Babraham Bioinformatics

An Introduction to Unix

A basic introduction to Unix concepts

Version 0.2

Licence

Introduction

The automated processing of large datasets is a task which is generally not well suited to interactive graphical programs. Anything which is required to run for long periods of time, or which will be processing large volumes of data in a manner which will generate too much output to be interactively displayed has traditionally happened on command line Unix systems.

Today Unix is very widely used for data processing due to its comprehensive set of automation tools and the ease with which new command line programs can be written. Whilst there are graphical programs which are well suited to summarise and analyse the output of the processing of large datasets much of the initial processing and collation is done on Unix systems.

This manual aims to provide an introduction to the common parts of Unix which anyone needing to use a command line based Unix interface will need to know to be able to effectively run an analysis. It does not go into any detail of specific analyses but sticks to core Unix concepts which will be required whatever you intend to do. It is aimed at people with little or no previous Unix experience.

Unix Commands

All programs in Unix are accessed by typing the program's name into a command shell. When you open a command shell you should see the last line looks something like this:

```
[andrewss@rocks1 ~]$
```

If you wanted to run the "whoami" command to find out what your username is you'd simply type that after the $ and press return.

```
[andrewss@rocks1 ~]$ whoami
andrewss
```

The system will search through a set of directories, called the PATH, to try to find a match to the command name you gave. If it can't find a match you'll get an error like this:

```
[andrewss@rocks1 ~]$ whoareyou
-bash: whoareyou: command not found
```

All operations in Unix are case-sensitive so you need to get the case as well as the command name correct. In general all core Unix commands are all lower case.

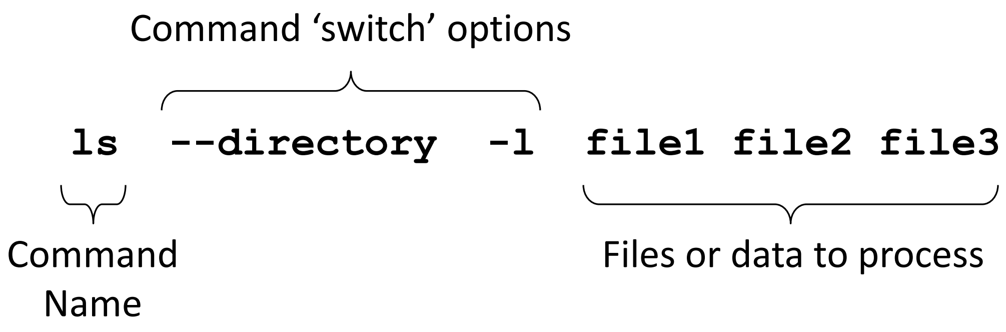
```
[andrewss@rocks1 ~]$ whoami
andrewss
[andrewss@rocks1 ~]$ WhoAmI
-bash: WhoAmI: command not found
```

If you want to run a program which isn't in your PATH then you can do this by specifying the full location of the program rather than having the shell search for it. This also applies to running a program in your current directory where you'd need to tell the shell it was OK to run that program even though it wasn't in your PATH

```
[andrewss@rocks1 ~]$ /bi/home/andrewss/whoareyou
You are number 3
[andrewss@rocks1 ~]$ ./whoareyou
You are number 3
```

Unix commands generally require you to provide a series of options along with the command name so that the program knows exactly what to do. There are interactive Unix programs which will prompt you for further information, but in general commands are not interactive and you need to supply all information up front.

Additional information is provided either by adding command 'switches' after the program name or by providing data to process (usually lists of file names) at the end. The typical anatomy of a Unix command is therefore something like:



Command switches are generally specified by either having two dashes and a long name, or a single dash and a single letter name. For a specific option there are often both single letter and long option variants, so `-d` and `--directory` might do the same thing.

For single letter command switches you can join the options together in a single set of option letters to save space, so instead of doing:

```
-l -t -r
```

You could simply do

```
-ltr
```

Although some command options tend to get reused by a lot of programs there is no enforced standard for what options are provided to any program, so the only way to be sure is to look at its documentation. For most core Unix commands you can see the documentation for it by looking at the manual page. You can access this by running:

```
man [command name]
```

Man pages tend to be fairly technical, but they will always list all of the command switches and what they do. They will also say what other data needs to be passed after the switches. The start of the man page for ls looks like this for example:

```
LS(1)                     User Commands                     LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information about the FILEs (the current directory by
       default).  Sort entries alphabetically if none of -cftuvSUX
       nor --sort.

       Mandatory arguments to long options are mandatory for short
       options too.

       -a, --all
              do not ignore entries starting with .
       -A, --almost-all
              do not list implied . and ..

       --author
              with -l, print the author of each file

       -b, --escape
              print octal escapes for nongraphic characters
```

