# Cognixia

A Collabera **LEARNING SOLUTIONS COMPANY**

Cognixia:
# Unix Prerequisite

# UNIX Introduction

## What is UNIX?

UNIX is an operating system which was first developed in the 1960s, and has been under constant development ever since. By operating system, we mean the suite of programs which make the computer work. It is a stable, multi-user, multi-tasking system for servers, desktops and laptops.

UNIX systems also have a graphical user interface (GUI) similar to Microsof Windows which provides an easy to use environment. However, knowledge of UNIX is required for operations which aren't covered by a graphical program, or for when there is no windows interface available, for example, in a telnet session.

## Types of UNIX

There are many different versions of UNIX, although they share common similarities. The most popular varieties of UNIX are Sun Solaris, GNU/Linux, and MacOS X.

Here in the School, we use Solaris on our servers and workstations, and Fedora Linux on the servers and desktop PCs.

## The UNIX operating ystem

The UNIX operating system is made up of three parts; the kernel, the shell and the programs.

### The kernel

The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs and handles the filestore and communications in response to system calls.

As an illustration of the way that the shell and the kernel work together, suppose a user types **rmmyfile** (which has the effect of removing the file **myfile**). The shell searches the filestore for the file containing the program **rm**, and then requests the kernel, through

system calls, to execute the program `rm` on **myfile**. When the process `rmmyfile` has finished running, the shell then returns the UNIX prompt % to the user, indicating that it is waiting for further commands.

## The shell

The shell acts as an interface between the user and the kernel. When a user logs in, the login program checks the username and password, and then starts another program called the shell. The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out. The commands are themselves programs: when they terminate, the shell gives the user another prompt (% on our systems).

The adept user can customise his/her own shell, and users can use different shells on the same machine. Staff and students in the school have the **tcsh shell** by default.

The tcsh shell has certain features to help the user inputting commands.

Filename Completion - By typing part of the name of a command, filename or directory and pressing the [**Tab**] key, the tcsh shell will complete the rest of the name automatically. If the shell finds more than one name beginning with those letters you have typed, it will beep, prompting you to type a few more letters before pressing the tab key again.

History - The shell keeps a list of the commands you have typed in. If you need to repeat a command, use the cursor keys to scroll up and down the list or type history for a list of previous commands.

## 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:

- 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.

