Introduction to Unix

Bernie Pope (bjpope@unimelb.edu.au)
Victorian Life Sciences Computation Initiative.

- Introduction
  - About this document
    - Audience and purpose
    - Typographic conventions
  - History and philosophy of Unix
- Login and logout
  - Remote login
    - SSH login from Windows
    - SSH login from Mac OS X and Linux
  - Logout
- The shell and the command line
  - An example command
  - Command history
  - Command editing
    - Tab completion
- Getting help
  - The Unix manual
  - Other sources of help
- The file system
  - The home directory
  - The working directory
  - File paths
    - Special file path names
  - Listing the contents of a directory
    - Hidden files
  - Changing the working directory
  - Making directories
  - Copying and moving (renaming) files
  - Removing files and directories
  - File permissions
    - Changing the permissions on a file
    - Changing the group of a file
    - How to keep your files private
  - File name patterns

- Copying files between machines
- Working with files
  - File types
  - Displaying the contents of files
  - Searching for patterns in files
  - Comparing files for differences
  - Editing files
  - Other handy commands which operate on files
- Processes
  - Programs and the search path
  - Process management
    - Job control
    - Terminating processes
  - Standard input and output
    - Output redirection
    - Pipes
- Shell scripting
- Command summary

Introduction

About this document

Audience and purpose

This document was written primarily as a source of information for VLSCI users who are new to Unix, however the concepts are applicable to a much wider audience. A small number of points are specific to our systems, but most of the document applies to any computer running a version of Unix.

The aims of the document are twofold:

1. To help you become a productive Unix user.
2. To show you the “Unix Way”, so that you can go off and discover the rest of the system on your own.

It is impossible for a such a short document to explain the entirety of Unix, so we focus on the key issues from a user’s perspective. We make no attempt to be exhaustive in our explanations, instead we prefer to use examples to demonstrate how things work. We hope that you will experiment on your own, and where necessary, follow up a topic with further

reading on the internet, in a book, or in the online Unix manual.

Typographic conventions

In this document textual interaction with the computer is indicated by like so:

- `Input from the user is in black font.`
- `Output from commands is in red font.`
- `The shell command prompt is in blue font.`

We use an ellipsis (`...`) to indicate places where the output of a command has been truncated for brevity.

History and philosophy of Unix

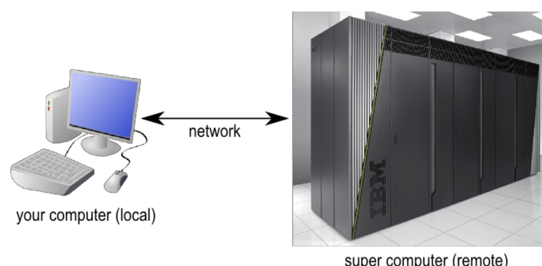
Unix was created at AT&T Bell labs in 1969 following the more ambitious, but less successful, MULTICS project. The popularity of the system grew quickly, especially at universities, because it was cheap, portable, and because the entire source code was provided to users. During the 1980s and 1990s several commercial implementations of Unix were created, particularly for the domain of high-end scientific workstations. The proliferation of many different versions of Unix prompted various standardisation efforts, most notably the Portable Operating Systems Interface (POSIX). Increasingly restrictive software licences and legal battles over Unix prompted the creation of the GNU project (GNU's Not Unix) to make an operating system based on the philosophy of free and open software. In the early 1990s an open source kernel, called Linux, was created for IBM PC compatible computers. The combination of this kernel with the tools created by the GNU project led to the GNU/Linux operating system, which is now one of the most widely used Unix-like systems in the world. In recent times GNU/Linux has been ported to a wide variety of systems; everything from wristwatches to supercomputers. It is also worth noting that Unix is the foundation of the Apple Mac OS X operating system, and thanks to POSIX, the standard suite of Unix tools is also available on that platform.

Unix remains popular after 40 years, especially in scientific computing, because of its robustness and flexibility. From the very beginning it was designed to support multiple users running multiple tasks concurrently, which means that security and stability are key parts of its design. The flexibility of Unix comes from a philosophy of building complex systems as compositions of smaller and simpler tools. Unix has accumulated a large suite of small programs which "do one thing and do it well", and which adhere to a common interface. This enables users to build their own custom tools by simply plugging existing ones together. It is this inherent programmability which makes Unix adaptable to a wide range of computing tasks, and it is the reason that Unix users continue to favor the command line interface, even when graphical user interfaces are commonplace on modern computers.

Login and logout

Remote login

A *remote login* is a connection between your personal computer and another computer via a network (normally the internet). Your computer is *local* and the other computer is *remote*. In the case of VLSCI, the remote system will be one of our supercomputers. A remote log in allows you to type commands into your computer and have them execute on the remote system, with the results displayed on your screen.



There are a few ways to remotely login to a Unix computer, but at VLSCI we require you to use the Secure Shell (ssh). It is secure because it encrypts all the data sent between the connected computers, which, amongst other things, prevents eavesdroppers from seeing your password in plain text.

In order to use ssh you must run an *ssh client* on your computer and tell it the hostname of the remote computer. These are the hostnames of the supercomputers at VLSCI:

- avoca.vlsci.unimelb.edu.au
- merri.vlsci.unimelb.edu.au
- bruce.vlsci.unimelb.edu.au

Avoca is an IBM BlueGene/Q whereas Merri and Bruce are x86 clusters. More information about these machines is available on this web page:

http://www.vlsci.org.au/page/computer-software-configuration

There are a variety of ssh clients available depending on which operating system you use; we show how to use ssh clients on Windows, OS X and Linux below.

SSH login from Windows

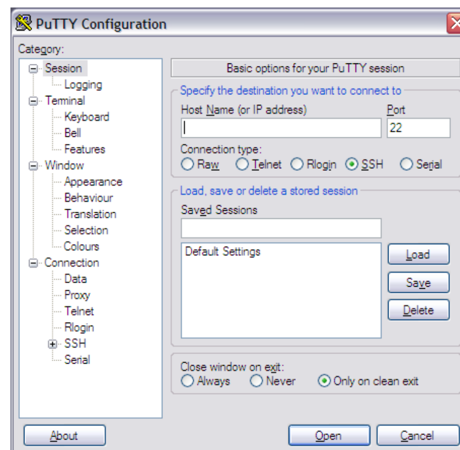
On Microsoft Windows (95, 98, 2000, XP, Vista, 7) we recommend that you use the PuTTY ssh client. PuTTY (putty.exe) can be downloaded from this web page:

http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

Documentation for using PuTTY is here:

http://www.chiark.greenend.org.uk/~sgtatham/putty/docs.html

When you start PuTTY you should see a window which looks something like this:



To connect to one of the computers at VLSCI you should enter its hostname into the box entitled “Host Name (or IP address)”, then click on the *Open* button. All of the settings should remain the same as they were when PuTTY started (which should be the same as they are in the picture above).