If you find that the program you're using doesn't have a man page (many scientific applications won't have one for example), then you can normally get at the programs help by adding the `--help` or `-h` command switch.

```
[andrewss@rocks1 ~]$ fastqc --help

            FastQC - A high throughput sequence QC analysis tool

SYNOPSIS

        fastqc seqfile1 seqfile2 .. seqfileN

    fastqc [-o output dir] [--(no)extract] [-f fastq|bam|sam]
           [-c contaminant file] seqfile1 .. seqfileN

DESCRIPTION

    FastQC reads a set of sequence files and produces from each one a quality
    control report consisting of a number of different modules, each one of
    which will help to identify a different potential type of problem in your
    data.
```

Files and Directories

All of your file and directory management in a Unix environment is generally achieved using command line programs. There are graphical file managers, but it's a good idea to learn how to manage your files using the command line, since many of the operations you learn in doing this will be of use when processing data using other command line programs later on.

File Paths

Unix uses a file system based on a tree of directories. When you see the full path to a directory it will be shown as a list of directory names starting from the top of the tree right the way down to the level where your directory sits. Directory names in a path are separated by a `/` (forward slash) character. A typical directory name might be something like:

```
/home/andrewss/data/february/RNA-Seq
```

Directory names can contain spaces but you'll often find that people working in Unix prefer to keep both file and directory names without spaces since this simplifies the process of working with these files on a command line. Directory or file names containing a space must be specified on the command line either by putting quotes around them, or by putting a `\` (backslash) before the space.

```
"/home/andrewss/Directory Containing Spaces"
/home/andrewss/Directory\ Containing\ Spaces
```

Unix does not have separate drives as you might see under windows, but instead can attach different storage devices at any point in the directory tree, so you could add in a new disk and put that at `/home` so all directories under /home go onto that disk. When using the filesystem you generally don't need to worry about where physical disks or partitions exist in the file tree.

When specifying directory paths you can do this either by constructing a full path from the top of the directory tree (called the root directory or `/`), or you can construct a relative path from your current location. To help in constructing paths there are some directory aliases which you can use:

`.`	(single dot) means the current directory
`..`	(double dot) means the directory immediately above the current directory
`~`	(tilde) means the home directory for your user

So you can make directory paths like this:

- `/data/projects/simon/analysis/` - a full directory path to a specific location

- `projects/analysis/` - a directory called analysis which is in a directory called projects which is in the current directory. This could also be written `./projects/analysis`

- `../../analysis/` - go up two directory levels from the current directory and then down into a folder called analysis

- `~/scripts/` - a directory called scripts which is inside your home directory

One additional convention you might see is that any file or directory whose name starts with a dot is classified as a hidden directory and will not show up in normal file listings. You should therefore avoid using a dot at the start of a file or directory name unless you want it to be hidden.

Whilst you can type out full directory or file paths, a really helpful tool on the command line is tab completion. If you start typing a file path you can press the tab key - if the part you have typed only matches one possible file or directory then the rest of the name is filled in automatically. If it can match more than one completion then nothing will happen, however if you press tab again the shell will show you what possible completions there could have been so you can see what options you have.

```
$ /bi/home/andrewss/Sc [hit tab]

$ /bi/home/andrewss/Scripts/ (automatically completes the name)

[hit tab] - nothing happens
[hit tab again]

Bash/ Perl/
$ /bi/home/andrewss/Scripts/
```

Shows the possible completions and restores the command so you can continue it.

Listing files and directories

To see what files are already on your filesystem you can use the `ls` command. If you run this without any arguments it will provide a simple listing of the contents of the current directory. If you pass it a specific directory path it will list the contents of that directory instead.

Common options you'll use with `ls` are:

- `ls -l` List in long format which shows the permissions, ownership, modification date and size of files in addition to their name. File sizes are shown in bytes, but you can add the `-h` flag to get them in more human friendly units (ie `ls -lh`)
- `ls -a` Lists files but includes files whose name starts with a `.` and which would normally be hidden
- `ls -d` When listing a specific directory lists the directory itself and not its contents.

