Running commands with the Unix shell

Jeremy Sanders

October 2011

1 First things

The Unix shell is a *command line interface* (CLI) for running Unix programs. The typical way to run a program is by entering its name in the shell, which is the most simple type of command. Commands can also be placed in a file called a script for shell automation (see the notes on scripting later).

You should first log in to your computer. If you open a new *console* or *terminal* window (the terms are often interchangeable), the computer will start a new shell. Shells in new windows should begin with your home directory as their current directory. By default files will be read from the current directory (directories are called folders in other operating systems). Each console window has its own shell, with its own current directory and settings (environment).

In this document I will try to explain how to use the shell for simple operations. I suggest that you try out the commands below by typing them into the shell. The shell has a prompt which means that it is ready to accept commands to be inputted. The prompt for the tcsh shell is '>' by default. If I place > at the start of the line, do not try to type it into the shell. It is just to indicate a shell command. In this document <`--` or # and the following text are comments explaining what a command does.

Unix commands are given names which are usually short words or acronyms. The first item that you type at a prompt is usually the name of the command. Many commands have parameters, which are supplied as *arguments* - a list of text items separated by spaces. Often arguments are filenames.

The `pwd` command will tell you what the current directory of the shell is

```
> pwd
/home/user
```

By typing `pwd` you ask the shell to look for the program 'pwd' on the current path (see §13) and to run it. Some commands are actually built into the shell and are not separate programs. You can see where a command or program resides by using

```
> which pwd
/bin/pwd
```

The `date` command prints its output to the console,

```
> date
Mon Sep 16 20:13:35 BST 2002
```

More accurately commands write to a Unix *stream* called *stdout*, which is normally your console. They read from *stdin* and write error messages to *stderr*.

A basic command which takes arguments is `echo`:

```
> echo "hello, world"
hello, world
```

`echo` just prints out its arguments to stdout, the screen. Note that we use the quote symbols around hello, world, to make it a single argument to echo (see §14). The shell interprets spaces as separating arguments, unless they are enclosed in quotes (single or double) or they are preceeded by a backslash.

We can take the output from programs and put it into a file:

```
> echo "Hello, world" > testfile
```

The '>' symbol tells the shell to *redirect* the stdout output from any command into a file, which is called 'testfile'. This file is placed in the current directory by default. If 'testfile' already exists then tcsh will not overwrite the file (depending on certain shell settings, specifically noclobber). You can force an overwrite by using '>!'.

We can add text to existing files too:

```
> echo "Hello again" >> testfile
```

'>>' is a shell redirection which *appends*, or adds to a file. Now we have a look in the file to check that everything worked:

```
> more testfile
Hello, world
Hello again
```

The `more` command prints out a file to the screen, one screenful at a time (press space to get the next screen or q to exit). An alternative way to display a file is to use the `cat` command, which does not pause at the end of each page.

Unix commands such as `cat` and `more` can be combined with a *pipe* (given the symbol |), which takes the output of the first command and passes it to the input of the second:

```
> cat testfile | more
```

This command does the same thing as the previous `more testfile`. If a command takes input we can redirect input from a file into it.

```
> sort < testfile > testfile_sorted
```

The `sort` command sorts its input (here the contents of `testfile`) and sends the sorted lines to the output, which then is sent to create the file `testfile_sorted`.

`echo` is not usually used for making files. Text editors are useful programs for editing the contents of files. Start emacs, a text file editor, to modify the file:

```
> emacs testfile &
[1] 10735  <-- this tells you emacs is running as process 10735
```

The command `emacs` starts the emacs editor with the specified file(s) (or none if no arguments are specified). What is different here is that we started it in the *background* (with the & symbol), allowing us to carry on typing in the shell. We can bring it back to being a *foreground* process by typing

```
> fg
emacs  <-- this tells you emacs is in the foreground
```

You can no longer type commands in that shell. However you can press Ctrl+Z in the shell to *suspend* the emacs *process* (a process is a running Unix program). That means the process still exists but is frozen in its current state. If you pressed Ctrl+Z you can then type

```
> bg
```

which will allow the emacs process to carry on in the background again. If you suspend a program it will stop until it is told to continue using `bg`, `fg` or `kill`.

2 Editing the file

We can move the cursor around in emacs using the cursor keys, or clicking with a mouse. Read the separate document on using emacs for help. Typing letters will insert them at the current position of the cursor. The backspace key should delete text.