In some circumstances you will be presented with a window entitled *PuTTY Security Alert.* It will say something along the lines of “The server’s host key is not cached in the registry”. This is nothing to worry about, and you should agree to continue (by clicking on Yes). You usually see this message the first time you try to connect to a particular remote computer.

If all goes well, a *terminal* window will open, showing a prompt with the text “**`login as:`**”. An example terminal window is shown below. You should type your VLSCI username and press *enter*. After entering your username you will be prompted for your password. Assuming you type the correct username and password the system should then display a welcome message, and then present you with a Unix prompt. If you get this far then you are ready to start entering Unix commands and thus begin using the remote computer.

SSH login from Mac OS X and Linux

Both Mac OS X and Linux come with a version of ssh (called OpenSSH) that can be used from the command line. To use OpenSSH you must first start a terminal program on your computer. On OS X the standard terminal is called *Terminal*, and it is installed by default. On Linux there are many popular terminal programs including: *xterm*, *gnome-terminal*, *konsole* (if you aren't sure, then xterm is a good default). When you've started the terminal you should see a command prompt. To log into Merri, for example, type this command at the prompt and press *return* (where the word ***username*** is replaced with your VLSCI username):

```
ssh username@merri.vlsci.unimelb.edu.au
```

The same procedure works for avoca and bruce, or any other machine where you have an account.

You may be presented with a message along the lines of:

```
The authenticity of host 'merri.vlsci.unimelb.edu.au
(128.250.166.34)' can't be established.
...
Are you sure you want to continue connecting (yes/no)?
```

Although you should never ignore a warning, this particular one is nothing to be concerned about; type **yes** and then press *enter*. If all goes well you will be asked to enter your password. Assuming you type the correct username and password the system should then display a welcome message, and then present you with a Unix prompt. If you get this far then you are ready to start entering Unix commands and thus begin using the remote computer.

Logout

When you have finished using the remote computer you can terminate your connection with the **logout** command, or simply quit your ssh client.

The shell and the command line

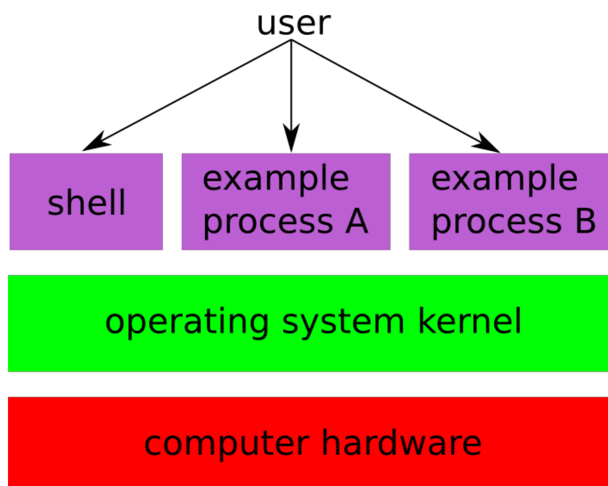
When you login to a Unix machine you are confronted with a prompt: a short piece of text indicating that the system is ready for you to type a command. The prompt on Merri (a computer at VLSCI) might look like this:

```
[username@merri ~]$
```

where the word **username** is replaced by your actual username on the system. The content of the prompt is configurable, and it might be different on other systems that you encounter. In this document, for simplicity, we show the prompt as a single dollar sign (**$**).

The command line itself is provided to you by a program called the *shell*. The shell acts as an intermediary between you and the underlying operating system. Each time you enter a command, the shell parses it and then asks the system to carry it out. The shell is also programmable, which means you can automate sequences of commands that would be tedious to enter manually. Most Unix systems provide several alternative shells for you to use, but most GNU/Linux systems use the BASH shell as the default. This document assumes you are using BASH, but most of the examples should work equally well in other shells.

The diagram below shows how the shell relates to various other components in a computer system:



At the bottom of the diagram sits the computer hardware. Immediately above the hardware is a software abstraction called an *operating system kernel.* The kernel is a piece of software

which manages the resources of the computer (disk drives, screen, keyboard *etcetera*) and makes them available to your programs. User applications sit on top of the stack, and interface with the hardware via the kernel. The shell is just an ordinary user application that happens to provide a convenient command interpreter.

It should be noted that users can interact with Unix in other ways besides the shell. Graphical user interfaces (GUIs) are also available, and most users run a shell inside a GUI window nowadays. Nonetheless, this document focuses on the shell because its programmability offers productivity gains unrivalled by other interfaces.

An example command

Here's a simple example command to get you started. Type the following command into the shell and press enter:

```
$ date
Tue Mar  6 21:05:56 EST 2012
```

As you can see the output of the command is the current local date and time.

Note that once the previous command has completed the shell displays a new prompt indicating that it is ready for the next command to be entered.

Many Unix commands accept arguments (sometimes called flags) which request specific features to be activated. For example you can ask the **`date`** command to produce its output in Coordinated Universal Time (UTC) like so:

```
$ date -u
Tue Mar  6 10:06:14 UTC 2012
```

It is not uncommon for a command to accept many different arguments, and, in most cases, more than one argument can be supplied at the same time. Arguments are separated by one or more space characters and they are usually case-sensitive.

How do you find out what arguments are accepted by a command? In the case of **`date`** you can pass it the **`--help`** argument and it will print out some usage information, but unfortunately not all commands behave the same way. This is related to the more general question about how to get help about using a Unix system, which we cover in more detail in the section entitled *Getting Help* later in this document.

Command history

A convenient feature of the shell is that it keeps a record of the most recent commands that

you have entered. It is often the case that previous commands are repeated in the near future. You can go back to previous commands by using the up-arrow key (and forwards with the down-arrow key). The command history saves you from typing it all in again each time, which is very useful when the command was large.

You can also view your most recent commands with **history**:

```
$ history
...
980  ls -la
981  ./myprog
992  man awk
```

Each command in the history is numbered and the shell lets you run the command again by prefixing its number with an exclamation mark. For example, the command "**!980**" would cause "**ls -la**" to be run again.