Creating directories

The command to create a directory is `mkdir`. If you simply pass it a new name it will create a directory with that name in your current location. You can also pass it a full path to the new directory you want to create but this will only work if only the last item in the path needs to be created. If you want to create a set of directories all at once you need to pass the `-p` flag.

```
$ ls -l
total 0
$ mkdir new_directory
$ ls -l
total 2
drwxrwxr-x 2 andrewss bioinf 0 Oct 11 11:36 new_directory
```

```
$ mkdir another_dir/with_subdir
mkdir: cannot create directory `another_dir/with_subdir': No such file or
directory


$ mkdir -p another_dir/with_subdir
$ ls -l
total 4
drwxrwxr-x 3 andrewss bioinf 29 Oct 11 11:36 another_dir
drwxrwxr-x 2 andrewss bioinf  0 Oct 11 11:36 new_directory
$ ls -l another_dir/
total 2
drwxrwxr-x 2 andrewss bioinf 0 Oct 11 11:36 with_subdir
```

Changing directory

When working in Unix your command shell has one directory which it is using as its working directory, and all file paths you use start in that directory unless you specifically say otherwise. When you first log in your working directory will be your home directory, but you can change this to wherever you actually want to do your work.

You can find your current working directory path using the pwd command. You will also see that the name (not the path) of your current directory will also be shown in your command prompt.

```
[andrewss@rocks1 bioinf]$ pwd
/bi/group/bioinf
```

You can change your working directory using the `cd` (change directory) command. You pass this either a relative or an absolute path to a new directory and your working directory will be moved to that location.

```
$ pwd
/bi/group/bioinf

$ cd Steven/
$ pwd
/bi/group/bioinf/Steven

$ cd ../..
$ pwd
/bi/group

$ cd ~
$ pwd
/bi/home/andrewss

$ cd /bi/scratch/Genomes/
$ pwd
/bi/scratch/Genomes
```

Wildcards

When specifying a file or directory you often want to generate a list of several related files (all the files in the current directory ending in .txt for example). Unix provides an easy way to do this using wildcards.

A wildcard is simply part of a file or directory path which can match multiple pieces of text. There are actually quite a large number of ways of specifying a wild card, but the two common ones which will work for most situations are:

1. `*` This matches any number of any character (including the empty string)
2. `?` This matches exactly one instance of any character.

You can construct file or directory paths using multiple wildcards and your command shell will substitute these with the matching set of real file or directory paths before running the command, so from the point of view of the command it will look as if you actually typed out all of the individual files.

If I wanted to list all of the Perl (.pl) files in the current directory I could therefore do:

```
$ ls *pl
copy_bristol_data.pl  make_fastq_from_raw.pl  remove_spaces.pl
extract_tiles.pl      remove_blank_lines.pl   rename_tophat_bam_files.pl
```

I can also use multiple wildcards - if I wanted to list all text files (.txt) in any subdirectory of the current directory I can do:

```
$ ls */*txt
bin/CREATING_CUSTOM_GENOMES.txt       FastQC/RELEASE_NOTES.txt
bin/INSTALL.txt                       group/file_list.txt
bin/LICENSE.txt                       SeqMonk/CREATING_CUSTOM_GENOMES.txt
bin/NOTICE_FOR_COMMONS_MATH_CODE.txt  SeqMonk/INSTALL.txt
bin/README.txt                        SeqMonk/LICENSE.txt
bin/RELEASE_NOTES.txt                 SeqMonk/NOTICE_FOR_MATH_CODE.txt
FastQC/INSTALL.txt                    SeqMonk/README.txt
FastQC/LICENSE.txt                    SeqMonk/RELEASE_NOTES.txt
FastQC/README.txt                     subread-1.3.5-p5/README.txt
```

One word of warning (especially when using wildcards when deleting!). Be very careful to ensure that you don't accidentally leave a space between a wildcard and the rest of your string. If instead of doing:

```
ls *pl
```

You did

```
ls * pl
```

You actually create two file lists, one from * and one from pl. * on its own will list everything in the current directory and unless you have a file called pl the second won't match anything. If you had linked this wildcard to a delete operation this is a really easy way to wipe out lots of files.

Viewing, Creating, Copying, Moving and Deleting Files

Once you are happy with how to move around your filesystem and create directories we can look at how to create, move and delete files.

Viewing files

Most of the data files you'll be working with on a Unix system will be simple text files which you can look at within your command shell. You can put the whole contents of a file onto your screen using the cat command.

```
$ cat test.seq
>test
ATGATGGAA
```

Many files will be too big to fit onto a single command window so to use these you can use a paging program which will allow to scroll through a large file to view it in a more sensible manner. The most common paging program is called `less`.

```
$ less fastq_screen.conf
# This is a configuration file for fastq_screen

###########
## Bowtie #
###########
## If the bowtie binary is not in your PATH then you can
## set this value to tell the program where to find it.
## Uncomment the line below and set the appropriate location
##

#BOWTIE /usr/local/bin/bowtie/bowtie
#BOWTIE2 /usr/local/bowtie2/bowtie2

fastq_screen.conf
```

When less is running you can use the following keyboard shortcuts to move around:

- [space] go forward one page
- `b` go back one page
- `j` go forward one line
- `k` go back one line
- `/`[some text] search for the next occurrence of [some text]
- `q` exit.

