the names of more fruit:

```
peach
grape
orange
```

and then ^D (control D) to stop.

To read the contents of the file, type:

```
% cat list1.txt
```

You should now have two files. One contains six fruit, the other contains four fruit. We will now use the cat command to join (concatenate) list1.txt and list2.txt into a new file called biglist.txt. Type:

```
% cat list1.txt list2.txt > biglist.txt
```

What this is doing is reading the contents of list1.txt and list2.txt in turn, then outputting the text to the file biglist.txt.

To read the contents of the new file, type:

```
% cat biglist.txt
```

Piping the output of a command to a file can be extremely useful. For example, imagine that we are running a large script which carries out some complex manipulation of our data, taking a long time to complete. Typically we would setup our script so that it prints out status messages telling us that certain tasks have been completed or certain files have been read/written. By piping the output of our script to a file we can record these log messages for future reference or for situations where we may want to leave the script running after we have logged out (and hence there will be no standard output).

1.4.3 Redirecting the Input

We use the < symbol to redirect the input of a command.

The command sort alphabetically or numerically sorts a list. Type:

```
% sort
```

Then type in the names of some vegetables. Press Return after each one.

```
carrot
beetroot
artichoke
^D
```

The output will be:

```
artichoke
beetroot
carrot
```

Using < you can redirect the input to come from a file rather than the keyboard. For example, to sort the list of fruit, type:

```
% sort < biglist.txt
```

and the sorted list will be output to the screen.

To output the sorted list to a file, type:

```
% sort < biglist.txt > sorted_list.txt
```

Use cat to read the contents of the file sorted_list.txt.

1.4.4 Pipes

To see who is on the system with you, type:

```
% who
```

Note: If you are the only person currently logged into the system then try `who -a` and use this in replace of `who` below.

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 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 (`|`). Pipes are one of the most useful features of Unix...

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
```

Pipes can be used to string together multiple commands, making them extremely powerful. For example, in order to find out out many **other** users (i.e. excluding ourselves) are logged onto the system we could use the following:

```
% who | grep -v [username] | wc -l
```

where `[username]` should be replaced by your own user name.

What we have done here is run the `who` command to find out who is logged into the system. The output has then been piped to the command `grep` which has **removed** (as a result of the `-v` flag) all lines containing our username. Finally, this output has been piped to `wc` which has counted the number of lines.

Exercise 3b

Create a file called `current_users.txt`, that holds a **sorted** list of all *other* users (i.e. excluding yourself) that are currently logged in to the system.

1.4.5 Summary of commands

Command	Description
`command > file`	redirect standard output 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

1.5 Part 4

1.5.1 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 `unix_tutorial` 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
```

1.5.2 Filename conventions

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.

It is important to recognise that a directory is merely a special type of file. So the rules and conventions for naming files apply also to directories.

1.5.3 Getting Help

On-line Manuals

There are inbuilt manuals which gives information about most commands. The manual pages tell you which flags a particular command can take, and how each flag modifies the behaviour of the command. Use the `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 flags etc.

apropos

When you are not sure of the exact name of a command you can try using the `apropos` command:

```
% apropos keyword
```

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

```
% apropos copy
```

1.5.4 Summary of commands

Command	Description
`*`	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

CHAPTER

TWO

ADVANCED UNIX

2.1 Some data to work with

In this section we are going to work with a couple of data files. You can get them from these links:

- data_file_1.txt
- data_file_2.txt

2.2 Part 1

2.2.1 File system security (access rights)

In your unixstuff directory, type:

```
% ls -l
```

where `-l` produces a long listing. You will see that you now get lots of details about the contents of your directory.

Each file (and directory) has associated access rights, which may be found by typing `ls -l`. Also, `ls -lg` gives additional information as to which group owns the file (beng95 in the following example):

```
-rwxrw-r-- 1 ee51ab beng95 2450 Sept29 11:52 file1
```

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 be the starting symbol of the string.

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) (`ee51ab` in the above example);
- the middle group gives the permissions for the group of people to whom the file (or directory) belongs (`eebeng95` in the above example);
- the rightmost group gives the permissions for all others.

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 (e.g. your mailbox file).

2.2.2 Changing access rights

chmod (changing a file mode)

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

Create a new directory with the command:

```
% mkdir science_dir
```

and a new file within that directory with the command:

```
% touch science_dir/science.txt
```

Try changing access permissions on the directory using:

```
% chmod a-x science_dir
```

Use `ls -l` to check that the permissions have changed.

See if you can see the file now:

```
% ls science_dir
```

Fix the permissions and then delete the directory and its contents using:

```
% rm -rf science_dir
```

2.2.3 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 do 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 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**). To put it in the background, type:

```
% bg
```

Note: do not background programs that require user interaction e.g. `pine`.