## The Directory Structure

All the files are grouped together in the directory structure. The file-system is arranged in a hierarchical structure, like an inverted tree. The top  of  the  hierarchy  is  traditionally called **root** (written as a slash / )
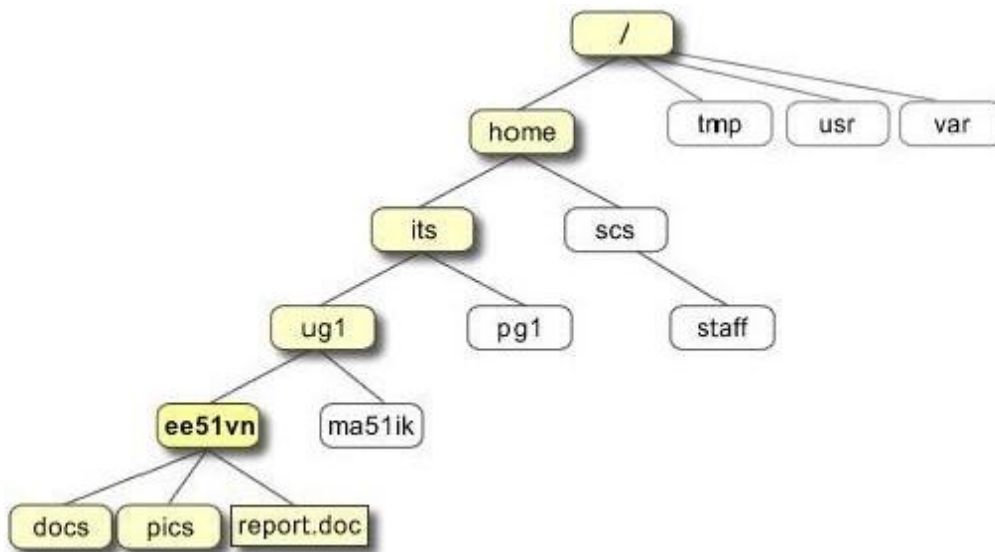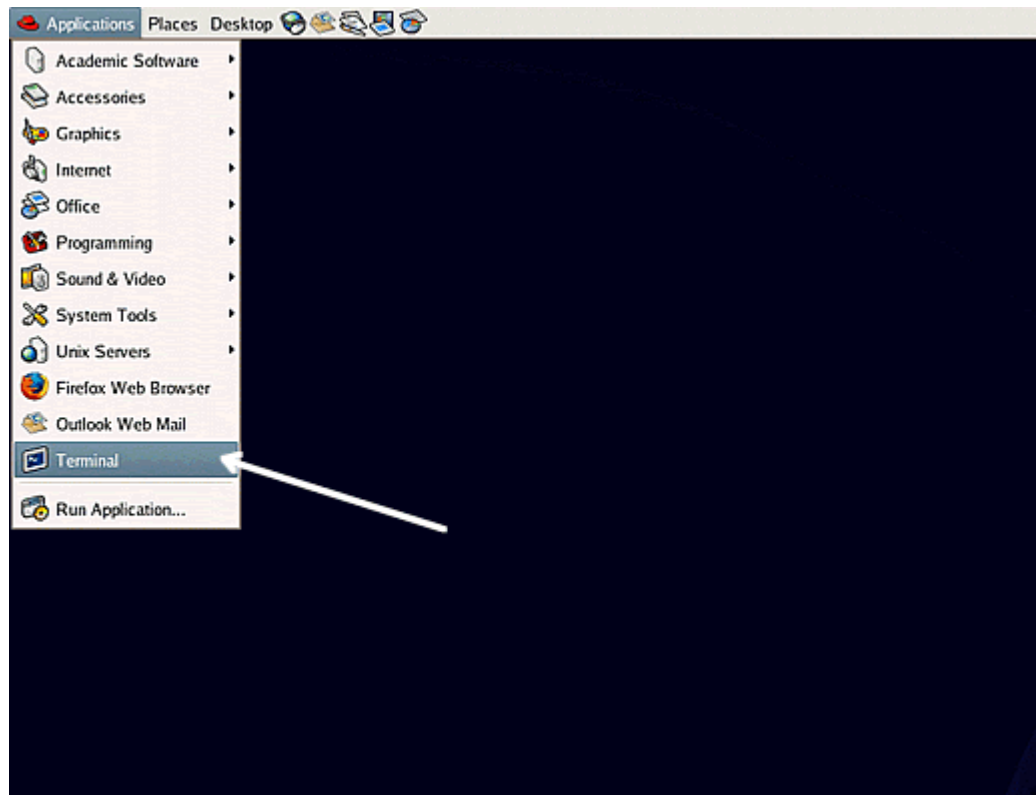


In  the  diagram  above,   we  see  that  the  home   directory of        undergraduate student **"ee51vn"** contains        the two      sub-directories                (**docs** and   a   file called **report.doc**.              and **pics**)

The full path to the file **report.doc** is **"/home/its/ug1/ee51vn/report.doc"**

## Starting an UNIX terminal

To   open   an   UNIX   terminal   window,   click   on   the   "Terminal"   icon   from Applications/Accessories menus.