Let us make some modifications to the file. You can add the line ‘My name is blah’ to the end of the file. We can save the file either by choosing ‘File’ on the menu at the top of the window with the mouse, then selecting ‘Save (current buffer)’, or by typing ‘C-x C-s’ (that means hold down ‘Ctrl’, press x and then press Ctrl and s), or clicking on the toolbar icon for saving.

Key shortcuts in emacs are very useful to learn as they speed up things up. We can exit emacs by selecting the File menu and choosing ‘Exit emacs’, or type ‘C-x C-c’. Keyboard shortcuts are shown on the menus next to the description. Where it says M-something, the M means hold down the ‘Alt’ or ‘Meta’ key.

3 File manipulation

We can look at the contents of the current directory, which is your home directory, and includes the file you just made.

```
> ls            <-- ls is the list files command
anotherfile  testfile

> ls | more     <-- displays directory contents one screen at a time
anotherfile  testfile

> ls testfile
testfile

> ls testfile xx
ls: xx: No such file or directory
testfile

> ls -l         <-- show details about each of the files
total 4
-rw-r--r--    1 jss      users    99 Aug 31 11:27 anotherfile
-rw-r--r--    1 jss      users    25 Aug 24 22:08 testfile

^^^^^^^^^^   ^^ ^^^^^   ^^^^^^  ^^^^ ^^^^^^^^^^^^ ^^^^^^^^^^^
Permissions  ln user    group   size  mod. date    filename
```

`ls` shows the contents of a directory or shows you the files that exist from a list of files. `-l` is an *option*, flag or switch which modifies the behaviour of `ls` to print a long listing (with more detail).

Many Unix commands have options (which almost always start with a `-` sign), and you can often use more than one option on a command. Some commands require options to come before the filenames on a command line. Many newer versions of commands (especially those created by the GNU project) have long-options, which start with `--`, e.g. `ls --size` in Linux. There are often short and long versions of the same option.

The output from `ls -l` shows the *permissions* of the file (who can access it), the number of *links* (I will not explain that here), the *user* who owns it, the *group* which has access to it, the size of the file in bytes or characters, the date and time it was last modified, and its name.

We can copy a file with `cp`:

```
> cp testfile testfile2
> more testfile2
Hello, world
Hello again
```

Delete a file with `rm` (remove):

```
> rm testfile2
```

Note you do not get any warning when you do this! You can put

```
alias rm "rm -i"
```

in your `.mytcshrc` settings file (see §7 below) to stop you deleting things by accident.

Rename a file with `mv` (move):

```
> cp testfile testfile2
> mv testfile2 a_great_file
```

4 The X Terminal

You are currently using an X desktop (see the *Basic Unix Principles* notes for details). The window you are typing into is called an X-Terminal (or `xterm` for short).

Feel free to start another terminal with the `xterm &` command. Note that the terminal you get looks different because there are actually several different programs which act as a terminal. `xterm`, the original, is old. If you are using the Gnome desktop environment, use `gnome-terminal` instead. KDE has the `konsole` terminal.

Opening more than one terminal is very useful and allows you to have several programs running, or several directories open, at once. New terminal programs also let you open up several tabs inside a terminal window.

You can also scroll each terminal with `shift+pageup` and `shift+pagedown` or using the scroll bar. Terminal programs usually remember a certain number of lines of history (see the program settings to change this).

You can also copy and paste text in the terminal. Copy and paste are unfortunately a bit different on Unix compared to other platforms. The easiest way to copy simple text in Unix is to select some text with the left mouse button (click and drag right), and then paste with the middle mouse button.

Try this out! You can also select ‘words’ and filenames with a left double-click. This copy and paste operation also works in emacs.

Many modern programs also allow you to copy and paste using the menu bar or the standard `ctrl+c` and `ctrl+v` keyboard shortcuts.

5 Keys, tab completion and command history

Useful keys to use in a shell are the left and right cursor keys, `ctrl+a` to move to the start of the line, `ctrl+e` to move to the end, and `alt+b` and `alt+f` to move back and forward ‘words’. `ctrl+d` deletes the character under the cursor and backspace deletes the previous one. These movement keys also work in emacs (that’s why they are known as the ‘emacs keybindings’. Note that if you are using gnome-terminal, you will need to turn off ‘Enable menu access keys’ to use `alt+f`.

Shells have a great feature called filename completion, where you do not have to type in a full filename. If you press the `tab` key after type a few letters, the shell will try to work out what the rest of the filename is based on what files exist. If there are several possible files, it will complete the filename until the name is ambiguous.