Command editing

The command line can be edited in a number of ways:

- Forward and back arrows move the cursor left and right.
- The delete key erases the character to the left of the cursor.
- Control-k deletes everything from the cursor to the end of the line.
- Control-u deletes everything from the cursor to the start of the line.
- Control-e moves the cursor to the end of the line.
- Control-a moves the cursor to the start of the line.
- Control-r searches in reverse through the history.

Tab completion

Pressing the tab key causes the shell to expand the text sitting just prior to the cursor:

- If zero names match then nothing happens.
- If exactly one file name matches then the text is expanded to that name.
- If more than one file name matches then a list of matching names will be displayed. You will need to type one or more characters to disambiguate before pressing tab to continue the completion.

Getting help

One of the biggest problems facing a new Unix user is the sheer volume of commands

available. One of the downsides of the command line compared to a graphical interface is that there is no ordered searchable menu. Even if you know that a command exists you still may not know how to use all its features. So how do you discover new commands, and how do you learn how to use existing ones?

The Unix manual

The traditional approach is to use the built in Unix manual, which is provided by the **`man`** command:

```
$ man time
```

(use the arrow keys to move up and down and **`q`** to quit the manual).

The manual has an entry for most of the standard Unix commands and for some of the less common ones too. It even has an entry for itself:

```
$ man man
```

You can search for articles in the manual by keyword using the **`-k`** argument:

```
$ man -k compiler
..
fort77                (1p) - FORTRAN compiler (FORTRAN)
gcc                   (1)  - GNU project C and C++ compiler
zic                   (8)  - time zone compiler
```

Most keyword searches in the manual return large lists of results (we show only a small part of the output above), so you will need to spend some time sifting through them to find the command you are looking for.

The Unix manual is divided into multiple sections and a given command could appear in many different sections. In the output of **`man -k`** above you will see numbers in parentheses, which indicate the section of the manual referred to by a particular entry. If you want to look at a specific section of the manual then you need to provide the section name as an argument.

Compare the results of these two commands:

```
$ man 1 cp
$ man 1p cp
```

Other sources of help

In truth the Unix manual is not particularly friendly to beginners and you'd be excused for trying to look elsewhere for help.

Nowadays there is a wealth of information about Unix on the Internet. In fact there is so much information that it is hard to know where to look, although a well constructed Google search can help you narrow down the results. Sometimes nothing beats a good old-fashioned textbook, and there are plenty to choose from. A popular choice for beginners is *Learning the UNIX Operating System* by Peek, Todino-Gonguet and Strang.

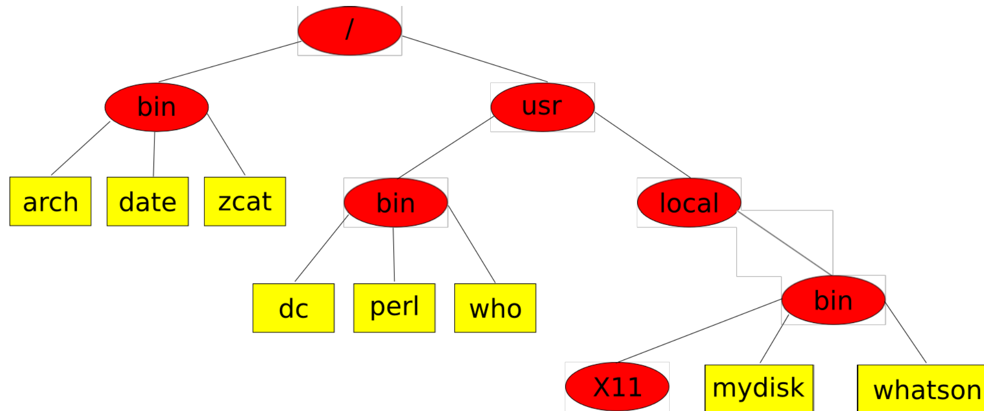
If you are a VLSCI user then you can always ask us for help in a number of ways:

- Come over to our offices and speak to someone in person.
- Write your question in an email and send it to help@vlsci.org.au
- Post the question on our forum and see if another one of our users has a solution for you: https://help.vlsci.unimelb.edu.au/forum/

The file system

Unix provides a *file system* to store and organise data which resides on permanent storage such as disk drives (the actual storage media could be any technology). The file system consists of directories and files. If you are a Windows or Mac user, directories represent the same concept as *folders* on those systems. Files contain data and programs, and directories contain files and possibly other directories, thus forming a hierarchy. At the top of the hierarchy is the *root* directory, which is usually denoted by forward-slash character (/).

The diagram below illustrates a small part of a typical Unix file system, rendered as an upside down tree (the root is at the top and the branches extend downwards). The example is for illustrative purposes only, the file system on a real computer would contain many more files and directories.



In the diagram directories are indicated by red ellipses and files are indicated by yellow rectangles. Note that directories may contain other (sub) directories and also files, and directories can be empty too. The names of the files and directories have been chosen from a real system, but they are not significant in this discussion. However, there are a few things to note about names in general:

1. The syntax for names does not distinguish between directories and files.
2. Names do not have to be unique. In the example above there are three directories with the name **`bin`**.
3. On most Unix systems file names are case sensitive.
4. Filenames may contain space characters, but this practice is often discouraged because some older Unix tools don't handle them well.

As we shall see shortly, Unix provides a naming scheme called *file paths* which accommodates duplicate names but allows us to refer to things unambiguously.

The home directory

Each user has their own private directory called their *home*, where they can store their own files and subdirectories. You can discover the name of your home directory with the following command:

```
$ echo $HOME
/home/username
```

The exact location of your home directory will vary from system to system, but it will usually end in your username.

The working directory

The working directory indicates the place in the file system where you are working. Many

commands operate on files in the working directory by default unless you specify otherwise. When you first log into the computer your working directory is set to be your home directory. You can discover the name of the working directory with the **pwd** command (for "print working directory"):

```
$ pwd
/usr/local/bin
```

You can change the working directory with the **cd** command (more on this later).

File paths

A *file path* identifies files and directories within the context of the file system hierarchy. They are called "paths" because they are written as a sequence of steps between two places. The steps indicate which directories you must traverse to get from the start to the end of the path. There are two basic forms of file paths, distinguished by where they start:

1. *Absolute file paths*: start at the root directory, for example: **/usr/local/bin**
2. *Relative file paths*: start at the working directory, for example: **src/C/Makefile** (note the absence of a forward-slash at the start of the path).

Unix uses the forward-slash character as a path separator. The notation **D/F** indicates that **F** is contained in the directory **D**. The root directory is just a special case with nothing on the left of the slash.