An UNIX Terminal window will then appear with a % prompt, waiting for you to start entering commands.
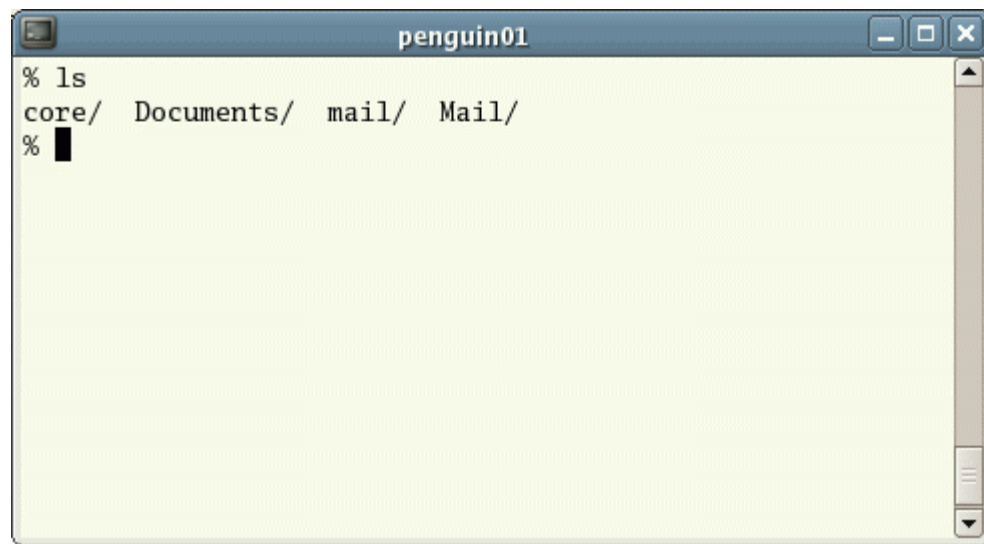
## 1.1 Listing files and directories

### ls (list)

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, **ee91ab**, 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 ( lowercase L and lowercase S ) lists the contents of your current working directory.
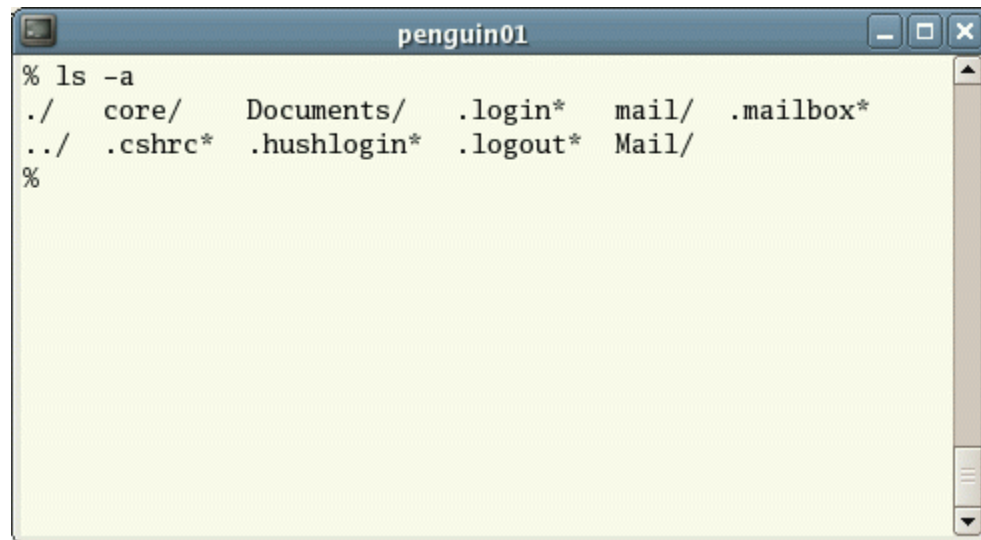


There may be no files visible in your home directory, in which case, the UNIX prompt will be returned. Alternatively, there may already be some files inserted by the System Administrator when your account was created.

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 not change them unless you are very familiar with UNIX!!!

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

```
% ls -a
```

As you can see, `ls -a` lists files that are normally hidden.

```
penguin01
% ls -a
./    core/    Documents/    .login*    mail/    .mailbox*
../   .cshrc*  .hushlogin*   .logout*   Mail/
%
```

`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 which options a particular command can take, and how each option modifies the behaviour of the command. (See later in this tutorial)

## 1.2 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 in the course of this tutorial. To make a subdirectory called unixstuff in your current working directory type

```
% mkdirunixstuff
```

To see the directory you have just created, type

```
% ls
```

## 1.3 Changing to a different directory

### cd (change directory)

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 (**..**)

### The current directory (.)

In UNIX, (**.**) means the current directory, so typing

```
% cd .
```

> NOTE: there is a space between cd and the dot

means stay where you are (the **unixstuff** directory).

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.

## The parent directory (..)

(**..**) 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.