Pressing `ctrl+d` in tcsh lists the possible filenames which so far match the partial filename being completed (care - this closes a shell if there is nothing on the line).

Have a go at typing `ls tes` and pressing `tab` to see what is matched (as long as there is testfile in the current directory). The `tab` key also completes partial commands (try `gnome-term`[tab]).

The shell also remembers the most recent commands that you have typed in. You can recall them with the up and down cursor keys (repeatedly press up to get older commands). You can press enter to run the same command again or edit it with the cursor keys, delete, etc... The `history` command lists out the commands you have typed.

Another occasionally useful feature is to use an exclamation command to recall a command of the same type. For instance if we do:

```
> man tcsh          <-- look at man page for tcsh
> ls *.txt
> !man              <-- recalls man tcsh
> !ls               <-- recalls ls *.txt
```

Take care doing this for dangerous commands like rm (memory often fails us!). They also don’t work in scripts. You can actually do quite a bit more with !, but as I don’t bother to remember how it works I won’t tell you.

6 Wildcards

If a command can take a list of files, then it is possible to avoid typing them by using a *wildcard* which matches the names of files. For example, suppose we have the files testfile1, testfile2, fred, freddy and frid, then:

```
> ls testfile*  # same as typing 'ls testfile1 testfile2'
testfile1   testfile2
> ls fred*      # same as typing 'ls fred freddy'
fred  freddy
> ls *es*1
```

```
testfile1
> ls fr?d
fred  frid
> ls fr[ei]d
fred  frid
> ls testfile* fred*
testfile1  testfile2  fred  freddy
```

`*` will match any number of following characters (including none), `?` matches a single character (but not none), and sets of matching characters can be put between square brackets. Wildcard expressions are expanded so that the program gets a list of normal filenames. Experiment with wildcards as they are invaluable!

7 Hidden files

If we make a file with a name starting with a dot '.' then it will not show up using the normal ls command. Dot-files (as they are known) are used to store settings (preferences files) typically in your home directory.

Important dot-files include `.mytcshrc` which is a script which is run every time you open a new tcsh shell. `.emacs` is a file which holds emacs settings. `.Xdefaults` contain settings for older Unix graphical programs.

```
> ls -a       <-- list all files, including hidden files
.  ..  .emacs  .mytcshrc
```

In every directory there are also two hidden directories, '.' and '..'. See the next section for details.

8 Directories

Directories are organisational containers for files (also known as 'folders' on other operating systems). Directories are a good way to organise files that have a particular purpose, or belong to a particular project. It is generally a good idea to keep your home directory fairly empty of files, storing them subdirectories (a subdirectory is a directory inside a directory).

Directories have parents – the directory that directory is contained within. They can also have children – directories within that directory. Every directory has the root directory (/) as an ancestor.

Let us start by making a directory:

```
> pwd               <-- shows current directory
/home/username
> mkdir testdir     <-- make a directory
> cd testdir        <-- change to directory
> pwd
/home/username/testdir
> ls                <-- nothing here yet
> touch hello       <-- makes a blank file called hello
> ls
hello
```

```
> ls -a             <-- show hidden files
.  ..   hello
```

Various ways to change to this new directory are

```
> cd                <-- takes you to /home/username
> cd testdir        <-- takes you to /home/username/testdir
```

or any of these

```
> cd /home/username/testdir <-- absolute path
> cd ~username/testdir      <-- equivalent
> cd ~/testdir              <-- equiv. if your are username
```

or

```
> cd /home
> cd username/testdir
```

or even

```
> cd /                      <-- go to root directory
> cd home                   <-- now in /home dir
> cd username               <-- now in /home/username
> cd testdir                <-- now in /home/username/testdir
```

We can refer to a file in another directory by building up a *path*, a list of directories which are inside each other, and then the filename, e.g.

```
> cd /home/jss
> more testdir/hello        <-- shows hello in /home/jss/testdir
> more t1/t2/hello          <-- shows hello in /home/jss/t1/t2
> more /home/jss/test       <-- shows test in /home/jss
> cd /
> more home/jss/test        <-- shows test in /home/jss
> cd /home
> more jss/test             <-- shows test in /home/jss
```

An *absolute path* is one which starts from the root directory, '/', e.g. `/home/username/test.txt`.