Text editors

There are many different text editors on Unix systems and there is no single editor which everyone has settled on as being the recommended one to use. The two most commonly used editors are vi and emacs but both of these are fairly intimidating to new users and have some very unusual features in the way that they operate, but they are both very powerful and are both likely to be found on most Unix systems (vi is guaranteed to be there). There are many introductory guides to these editors and the specifics of their use is beyond the scope of this manual but it would be a good idea to become familiar with one of these at some point.

For simple text editing there is a simple editor called `nano` which you can launch by typing:

```
nano some_file.txt
```

This will edit `some_file.txt`, and will create it if it doesn't exist. It has a very simple text based interface but it's generally functional and will do basic editing for you.

```
GNU nano 2.0.9                   File: test.txt                         Modified

This is a simple text file which I can edit using simple controls.

The commands listed at the bottom are all run by pressing Control
plus the listed character, so Control+O writes out the text and
Control+X exits.



^G Get Help     ^O WriteOut     ^R Read File    ^Y Prev Page    ^K Cut Text     ^C Cur Pos
^X Exit         ^J Justify      ^W Where Is     ^V Next Page    ^U UnCut Text   ^T To Spell
```

Command Redirection

Another common way to create files in Unix is by sending the output of a command to a file. Some commands will explicitly ask for a location to save results, but many will simply write the results to their standard output location which will normally go to the screen, but can be sent to a file.

Output redirection is achieved using the `>` symbol. You simply add this after the command and any normal output will get send to the file name you specify immediately afterwards.

```
[andrewss@rocks1 group]$ ls -l > file_list.txt
[andrewss@rocks1 group]$ cat file_list.txt
total 1054
drwxrwxr-x  4 andrewss bioinf   69 Apr 22 12:29 Alison_Galloway
drwxrwxr-x  6 fkrueger bioinf  765 Jun  3 14:19 Andrew_Dimond
drwxrwxr-x  5 andrewss bioinf  121 Oct 10 10:45 Cameron_Osborne
drwxrwxr-x  6 ewelsp   bioinf  100 Feb 22  2013 Chandra_2012
drwxrwxr-x  3 andrewss bioinf   36 Jun 17 12:17 Claire_Dawson
drwxrwxr-x  2 fkrueger bioinf  234 Apr 30 16:22 Claire_Senner
drwxrwxr-x  6 ewelsp   bioinf  189 Sep 25 13:29 clusterflow
drwxrwxr-x  4 ewelsp   bioinf   46 Feb 21  2013 Dawlaty_2013
```

Using `>` will create a new file, or if the file exists it will wipe the existing contents and start afresh. If you want to append content to an existing file you can use `>>` instead.

All programs actually have two types of output they can produce, standard output (often called stdout) is for expected output, and standard error (stderr) is for progress messages, warnings and errors. Normal redirection only redirects stdout. If you want to redirect stderr you can either send this to a different file using `2>` [error file] or you can mix it in with the standard output using `2>&1`.

Moving and renaming

In Unix the process to rename a file or directory to a new name and the process of moving it to a different place in the filesystem are the same. They are both achieved using the mv command, which requires you to provide the original name and location of the file and the new name and location you want to use.

```
$ ls
a_file  subdir

$ mv a_file b_file
$ ls
b_file  subdir

$ mv b_file subdir/
$ ls subdir/
b_file
```

If the location to move to is a directory which already exists then the file will be moved into that directory under the same name.

Deleting

Deletion of files and directories is most commonly achieved with the rm (remove) command. This takes in a list of files and removes them. If the -r flag is specified then the remove is ‘recursive’ and this will remove any directories listed and their contents. The rm command does not prompt you for any confirmation and with the -r flag it can potentially delete large portions of your file tree, so be very careful when using this command.

```
$ ls -l
total 4
-rw-rw-r-- 1 andrewss bioinf  0 Oct 11 12:53 a_file
drwxrwxr-x 2 andrewss bioinf 24 Oct 11 12:49 subdir

[andrewss@rocks1 Mkdir_test]$ rm a_file
[andrewss@rocks1 Mkdir_test]$ ls -l
total 2
drwxrwxr-x 2 andrewss bioinf 24 Oct 11 12:49 subdir

[andrewss@rocks1 Mkdir_test]$ rm subdir/
rm: cannot remove `subdir/': Is a directory

[andrewss@rocks1 Mkdir_test]$ rm -r subdir/
[andrewss@rocks1 Mkdir_test]$ ls -l
total 0
```

Permissions

The decision by the filesystem for whether or not to allow you to access data in certain files or directories is made based on a simple set of permissions. Permissions themselves are based on the ownership of each file or directory. When it is created every file is tagged with an owner and a group and it picks up a set of permissions which says what other users can do based on whether they match the owner or group of the file they're trying to access.