Note: 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 (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 will look something like this -

**/home/its/ug1/ee51vn**

which means that **ee51vn** (your home directory) is in the sub-directory **ug1** (the group directory),which in turn is located in the **its** sub-directory, which is in the **home** sub-directory, which is in the top-level root directory called " / " .

/

home    tmp    usr    var

its    scs

ug1    pg1    staff

ee51vn    ma51ik

docs    pics    report.doc
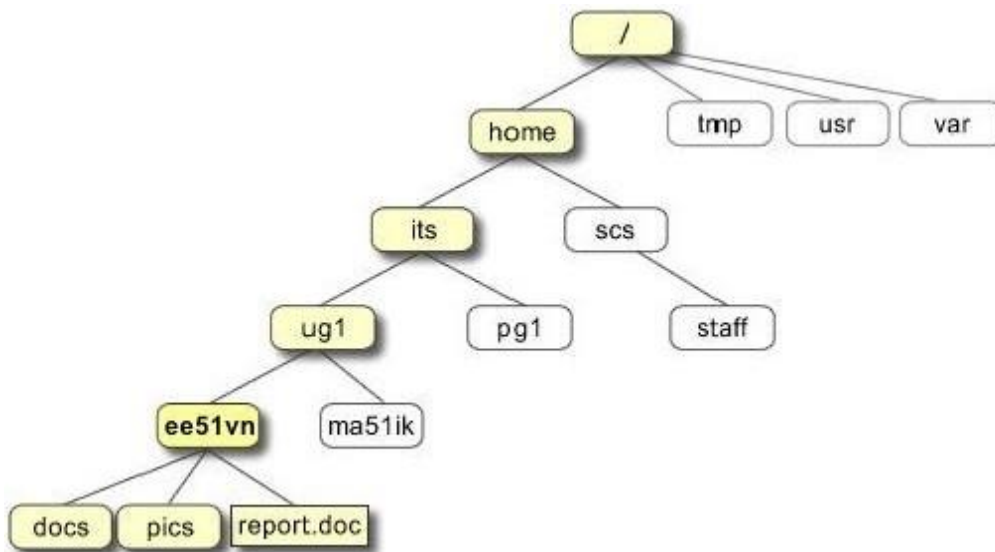
## Exercise 1b

Use the commands `cd`, `ls` and `pwd` 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

**% lsunixstuff**

to list the conents of your unixstuff 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. To list the contents of your backups directory, you must type

```
% lsunixstuff/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

```
% ls ~
```

would list?

What do you think

```
% ls ~/..
```

would list?

# Summary

| Command | Meaning |
|---|---|
| `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 |

## 2.1 Copying Files

## cp (copy)

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, cd to your **unixstuff** directory.

**% cd ~/unixstuff**

Then at the UNIX prompt, type,

**% cp /vol/examples/tutorial/science.txt .**

Note: 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.

(Note: The directory **/vol/examples/tutorial/** is an area to which everyone in the school has read and copy access. If you are from outside the University, you can grab a copy of the file here. Use 'File/Save As..' from the menu bar to save it into your **unixstuff** directory.)

## Exercise 2a

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

## 2.2 Moving files

### mv (move)

`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 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/.
```

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

## 2.3 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 **science.txt** file then delete it.

Inside your **unixstuff** directory, type

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

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 UNIX 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, and 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
```

Q. 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`, type a forward slash [**/**] followed by the word to search

```
/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 science.txt
```

As you can see, `grep` has printed out each line containg 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
```

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

## Summary

| Command | Meaning |
|---|---|
| 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 |
| 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 |

## 3.1 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, by default, to 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 specifing a file to read

```
% 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 specifing 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.

## 3.2 Redirecting the Output

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

```
% cat > list1
```

Then type in the names of some fruit. Press [**Return**] after each one.