Consider this absolute path as an example:

```
/usr/local/bin/whatson
```

- **usr** is a directory contained in the **/** (root) directory.
- **local** is a directory contained in the **/usr** directory.
- **bin** is a directory contained in the **/usr/local** directory.
- **whatson** is a file (or perhaps a directory) contained in the **/usr/local/bin** directory.

Relative file paths are interpreted by appending them onto the working directory. For example, suppose we have the following relative path:

```
src/fractal.c
```

If the working directory is:

```
/home/username
```

then the absolute path of the file would be:

```
/home/username/src/fractal.c
```

Absolute paths unambiguously refer to a particular file or directory on the system, but they can become quite long and thus tedious to type. Relative paths are interpreted with respect to the working directory, which means that they are usually short. Indeed, the relative path of any file in your working directory is just the name of the file on its own - it couldn't be any shorter than that!

Special file path names

Unix also provides a few shorthand notations for certain paths:

- The tilde character (**~**) refers to your home directory, for example:
  ~/src/fractal.c
  is equivalent to:
  /home/username/src/fractal.c
  if your home directory is:
  /home/username/

- The single dot (.) refers to the working directory, for example:
  ./README.txt
  is equivalent to:
  README.txt

- The double dot (**..**) refers to the parent directory, for example:
  ../../sequences.txt
  refers to the file **sequences.txt** in the parent of the parent (i.e. grandparent) of the working directory.

Listing the contents of a directory

The contents of a directory can be listed with the **ls** command (lower case **LS**). In its simplest form it lists the names of the immediate contents of the working directory in alphabetical order:

```
$ ls
aadvark.c  a.out  data  memo.txt  zebra.hs
```

You can ask **ls** to give you a longer, more detailed listing, by passing it the **-l** (lower case **L**, for "long") argument:

```
$ ls -l
total 88
-rw-r--r-- 1 foo      staff     94 Feb  9  9:24 aadvark.c
-rwxr-xr-x 1 foo      staff   6713 Mar  9 15:24 a.out
drwxr-xr-x 2 foo      guest   4096 Mar  9 15:21 data
-rw-r--r-- 1 fred     student   43 Jan 12 20:26 memo.txt
-rw-r--r-- 1 wilma    student   12 Sep 19  2011 zebra.hs
```

The text "**total 88**" means that the current directory consumes 88 disk blocks, which says something about how much space it requires. This is a low-level detail that is not particularly important for the current discussion.

Next we see a list of information about five files. There are nine columns in each line which provide the following details:

1. Permissions of the file (who can read/write/execute it), e.g. **-rwxr-xr-x**
2. Number of links to the file (refer to information about file system links), e.g. **2**
3. Name of the user who owns the file, e.g. **fred**
4. Name of the user group associated with the file, e.g. **student**
5. Size of the file in bytes, e.g. **43**

6-8. Date/time that the file was last changed, e.g. **Mar 9 15:24**

9. Name of the file or directory, e.g. **memo.txt**

The **ls** command accepts an optional list of file paths as arguments. This allows you to list the contents of the named directories rather than just the working directory. For example:

```
$ ls ~/bin
c2hs  killmyjobs.sh
```

Hidden files

By default the **ls** command does not display information about files whose names start with a dot character (these are often called *dot-files*). If you want to see those files then you can use the **-a** argument to list *all* files. The most common place to find dot-files is in your home directory. These are often configuration files that control the behaviour of programs that you use. For example, the file **~/.bashrc** contains configuration options for the BASH shell.

Changing the working directory

When you log into a Unix computer your working directory is set to your home directory. You can change the working directory with the **cd** command (for "change directory"):

```
$ cd /usr/local/
```

The **cd** command accepts a file path as its argument, which specifies the new desired value of the working directory. The command will fail if the directory does not exist or you don't have permission to read it. If you run the **cd** command without any arguments then it will set the working directory to your home directory.

Making directories

You can create your own directory using the **mkdir** command:

```
$ mkdir test
```

The **mkdir** command accepts one or more file paths as its arguments which specify the names of the directories that you want to create. It will fail to create a directory if it already exists or if you have insufficient permissions.

Copying and moving (renaming) files

You can copy one or more files from one place in the file system to another using the **cp** command:

```
$ cp .bashrc .bashrc.backup
```

The above example copies the file **.bashrc** to **.bashrc.backup** (and does not change **.bashrc**).

The file(s) being copied are called *sources* and the place being copied to is called the *destination*. If there is one source then the destination could be either a file or a directory, but if there are multiple sources then the destination must be a directory (because it is not possible to copy multiple files into a single file). If the destination is a directory then the sources are copied into the directory keeping their original names.

The **cp** command will not copy directories by default, but this can be overridden with the **-R** flag, which will cause **cp** to copy the contents of a directory recursively (that is the directory will be copied, plus all its files and subdirectories transitively).

You can move (rename) a file or a directory using the **mv** command:

```
$ mv .bashrc .bashrc.old
```

The above example renames the file **.bashrc** to the new name **.bashrc.old**.

Warning: by default the `cp` and `mv` commands will overwrite a destination file if it already exists. This means you will lose the old contents of the destination file!

Removing files and directories

You can remove files with the **`rm`** command and remove directories with the **`rmdir`** command.

The following command will remove the file called **`.bashrc.backup`** from the working directory:

```
$ rm .bashrc.backup
```

The following command will remove the directory called test from the working directory:

```
$ rmdir test
```

Note that **`rmdir`** only removes empty directories. If you want to recursively remove a directory and all its files (and subdirectories and so on) then you can pass the **`-r`** argument to **`rm`**:

```
$ rm -r test
```

Warning: `rm` is a dangerous command (especially with the `-r` argument), it can cause you to lose precious data if you accidentally remove the wrong things. You must always maintain up-to-date backups of all your important files! Each time you use the `rm` command you should pause for a moment and ask yourself "do I really want to do this?". Some people use the `-i` argument which causes `rm` to prompt you before each removal.

File permissions

Every file and directory in the file system has permission attributes which specify whether the file may be read, written (modified) or executed by certain classes of users. The precise meaning of read, write and execute is explained by the table below (note the differences between files and directories):

	read	**write**	**execute**
file	Contents of file may be viewed.	Contents of file may be changed.	File may be executed as a program.

directory	A list of the directory contents may be viewed (i.e. user can **ls** the directory).	User may create and remove files in the directory.	The directory may be traversed by the user (i.e. user may **cd** into the directory).

The permissions are further divided into three categories of users:

1. The owner of the file.
2. The user group associated with the file.
3. All other users on the computer.

The owner of the file is normally the user who created it.