2.2.4 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 chrome
[3] Running vi
```

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.

2.2.5 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 &
[4] 3706

% 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 !!!

2.2.6 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

2.3 Part 2

2.3.1 Other useful Unix commands

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 disk your current directory is on, type:

```
% df .
```

To get the results in "human-readable" format, use the `-h` option:

```
% df -h .
```

du

The `du` command outputs the number of kilobyes used by each subdirectory. Useful if you have gone over quota and you want to find out which directory has the most files. In your home-directory, type:

```
% du
```

Again, to get the results in "human-readable" format, use the `-h` option:

```
% du -h
```

compressing files

This reduces the size of a file, thus freeing valuable disk space. For example, type:

```
% ls -l science.txt
```

and note the size of the file. Then to compress **science.txt**, type:

```
% compress science.txt
```

This will compress the file and place it in a file called `science.txt.Z`.

To see the change in size, type `ls -l` again.

To uncompress the file, use the `uncompress` command.

```
% uncompress science.txt.Z
```

A better method of compression is to use `gzip`. This also compresses a file, but is more efficient than compress. For example, to zip **science.txt**, type:

```
% gzip science.txt
```

This will zip the file and place it in a file called **science.txt.gz**.

To unzip the file, use the `gunzip` command.

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

Lastly, you can package a group of files together using `tar`. Try this by making a few files:

```
% man gcc > gcc.txt
% man tar > tar.txt
% man man > man.txt
```

and package them together using `tar`:

```
% tar cf man_pages.tar *.txt
```

You can look at the contents of your tar file using:

```
% tar tf man_pages.tar
```

You can also have `tar` compress the result on the fly for you:

```
% tar cfz man_pages.tgz *.txt
```

Compare the sizes of the two resulting files using `ls -l`.

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, type:

```
% file ~/*
```

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. To show this list type:

```
% history
```

If you are using the C shell, 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:

```
% set history=100
```

2.4 Part 3

2.4.1 Simple calculations in Unix using awk

awk is a powerful Unix command useful for performing line-by-line operations on ascii files (eg. tables of data). It can be used in two modes: command-line-mode and as a scripting language. To get an idea of how it works, let's look at the command-line usage first. Later we will create some simple awk scripts.

awk on the command line

The general format of a call to awk is as follows:

```
% awk 'PROGRAM' file
```

Where PROGRAM denotes what will be executed on each line and file is the file to be operated-on in a line-by-line fashion. The simplest program just prints the contents of the file:

```
% awk 'print' file
```

awk parses each line into a series of columns. Each column is referred to as $1, $2, $3, etc. The entire line is specified as $0. So, another way to print the whole file is:

```
% awk 'print $0' file
```

If you want to just print the first and 3rd columns though, you can do that with:

```
% awk 'print $1,$3' file
```

The real power of awk emerges when you start using it's pattern-matching and column operation functionality. For instance, assume you want to find all the files in your how directory owned by user user_name. You could run:

```
% ls -l | awk '/user_name/ {print $9}'
```

Here, /user_name/ is a match filter. For every line that contains the text between the /'s, the code between the {}'s will be executed. You can have more than one match filter:

```
% man ls | awk '/-A/ {print $0} /-f/ {print $0}'
```

Working with our sample data files, say we want to find all the galaxies marked as QSOs in the first file:

```
% awk '/QSO/ {print $0}' data_file_1.txt
```

Perhaps we want to keep the header information at the top:

```
% awk '/#/ {print $0} /QSO/ {print $0}' data_file_1.txt
```

awk allows for code to be executed before-and-after the line-by-line processing:

```
% awk 'BEGIN {print "# Below is a list of QSOs"} /#/ {print $0} /QSO/ {print $0} END{print "# Abo
```

awk can operate on multiple files:

```
% awk 'BEGIN {print "# Below is a list of QSOs"} /#/ {print $0} /QSO/ {print $0} END{print "# Abo
```

Now, what if we only want the first header line? Let's stretch awk's abilities a bit more:

```
% awk 'BEGIN {print "# Below is a list of QSOs";i=0} /#/ {i=i+1;if(i==1) print $0} /QSO/ {print $
```

Let's make a list of all the R-band magnitudes in the first file:

```
% awk 'BEGIN {print "# Below is a list of galaxies";i=0} /#/ {i=i+1;if(i==4) print "# "$6} ! /#/{
```

Suppose there's a calibration error and we want to remove 0.01 from all magnitudes less than 21. awk allows us to perform conditionals and mathematical expressions:

```
% awk 'BEGIN {print "# Below is a list of galaxies";i=0} /#/ {i=i+1;if(i==4) print "# "$6} ! /#/{
```

awk as a scripting language

awk can be used as a scripting language much like a shell script. Start the script with a line declaring that the file will run awk in **filter** mode. Turning our last example into a script you would have:

```
#!/usr/bin/awk -f
BEGIN {
    print "# Below is a list of galaxies";
    i=0
}