```
pear
banana
apple
^D {this means press [Ctrl] and [d] to stop}
```

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**

To read the contents of the file, type

```
% cat list1
```

## Exercise 3a

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

## 3.2.1 Appending to a file

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

```
% cat >> list1
```

Then type in the names of more fruit

```
peach
grape
orange
^D (Control D to stop)
```

To read the contents of the file, type

```
% cat list1
```

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** and **list2** into a new file
called **biglist**. Type

```
% cat list1 list2 >biglist
```

What this is doing is reading the contents of **list1** and **list2** in turn, then outputing the text to the file **biglist**

To read the contents of the new file, type

```
% cat biglist
```

## 3.3 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 animals. Press [Return] after each one.

```
dog
cat
bird
ape
^D (control d to stop)
```

The output will be

```
ape
bird
cat
dog
```

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
```

and the sorted list will be output to the screen.

To output the sorted list to a file, type,

```
% sort <biglist>slist
```

Use cat to read the contents of the file **slist**

## 3.4 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 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 |

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
```

### Exercise 3b

Using pipes, display all lines of **list1** and **list2** containing the letter 'p', and sort the result.

[Answer available here](#)

## Summary

| Command | Meaning |
|---|---|
| *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`<br>*file2 > file0* | concatenate file1 and file2 to file0 |
| `sort` | sort data |
| `who` | list users currently logged in |

## 4.1 Wildcards

## The * wildcard

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 ?wildcard

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

**% ls ?list**

## 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).

| Good filenames | Bad filenames |
|---|---|
| project.txt | project |
| my_big_program.c | my big program.c |
| fred_dave.doc | fred& dave.doc |

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.

## Apropos

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

```
% apropos keyword
```

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

```
% apropos copy
```

# Summary

| Command | Meaning |
|---|---|
| * | 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 |

# 5.1 File system security (access rights)

In your **unixstuff** directory, type

```
% ls -l (l for long listing!)
```

You will see that you now get lots of details about the contents of your directory, similar to the example below.
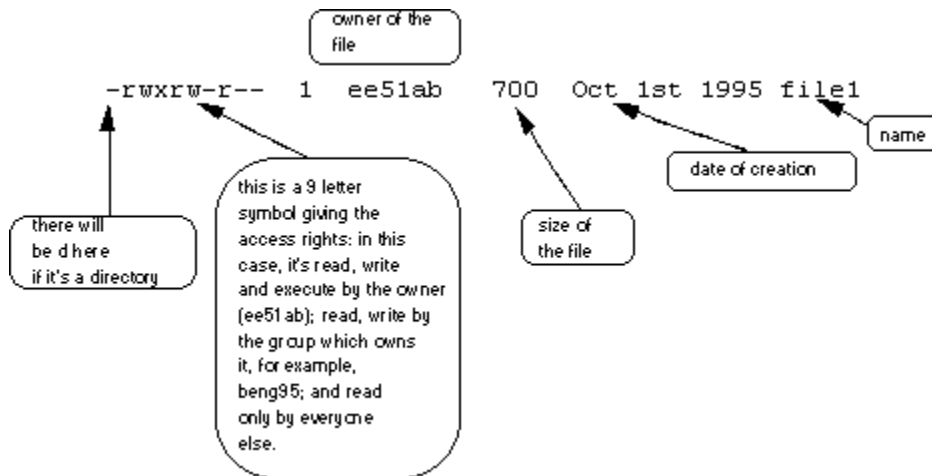
Each file (and directory) has associated access rights, which may be found by typing `ls -l`. Also, `ls -lg` gives additional information as to which group owns the file (beng95 in the following example):

```
-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). |

## 5.2 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-rwxbiglist
```

This will leave the other permissions unaffected.

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

```
% chmoda+rwbiglist
```

## Exercise 5a

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

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

## 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 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 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 1000
```

You can suspend the process running in the foreground by typing **^Z**, i.e.hold down the [**Ctrl**] key and type [**z**]. Then to put it in the background, type

```
% bg
```

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

## 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              1000
[2]                        Running                            netscape
[3] Running matlab
```

To restart (foreground) a suspended processes, type

```
% fg %jobnumber
```

For example, to restart sleep 1000, 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                         1000                          &
% ps

PID            TT            S            TIME            COMMAND
20077        pts/5        S            0:05            sleep            1000
21563         pts/5            T            0:00            netscape
21873 pts/5 S 0:25 nedit
```

To kill off the process **sleep 1000**, 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

| Command | Meaning |
|---|---|

| | |
|---|---|
| `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 |

# Other useful UNIX commands

## quota

All students are allocated a certain amount of disk space on the file system for their personal files, usually about 100Mb. If you go over your quota, you are given 7 days to remove excess files.