Each user belongs to one or more *groups* on the system. This provides a simple way to share files with a limited selection of users. You can find out which groups you are in with the groups command:

```
$ groups
guest tutorial printer
```

By setting the group permissions on files appropriately you can share them with other users on the system in the same group. Unfortunately each file can only have a single group at any one time, even though each user can belong to many groups.

The “other users” permissions apply to anyone who is not the owner of the file and not a member of the group of the file, in other words everyone else.

Recall from above that the output of the command **ls -l** shows, amongst other things, the permissions of the files in a directory, written as a ten character string like so:

```
-rwxr-x---
```

The meaning of each of the ten characters from left to right is as follows:

1. File type. Dash (**-**) means it is a regular file, (**d**) means it is a directory.

2-4. Owner read (**r**), write (**w**), execute (**x**).

5-7. Group read (**r**), write (**w**), execute (**x**).

8-10. Other read (**r**), write (**w**), execute (**x**).

A dash (-) in any of the read, write or execute positions means that the corresponding permission is denied.

The example **-rwxr-x---** can be decoded like so:

- It is a regular file (first character is a dash).
- The owner of the file can read, write and execute it (characters 2-4 are **rwx**).
- Any user in the group of the file can read and execute it, but they cannot write it (characters 5-7 are **r-x**).
- Everyone else who is not the owner of the file and not in the group of the file cannot read, write or execute it (characters 8-10 are ---).

Changing the permissions on a file

You can change the permissions of a file or directory using the **chmod** command. For example to add group read and write permissions to the file **fractal.c** we could use the command:

```
$ chmod g+rw fractal.c
```

The syntax of the chmod command is complicated, but basically it follows the pattern:

chmod *user-code operator permission-string* files

where:

- *user-code* is one of **u**,**g**,**o**,**a**
  - **u** : the owner of the file.
  - **g** : any user in the group of the file.
  - **o** : any user who is not the owner and not in the group of the file.
  - **a** : all users.
- *operator* is one of **+**,**-**,**=**
  - **+** : add these new permissions to the current permissions for the file.
  - **-** : remove these permissions from the current permissions of the file.
  - **=** : set these permissions of the file to be exactly these ones specified (but all other permissions remain unchanged).
- *permission-string* is some combination of **r**,**w**,**x**

More example uses of **chmod**:

- Add read and execute permissions for everyone to the file **myprog**:

  ```
  $ chmod a+rx myprog
  ```

- Remove read permissions for anyone who is neither the owner of, nor in the group of the file **myprog**:

```
$ chmod o-r myprog
```

- Set the permissions of the file **myprog** to be read and execute to the owner (and leave the permissions of other users unchanged):

```
$ chmod u=rx myprog
```

The **chmod** command accepts a **-R** argument which causes it to recursively change the permissions in all the files and subdirectories of a specified directory, for example:

```
$ chmod -R g+rw my_data_files
```

Changing the group of a file

You can change the group of a file or directory with the **chgrp** command, for example:

```
$ chgrp guest fractal.c
```

The first argument is the new group name and the rest of the arguments are the files and/or directories that you want to change.

The **chgrp** command accepts a **-R** argument which causes it to recursively change the group in all the files and subdirectories of a specified directory, for example:

```
$ chgrp -R staff my_data_files
```

How to keep your files private

It is important to know how to keep your confidential data private, but as you can see from the discussion above, Unix file permissions can be complicated.

At VLSCI we have hundreds of users and most of them are working on research projects. Due to the nature of their work, our users typically want to share data with people in their research project but hide it from everyone else. To facilitate this, on our systems, each user belongs to one or more projects, and each project has a corresponding Unix group of the form VRwxyz, where wxyz is a unique four digit number, for example VR1234.

For research related data, we recommend that you only provide read and write access to the owner of the file (that is you) and the group of the file (which should be your research group). You should also restrict execute permissions on directories to the owner and group. Or in summary, the permissions string on confidential files and directories should have three

dashes (**---**) on the far right hand side. An easy way to ensure this is set for a directory called, say **my_data,** is to use this command:

```
$ chmod -R ug=rwX,o= my_data
```

This recursively sets the permissions on files to be readable and writable to the owner and group but not to anyone else. Note carefully that the **X** is a capital letter. This is a special feature of **chmod** which sets the execute permissions on files and directories which were already executable to some user (this ensures that execute permissions are given to directories and programs but not to normal files).

If you belong to multiple projects then you have to be careful which group is set for your files and directories.

File name patterns

In many circumstances you find yourself wanting to run the same command on a large collection of files whose names match a certain pattern. For instance you might like to remove group read permissions on all (and only) those files ending in "**.txt**". If there are lots of files that match the pattern then it is unlikely you will want to enter the same command once for every file. Fortunately the shell provides a concise language of patterns which solve this problem:

```
$ chmod g-r *.txt
```

In the example above we have used the pattern ***.txt** to match any file name which ends in the four characters "**.txt**" (which includes files with no characters to the left of the dot). The shell automatically replaces any pattern in a command with all of the file names it finds which match the pattern.

There are many things you can do with file name patterns, but the two key operators you need to know are the asterisk (*****) and the question mark (**?**). The asterisk matches strings of length zero or greater, whereas the question mark matches any single character.

Some example patterns:

- All files that end in "**.c**":

```
*.c
```

- All files that start with "**fred**" and end in anything:

```
fred*
```

- All files that have "**`fred`**" somewhere in their name:

  `*fred*`

- All files that start with "**`sequence.`**" and end with any three characters:

  `sequence.???`

Copying files between machines

You can copy files between your local computer and a remote computer using the **`scp`** ("secure copy") command. If your local computer uses Mac OS X or Linux then you can use **`scp`** from the command line. However, if your local computer uses Windows then we recommend you use the WinSCP tool:

http://winscp.net/eng/index.php

The principals of **`scp`** are the same for all systems, but WinSCP uses a graphical interface instead of a command line. For the sake of brevity we show the syntax for the command line in this document and refer Windows users to the WinSCP manual:

http://winscp.net/eng/docs/start

To copy a file from your local computer to a remote compute (say Merri at VLSCI) you use the syntax:

`scp` *source user@host*:*directory*/*dest*

where *source* is the name of the file on your local computer, *directory*/*dest* is the path for the file on the remote computer (the place to put it), and *user* is your username on the remote computer. If your username is the same on the local and remote computers then you can omit it in the command. Relative destination paths are interpreted with respect to your home directory on the remote machine (which means an empty destination path refers to your home directory itself). Here is an example which copies the file **`secrets.txt`** from the local computer to **`$HOME/private`** on merri:

```
$ scp secrets.txt username@merri.vlsci.unimelb.edu.au:private/
```

(You should replace ***`username`*** with your actual username on the remote computer.)