/#/ {
    i=i+1;
    if(i==4) print "# "$6
}

! /#/{
    if($9<21.) print $9-0.01;
    else       print $9
}
```

Experiment with awk it has a lot more functionality than what is shown here and there are many good tutorials online.

2.5 Part 4

2.5.1 Unix 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.

Standard Unix variables are split into two categories, environment variables and shell variables. In broad terms, shell variables apply only to the current instance of the shell and are used to set short-term working conditions; environment variables have a farther reaching significance, and those set at login are valid for the duration of the session. By convention, environment variables have UPPER CASE and shell variables have lower case names.

2.5.2 Environment Variables

An example of an environment variable is the OSTYPE variable. The value of this is the current operating system you are using. Type:

```
% echo $OSTYPE
```

More examples of environment variables are:

- USER (your login name)
- HOME (the path name of your home directory)
- HOST (the name of the computer you are using)
- ARCH (the architecture of the computers processor)
- DISPLAY (the name of the computer screen to display X windows)
- PRINTER (the default printer to send print jobs)
- PATH (the directories the shell should search to find a command)

Finding out the current values of these variables.

ENVIRONMENT variables are set using the `setenv` command, displayed using the `printenv` or `env` commands, and unset using the `unsetenv` command.

To show all values of these variables, type:

```
% printenv | less
```

2.5.3 Shell Variables

An example of a shell variable is the history variable. The value of this is how many shell commands to save, allow the user to scroll back through all the commands they have previously entered. Type:

```
% echo $history
```

More examples of shell variables are:

- `cwd` (your current working directory)
- `home` (the path name of your home directory)
- `path` (the directories the shell should search to find a command)
- `prompt` (the text string used to prompt for interactive commands shell your login shell)

Finding out the current values of these variables.

SHELL variables are both set and displayed using the set command. They can be unset by using the unset command.

To show all values of these variables, type:

```
% set | less
```

So what is the difference between PATH and path ?

In general, environment and shell variables that have the same name (apart from the case) are distinct and independent, except for possibly having the same initial values. There are, however, exceptions.

Each time the shell variables home, user and term are changed, the corresponding environment variables HOME, USER and TERM receive the same values. However, altering the environment variables has no effect on the corresponding shell variables.

PATH and path specify directories to search for commands and programs. Both variables always represent the same directory list, and altering either automatically causes the other to be changed.

2.5.4 Using and setting variables

Each time you login to a Unix host, the system looks in your home directory for initialisation files. Information in these files is used to set up your working environment. The C and TC shells uses two files called .login and .cshrc (note that both file names begin with a dot).

At login the C shell first reads **.cshrc** followed by **.login**

.login is to set conditions which will apply to the whole session and to perform actions that are relevant only at login.

.cshrc is used to set conditions and perform actions specific to the shell and to each invocation of it.

The guidelines are to set ENVIRONMENT variables in the **.login** file and SHELL variables in the **.cshrc** file.

WARNING: NEVER put commands that run graphical displays (e.g. a web browser) in your .cshrc or .login file.

2.5.5 Setting shell variables in the .cshrc file

For example, to change the number of shell commands saved in the history list, you need to set the shell variable history. It is set to 100 by default, but you can increase this if you wish.

```
% set history = 200
```

Check this has worked by typing:

```
% echo $history
```

However, this has only set the variable for the lifetime of the current shell. If you open a new xterm window, it will only have the default history value set. To PERMANENTLY set the value of history, you will need to add the set command to the .cshrc file.

First open the **.cshrc** file in a text editor. An easy, user-friendly editor to use is nano, but other ones include emacs or nedit.

```
% nano ~/.cshrc
```

Add the following line AFTER the list of other commands.

```
set history = 200
```

Save the file and force the shell to reread its .cshrc file buy using the shell source command.

```
% source .cshrc
```

Check this has worked by typing:

```
% echo $history
```

2.5.6 Setting the path

When you type a command, your path (or PATH) variable defines in which directories the shell will look to find the command you typed. 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.

Let's create an executable in a directory called `test_path`:

```
% mkdir test_path
% echo #!\/bin\/csh    > test_path/siesta
% echo sleep 5        >> test_path/siesta
% chmod a+x test_path/siesta
```

If you explicitly give the full path to the program, it will execute 'sleep' for 5 seconds:

```
% ./test_path/siesta
```

But the computer doesn't know where it is otherwise:

```
% siesta
siesta: Command not found.
```

For this command to work anywhere without giving the explicit location every time, you need to add it to the end of your existing path (the **$path** represents this) by issuing the command:

```
% set path = ($path $PWD/test_path)
```

Test that this worked by trying to run `siesta` in any directory other that where it is actually located.

```
% cd; siesta
```

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

To add a path PERMANENTLY, place the following line at the end of your **.cshrc** file:

```
set path = ($path directory_to_the_command)
```