An individual user can belong to more than one user group, but a file can only have one group as its owner. You can see which groups you're in using the groups command.

```
$ groups wingetts
wingetts : bioinf seqfac
$ groups andrewss
andrewss : bioinf wheel bioapp seqfac repbase
```

For the actual permissions there are three main types of permission you can grant on a file these are:

1. Read
2. Write
3. Execute

..and these permissions are set separately for

1. User
2. Group
3. World (anyone)

When you list a set of files using ls -l you will see a permission string which tells you what permissions are on each file. This is the first set of characters at the start of each line.

```
drwxr-xr-x 4 andrewss bioinf 44 Jan 30  2013 Scripts
-rw-rw-r-- 1 andrewss bioinf 16 Oct 11 12:09 test.seq
```

The positions in this set are:

Position	Meaning
1	Is it a directory
2	Readable by user
3	Writable by user
4	Executable by user
5	Readable by group
6	Writable by group
7	Executable by group
8	Readable by world
9	Writable by world
10	Executable by world

Since 'executable' doesn't make any sense on a directory this flag on directories actually signifies whether the user is able to list the contents of the directory. If the have read but not execute permission on a directory then they can access its contents if they know the name of the file they want to access, but they can't get a list of the contents.

Generally the default permissions with which new files are created are sensible. On most systems they will be created so that the owner has read/write access, groups have read only access and world has no access but this can be changed by the system administrator.

Changing permissions

You can change the permissions on any file for which you are the owner using the `chmod` command. To do this you specify a file or set of files to modify and the type of change you want to make. Permissions can be specified in one of two ways:

`chmod g+rw` [some file]

In this structure the permission is defined in 3 parts:

1. Whose permissions we're changing (u=user g=group o=world a=all users (same as ugo))
2. Whether we're adding (+) or removing (-) permissions
3. Which permission we're changing (r=read w=write x=execute)

So in the above example g+rw would add read and write permissions to the group.

You can also specify permissions using a numeric scheme where 4=read, 2=write, 1= execute. You simply add together the permissions you want, so read, write would be 4+2 = 6. You then specify the 3 numbers to say what user,group and world can do, so

`chmod 750` [some file]
Would allow the user read,write,execute (4+2+1 = 7), group read, execute (4+1 = 5) and world nothing.

Pipes

One of the original design decisions for Unix was that it should consist of a lot of small programs which did one specific job very efficiently rather than larger programs which were very tied to a specific workflow. These small programs should then be able to be tied together very easily so you can create a string of programs which together perform the operation you want. The way that this linkage of programs occurs is via pipes.

A pipe in Unix is simply a way to connect the output of one program to the input of another. This then provides a mechanism by which you can read some data, filter it, sort it, summarise it and report on it (for example) all within a single Unix command.

To use a pipe to connect two programs, a and b you simply do:

```
program_a | program_b
```

This then runs program_a and then sends its output (stdout) to the input (stdin) of program_b. What will be displayed on the screen is the subsequent output of program_b.

For a more specific example we can list all of the text files in a directory and then instead of showing them we can count them using the wc program (with the -l option to show only the number of lines of output we got).

```
$ ls *txt | wc -l
18
```

This shows us that we had 18 text files. Some other common Unix programs which it might be useful to know about and which are often incorporated into pipes are:

Program	Function	Common switches
`wc`	Counts words/lines	`-l` (only lines)
`grep`	Selects lines matching patterns	`-v` (select non matching lines) `-i` (case insensitive)
`head`	Shows the top of a large file	-[number] show this many lines
`tail`	Shows the bottom of a large file	-[number] show this many lines
`sort`	Sorts a list of values	`-f` case insensitive
`uniq`	Deduplicates consecutive identical list members	`-i` ignore case

So if, for example I wanted to find the number of different lines for chr1 were in a text file I could do something like:

```
grep chr1 some_file.txt | sort | uniq | wc -l
```