To copy a file from the remote computer to your local computer you use the syntax:

scp *user@host:directory/source dest*

The same rules apply as before but *directory/source* is the path of the file on the remote computer and *dest* is the place to put it on the local computer. Here is an example which copies **`$HOME/private/secrets.txt`** on the remote computer to the working directory on the local computer:

```
$ scp username@merri.vlsci.unimelb.edu.au:private/secrets.txt .
```

(Note the dot at the end which refers to the working directory on the local computer).

The **`scp`** command supports a **`-r`** argument which tells it to copy entire directories recursively in a similar fashion to **`cp -R`**.

Working with files

File types

Unix treats all files (barring some special cases) as merely sequences of bytes. That is to say, for normal files, it does not distinguish between different types or formats. It is up to the programs which work with files to decide how to interpret their contents.

Having said that, users often think of files as having a *type* based on what kinds of operations they might like to perform on them. For example, we might think of a file as containing the source of a Python program, or as the sequence data from a DNA sample.

The **`file`** command tries to guess the type of a file based on its contents and various heuristics:

```
$ file main.c
main.c: ASCII C program text
$ file main.o
main.o: ELF 64-bit LSB relocatable, AMD x86-64, ..
$ file stats.r
stats.r ASCII English text
```

The last example above is a script from the R statistical programming language, which demonstrates that the **`file`** command is not perfect.

It is also common for users to distinguish files as being “text” or “binary” (though we emphasise again that Unix itself makes no such distinction). For example, the source code of

a HTML page is regarded as text, whereas the contents of a compiled program is regarded as binary. The Unix philosophy has tended to encourage the use of text files because they are generally "human readable" whereas binary files are generally not. However, a text representation of data might not always be as compact as a binary one (especially when numbers are being stored).

Traditionally text files contained sequences of bytes representing members of the ASCII character set, where each character occupies exactly one byte in the file. However, ASCII is slowly being superseded by Unicode, which contains all the ASCII characters plus many others, including characters from non Latin-based languages. Unicode characters are stored in files using a multi-byte encoding scheme such as UTF-8 (which was designed to be backwards compatible with ASCII).

Unix comes with a large number of commands which help you manipulate text files, and on GNU/Linux they generally support Unicode. However, the ASCII subset of Unicode is still widely used, and we stick to that character set in this document for simplicity.

Displaying the contents of files

You can display the entire contents of one or more files with the **cat** command:

```
$ cat poem.txt
fat cat
sat flat
```

The cat command is nice and simple, but it tends to be impractical for large files because it shows the entire contents at once. An alternative is the **less** command, which supports text scrolling and searching:

```
$ less stats.r
#############################################################
START_PHI <- 0
SIZE <- 100                # side of square grid in pixels
..
```

It is hard to show the output of the less command in this document, so we just show the first few lines in the example above. You can use the arrow keys to scroll up and down in the output, **/pattern** to search for "pattern" in the file, and **q** to quit.

Sometimes you only want to view the first or last few lines of a file: this can be done with the **head** and **tail** commands respectively:

```
$ head -3 stats.r
```

```
############################################################
START_PHI <- 0
SIZE <- 100                    # side of square grid in pixels

$ tail -5 main.c
     init();
     visualise(argc, argv);
     finalise();
     return 0;
}
```

The first argument to **head** and **tail** is the number of lines to display, which, if you leave it out, defaults to 10.

Searching for patterns in files

You can search for a pattern in one or more files using the **grep** command:

```
$ grep include *.c
main.c:#include "simulate.h"
queue.c:#include <assert.h>
queue.c:#include "types.h"
simulate.c:#include <gsl/gsl_rng.h>
```

The example above searches for all the lines that contain the pattern "**include**" in all the files in the working directory that end in "**.c**".

When **grep** finds a match for the pattern in a file it prints out the whole line. When multiple files are searched it prints the name of the file before the matching line.

The pattern can be a *regular expression* which allows sophisticated searches to be performed. We don't have space to describe regular expressions in this document, but there is quite a detailed entry on Wikipedia:

http://en.wikipedia.org/wiki/Regular_expression

The **grep** command has many arguments which control its behaviour. A couple of the more common ones are **-R** for recursive search, **-i** for case-insensitive search.

Comparing files for differences

You can find differences between two files using the **diff** command. For example, suppose we have two files with the contents of each shown in the table below:

hello.c	bonjour.c

`#include <stdio.h>`
`int main(void) {`
`    printf ("Hello World\n");`
`    return 0;`
`}`
	`#include <stdio.h>`
	`int main(void) {`
	`    printf ("Bonjour Monde\n");`
	`    return 0;`
	`}`

```
$ diff hello.c bonjour.c
3c3
<     printf ("Hello World\n");
---
>     printf ("Bonjour Monde\n");
```

The output of **diff** shows the lines where the files differ. The text **3c3** means that the 3rd line in the first file was changed to the 3rd line in the second file. The **<** character marks lines in the first file and the **>** character marks lines in the second file. If the files are the same then **diff** will produce no output.

Editing files

There are many ways to edit files on Unix, but perhaps the simplest way is to use the full-screen text editor called **nano**.

To create a new file, run **nano** with no arguments:

```
$ nano
```

To edit an existing file, run **nano** with the name of the file to edit as an argument:

```
$ nano poem.txt
```

A great feature of **nano** is that it has only a few commands and the most important ones are displayed at the bottom of the screen, like so:

```
^G Get Help          ^O WriteOut
^X Exit              ^J Justify
```

The **^** character means "hold down the Control Key while pressing the character next to it". So, for example, to exit **nano** you should hold down the Control Key and simultaneously press the **x** key (despite appearances the single-letter characters in **nano**'s commands should be lowercase). If you have changed the file before exiting **nano** will prompt you to save the

changes.

Experienced Unix users tend to prefer text editors with more features than **`nano`**. The most popular editors, especially for programmers, are **`vim`** and **`emacs`**. Unfortunately these editors have a much steeper learning curve, and so we don't cover them in this document.

Other handy commands which operate on files

There are lots of other commands on Unix which operate on files. We don't have space to describe them all in detail, but here is a short list which you might like to lookup in the future:

Command	**Description**
`awk`	Data extraction and reporting scripting language.
`cut`	Remove sections from lines in a file.
`find`	Search for files in a directory hierarchy using a pattern syntax.
`gzip`	File compression (to make the file consume less space).
`sed`	Tool for filtering and transforming text, similar to **`awk`**.
`sort`	Sort lines of text files based on defined fields in each line.
`tar`	Store many files in a single archive.
`tr`	Translate or delete characters in a file.
`uniq`	Report or omit repeated lines in a file.
`wc`	Count the number of lines, words and characters in files.