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

`% quota -v`

## 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 .`

## 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 -s *
```

The **-s** flag will display only a summary (total size) and the * means all files and directories.

## gzip

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 using `ls -l`. Then to compress science.txt, type

```
% gzip science.txt
```

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

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

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

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

## zcat

`zcat` will read gzipped files without needing to uncompress them first.

```
% zcat science.txt.gz
```

If the text scrolls too fast for you, pipe the output though `less`.

```
% zcat science.txt.gz | less
```

## 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 *
```

## diff

This command compares the contents of two files and displays the differences. Suppose you have a file called **file1** and you edit some part of it and save it as **file2**. To see the differences type

```
% diff file1 file2
```

Lines beginning with a **<** denotes file1, while lines beginning with a **>** denotes file2.

## find

This searches through the directories for files and directories with a given name, date, size, or any other attribute you care to specify. It is a simple command but with many options - you can read the manual by typing `man find`.

To search for all fies with the extention **.txt**, starting at the current directory (.) and working through all sub-directories, then printing the name of the file to the screen, type

```
% find . -name "*.txt" -print
```

To find files over 1Mb in size, and display the result as a long listing, type

```
% find . -size +1M -ls
```

## history

The C 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 (show command history list)
```

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
```

## 7.1 Compiling UNIX software packages

We have many public domain and commercial software packages installed on our systems, which are available to all users. However, students 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 lower-level language called assembly language. The assembly language code made by the previous stage is then converted into object code which are fragments of code which the computer understands directly. The final stage in compiling a program involves linking the object code to code libraries which contain certain built-in functions. This final stage produces an executable program.

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 being 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. You can usually use the **--help** option to get a list of interesting options for a particular configure script.

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.

## 7.2 Downloading source code

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

First create a download directory

```
% mkdir download
```

Download the software here and save it to your new download directory.

## 7.3 Extracting the source code

Go into your **download** directory and list the contents.

```
%                           cd                        download
% ls -l
```

As you can see, the filename ends in tar.gz. The tar command turns several files and directories into one single tar file. This is then compressed using the gzip command (to create a tar.gz file).

First unzip 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
```

Again, list the contents of the **download** directory, then go to the **units-1.74** sub-directory.

```
% cd units-1.74
```

## 7.4 Configuring and creating the Makefile

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
```

NOTE: The **$HOME** variable is an example of an environment variable. The value of **$HOME** is the path to your home directory. Just type

```
%                              echo                              $HOME
```

to show the contents of this variable. We will learn more about environment variables in a later chapter.

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.

## 7.5 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.

## 7.6 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
```

As an example, convert 6 feet to metres.

```
You                          have:                        6                          feet
You want: metres

* 1.8288
```

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
```

## 7.7 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 unlikey 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 100 kbytes in size. You can get more information on the type of file by using the file command.

```
% file units

units:  ELF  32-bit  LSB  executable,  Intel  80386,  version  1,
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 36 kbytes - a third of its original size. Two thirds of the binary file was debug code!!!

Check the file information again.

```
% file units

units:  ELF  32-bit  LSB  executable,  Intel  80386,  version  1,
dynamically linked (uses shared libs), stripped
```

Sometimes you can use the **make** command to install pre-stripped copies of all the binary files when you install the package. Instead of typing **make install**, simply type **make install-strip**

## 8.1 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.

## 8.2 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
```

## 8.3 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.

## 8.4 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.

## 8.5 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 nedit.

```
% nedit ~/.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
```

## 8.6 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.

For example, to run units, you either need to directly specify the units path (**~/units174/bin/units**), or you need to have the directory **~/units174/bin** in your path.

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

```
% set path = ($path ~/units174/bin)
```

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

```
%                                                         cd
% units
```

To add this path PERMANENTLY, add the following line to your .cshrc AFTER the list of other commands.

```
set path = ($path ~/units174/bin)
```