Processes

Programs and the search path

Almost every command you run in Unix is just an ordinary program which resides in the file system somewhere (although a small number of commands are built into the shell directly).

Unix programs are found in executable files, and come in two flavours:

1. Binary files which contain machine instructions.
2. Text files which contain scripts.

The programs which contain machine instructions are loaded into memory and executed directly by the computer's hardware, whereas the programs which contain scripts contain text commands to be executed by an interpreter (e.g. BASH, Python, Perl).

Each time you enter a command, the shell parses what you typed and determines the name of the program to run, which is often just the first word of the command text. For example, in the following the command the program to run is called **chmod**:

```
$ chmod -R ug=rwX,o= my_data
```

Once the shell has determined the name of the program to run it searches the file system for an executable file with the same name. This raises some important questions, such as where does it search, and what if there is more than one matching file? The shell solves these issues with a list of directories called a search path. You can print out the value of your search path using the following command:

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin
```

Each directory in the search path is separated by a colon (`:`) character. When the shell searches for a program it looks in each directory of the search path from left to right until a either a match is found or there are no more directories to consider. The first match is taken to be the program to run, therefore directories towards the left of the search path take precedence over directories towards the right. If no match is found then the shell will report an error of the form "**command not found**".

You can ask the shell to tell you where it finds a command using the **which** command:

```
$ which chmod
/bin/chmod
```

If the program is not in your search path then you must tell the shell exactly where to find it, for instance by giving an absolute path for the file. If the program is in the working directory then you should prefix its name with the two characters `./` (because the dot is shorthand notation for the working directory):

```
$ ./myprog -o out
```

Process management

In Unix terms, a process (also called a task) represents the running instance of a program combined with all its associated state. The state of a process includes, amongst other things, its allocated memory, a collection of environment variables, and various odds-and-ends to enable it to perform input and output. Processes are identified by a number called a *process ID* (PID), and each process is owned by a user, which affects what privileges it has on the system.

You can find out what processes are running with the **ps** command:

```
$ ps
  PID TTY          TIME CMD
21788 pts/30   00:00:00 bash
32360 pts/30   00:00:00 ps
```

By default **ps** prints a simple report about all the process that are owned by you. In the example above there are two processes, the first is the **bash** shell, and the second is the **ps** command itself (yes, **ps** prints information about its own process!). There are four columns in the output which provide the following information:

1. The PID of the process, e.g. `21788`
2. The terminal device connected to the process (for interactive processes), e.g. `pts/30`
3. The total amount of time the process has spent on the CPU, e.g. `00:00:00`
4. The name of the program running in the process, e.g. `bash`

You can see all the processes on the system with **ps -e**:

```
$ ps -e
PID TTY          TIME CMD
1 ?        00:00:05 init
2 ?        00:00:02 migration/0
3 ?        00:00:00 ksoftirqd/0
...
```

There are usually a large number of processes on a system at any moment, so the output of the above command is likely to be very long.

Unix is a multitasking operating system, which means that it can share the available computing resources amongst multiple active processes. It does this by maintaining a queue of processes that are runnable. A scheduler in the kernel decides which process to run next on a particular CPU. Running processes are allocated a slice of CPU time. When its time slice

is up the running process is preempted by the scheduler and put back in the queue until it gets another turn to run in the future. A process can also be blocked, which means it is not presently runnable because it is waiting for some external event, such as input from the user.

You can get a dynamic view of the current processes on the system using the **top** command, which shows the same kind of information as **ps** but in a continually updated panel (to see just your own processes type '**u**' and then your username, and to quit press '**q**').

Job control

Interactive programs, including the shell, read input from, and write output to a terminal device. Due to its multitasking nature, Unix allows a user to run multiple processes simultaneously. It is reasonable to let multiple processes share a terminal for output (which can be interleaved between them), but it is not reasonable for multiple processes to read their input from the terminal at the same time (otherwise chaos would ensue). The shell allows you to control which process has read access to the terminal by maintaining the concept of *foreground*, *background* and *suspended* jobs. A foreground job is a process that can read from the terminal, and there can only be one such job at any time. A background job is a process that can write to the terminal but not read from it. A suspended job is blocked from running, and it remains in that state until it is either awoken or terminated. There can be multiple background and suspended jobs at any time. Each job is numbered by the shell, which makes it easy to refer to in future commands, but the job number should not be confused with the PID.

The default behaviour of the shell is to run a new command as a foreground job. This is not a problem for short running commands because they only stay in the foreground for a brief moment. However it is desirable to run longer computations which don't need to read from the terminal in the background. This is achieved by placing an ampersand at the end of the command line:

```
$ xterm &
[1] 5391
```

The above example starts the **xterm** program and runs it in the background. The shell responds with the text "`[1] 5391`", which indicates this is job 1 with PID 5391.

You can find out the status of your current jobs with the jobs command:

```
$ jobs
[1]+  Running                    xterm &
```

You can bring a job back into the foreground with the **fg** command by specifying the job number:

```
$ fg 1
xterm
```

You can suspend the current foreground job by pressing Control-z. If we did that for the current example the shell would respond like so:

```
[1]+  Stopped                 xterm
```

This indicates that job 1 is suspended and control of the terminal is handed back to the shell. A suspended job can be returned to the foreground with **fg** or it can be sent to the background with the **bg** command:

```
$ bg 1
[1]+ xterm &
```

Terminating processes

A running process can be terminated in a number of ways:

- On its own without request from the user (perhaps because it runs out of input).
- After the user asks it to quit, such as by pressing Control-x in **nano**.
- After receiving a signal from the operating system that asks (or forces) it to quit.

You can request the foreground job to quit by pressing Control-c. This sends the process an interrupt signal. Most processes will terminate gracefully when given this signal.

You can also send an interrupt signal to a process using the **kill** command by naming its PID:

```
$ kill 5391
$ jobs
[1]+  Exit 15                 xterm
```

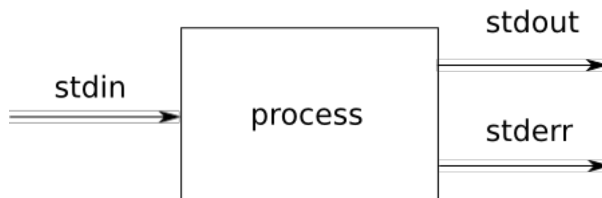
In extreme circumstances a process might not terminate when given the interrupt signal. This could be intentional, or it could be because the process has run out of control. You can be more forceful and send the process a *kill* signal (-9), which cannot be ignored:

```
$ xterm &
[1] 8518
$ kill -9 8518
```

Obviously you can only kill processes that you own.

Standard input and output

Every Unix process has three default streams associated with it: one input stream and two output streams associated with it. The input stream is called *standard input* (stdin), and the output streams are called *standard output* (stdout) and *standard error* (stderr):



As the names suggest, stdin is for input to the process, stdout is for normal output, and stderr is for error output.

Output redirection

By default the shell connects stdin to the terminal keyboard and stdout and stderr to the terminal screen, however, it is possible to redirect the streams to and from files.

For example we can redirect the stdout of the **ps** command and save it to a file like so:

```
$ ps > psout
```

The greater-than character (>) tells the shell to connect the stdout of the process to the file named on the right-hand-side. In this case the file is called **psout**, but you can choose any name you like. If a file with the same name already exists then it will be overwritten with new content, so you must use redirection with caution.

We can verify that **psout** contains the output of the **ps** command with **cat**:

```
$ cat psout
PID TTY          TIME CMD
7559 pts/8       00:00:00 bash
11524 pts/8      00:00:00 ps
```

You can redirect stderr in a similar fashion with **2>** (instead of **>**) and both of them at the same time with **&>**.

Standard input can be redirected to come from a file (instead of the keyboard) using the less-than (**<**) character, like so:

```
$ wc -w < psout
12
```

In the above example the **wc** command counts the number of words in the text in its stdin, which is redirected from the file **psout**.

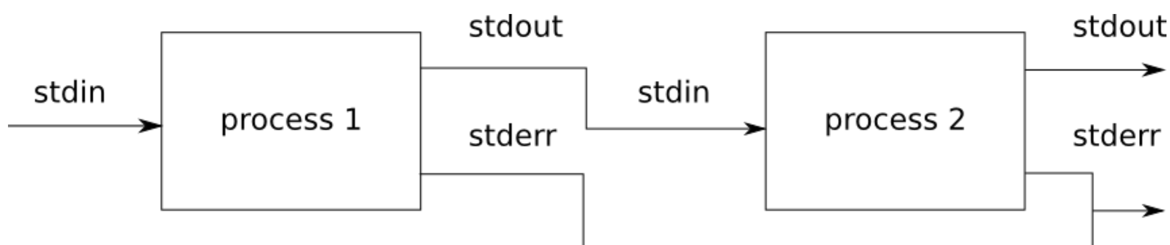
Both types of redirection can be used in the same command:

```
$ wc -w < psout > wcout
```

The above example redirects the contents of **psout** into the stdin of **wc** and then redirects the stdout of **wc** to the file **wcout**.

Pipes

In a similar fashion to redirection, the shell can also arrange for the stdout of one process to be connected to the stdin of another process (and merge the stderr streams):



This connection of processes is achieved using a pipe (**|**). For example we can **cat** the contents of the **psout** file and pipe it into the **wc** command like so:

```
$ cat psout | wc -w
12
```

You can build up even longer pipelines by chaining more than two processes together:

```
$ cat psout | sort | tail -1
  PID TTY          TIME CMD
```

And you can use file redirection with pipelines too:

```
$ cat psout | sort | tail -1 > out
```

Pipes are a handy way to build up complex tasks from simpler programs, and they demonstrate the advantages of the Unix philosophy of providing small programs which “do one thing well”.

Shell scripting

A shell script is an executable text file which contains a canned sequence of shell commands. Scripts allow you automate complicated tasks and avoid repeating yourself.

The first line in a shell script is special. It starts with a magic two letter sequence **#!** which is followed immediately by the path of the shell program:

```
#!/bin/bash
```

The rest of the file can contain any sequence of shell commands, exactly as you would type them at the prompt. For example we could turn our previous pipeline into a shell script by creating a text file with the following contents:

```
#!/bin/bash
cat psout | sort | tail -1
```

If you save the script to a file called **myprog** then you can make it executable like so:

```
$ chmod u+x myprog
```

And you can run **myprog** as a command in the normal way:

```
$ ./myprog
  PID TTY          TIME CMD
```

(note the ./ at the start of the file name because it is in the working directory and not in the search path).

The **bash** command language is actually quite full-featured. It has many of the things you'd expect to find in a programming language such as variables, conditionals, loops, functions, arrays and much more. It is impossible to cover all the features of the shell in this document, so we refer you to the Bash Scripting Guide:

http://tldp.org/LDP/abs/html/

The downside of shell scripting is that the syntax is optimised for making shell commands as concise as possible (to make them easy to type at the prompt), which tends to make the other constructs awkward. As such it is well suited to automating small to medium tasks which consist predominantly of calls to other commands on the system. For larger tasks which are

more algorithmically complex it is better to choose a more conventional programming language.

Command summary

Command	Description
Control-c	Interrupt (quit) the foreground job.
Control-z	Suspend the foreground job.
`awk`	Data extraction and reporting scripting language.
`bg`	Send a job to the background.
`cat`	Display the contents of a file.
`cd`	Change directory.
`chgrp`	Change the group of a file/directory.
`chmod`	Change the permissions of a file/directory.
`cp`	Copy a file/directory.
`cut`	Remove sections from lines in a file.
`date`	Display the date and time.
`echo`	Print a string.
`file`	Guess the type of a file.
`find`	Search for files in a directory hierarchy using a pattern syntax.
`fg`	Bring a background or suspended job to the foreground.
`grep`	Search the contents of files.
`groups`	Display the groups of a user.
`gzip`	File compression (to make the file consume less space).
`head`	Display the first few lines of a file.
`history`	List your command history.

jobs	List your active jobs.
kill	Terminate a process.
less	Display the contents of a file.
logout	End your login session.
ls	List the contents of a directory.
man	Lookup the Unix manual.
mkdir	Create a directory.
mv	Move (rename) a file/directory.
nano	Edit a file.
printenv	Display your environment.
ps	Show information about running processes.
pwd	Print the working directory.
rm	Remove one or more files.
rmdir	Remove an empty directory.
scp	Copy files between computers.
sed	Tool for filtering and transforming text, similar to **awk**.
ssh	Remote login to a computer.
sort	Sort lines of text files based on defined fields in each line.
tar	Store many files in a single archive.
top	Show current running processes.
tr	Translate or delete characters in a file.
uniq	Report or omit repeated lines in a file.
wc	Count the number of lines, words and characters in files.
which	Search for a command in the path.