In every directory exist two other directories, the '.' directory which is an alias of that directory, and '..' which is an alias of the directory that directory is in. For example:

```
> cd /home/username/testdir
> pwd
/home/username/testdir
> cd .
> pwd
/home/username/testdir      <-- the same!!
> cd ..
> pwd
/home/username              <-- moves ''up'' one
> cd testdir/..
> pwd
/home/username              <-- we went nowhere!
```

‘..’ and ‘.’ are useful to refer to files and directories which exist ‘up’ the tree from the current directory, for example:

```
> cd /home/jss/testdir
> more ../testfile         <-- shows testfile in /home/jss
> ls -ld .                 <-- shows info about curr. dir.
```

The `cd` command has a clever trick! If we type `cd -` we get the same directory as we were in before the previous `cd`.

```
> cd /home/user
> cd testdir/fred
> cd -
> pwd
/home/user
```

Life is much easier if we change the shell prompt (the > symbol) to include the name of the current directory. We could add the line

```
set prompt="%m:%c02> "
```

to your `.mytcshrc` file (prompt is an environment variable, see §13). When you restart your shell, your prompt will show the computer you are on (`%m`) and the directory you are in, relative to your home directory (`%c02`, the `02` says the maximum number of subdirectories to show).

An alternative shell prompt is

```
set prompt="%m:%/> "
```

which shows the entire path of your current directory from the root directory. Another quite good one is to put the following in your `.mytcshrc` file

```
set prompt="%m:%c02> "
if( $TERM == xterm ) then
    set prompt="%{\033]0;%n@%m:%~\007%}$prompt"
endif
```

Which puts your current working directory on the title of your xterms (read the manual to understand how this works).

Directories show up like this on `ls -l`:

```
drwxr-xr-x    2 jss      jss      4096 Sep 15 14:46 testdir
```

If we do `ls testdir`, this shows the *contents* of the directory, not its name. This can be avoided with the `ls -d` option. The `-F` flag on `ls` uses a special ’/’ character after each directory to enable you to distinguish them (and scripts):

```
> ls -F
normalfile  script*  testdir2/
```

9 Permissions

Files in Unix have permissions which say who is allowed to do what with them. It is a bad idea, for instance, to let people delete files in other peoples' home directories. Each file and directory has associated an *owner* and a *group*, which can be seen using `ls -l`. A group is a list of users. All users are in the group called users. The owner of the file can modify the permissions allowed for the owner, group and other people.

Permissions are shown by `ls -l` at the front of the listing, with a format like `-rwxr--r--`. The first character is a `d` if the file is a directory. The next three characters correspond to the permissions of the owner. `x` means that the owner can run this as a script or program, `r` means that the owner can read the file, and `w` means that the owner can write to the file. The following three characters are the permissions of members of the group. In the example above, the group can read the file, but not write nor run it. The next three characters are the permissions of other people, here they can read the file.

If you are the owner of the file or directory you can modify the permissions to disallow or allow access to the file by other people (see `man chmod` for more detail):

```
chmod o-r file   # ‘‘others’’ cannot read the file
chmod g-r file   # ‘‘group’’ cannot read the file
chmod og-r file  # group and others cannot read file
chmod og+r file  # group and others can read file
chmod uog+x file # everyone can run the file as a program
chmod -w file    # write protect file to help prevent overwrite
```

10 Symbolic links

A link is like another name for a file or directory, so you can access it in an easier way. There are two sorts of links "hard links" and "soft links", but I won't cover hard links here. Suppose we are in your home directory.

```
> pwd
/home/username
> echo 'hi there' > out.txt   <-- make out.txt
> more out.txt
hi there
> ln -s out.txt wibble        <-- make a link wibble to out.txt
> more wibble                 <-- wibble 'points to' out.txt
hi there
> echo 'more text' >> out.txt <-- add to out.txt
> more wibble
hi there
more text
> rm wibble                   <-- delete the link
> more out.txt                <-- original still exists
hi there
more text
```

We can make a link to directories even, and files in other directories. This can make a convenient shortcut. e.g.

```
> cd                          <-- goes to home directory
> ln -s /data/fred/testdir testdir
> cd testdir                  <-- goes to /data/fred/testdir
> cd ˜/testdir                <-- goes to /data/fred/testdir
```

11 Backticks

Backticks are a particularly useful way for inserting the result of one command into the command line of another. Backticks look like apostrophes, but curve the other way. The output from a command is used to replace the command within the backticks. For example

```
> cat ‘ls -rt | tail -1‘
```

prints out the last modified file to the screen (the `ls` command lists the files in reverse date order, and `tail` selects the last line).

12 Aliases

You can easily make shortcuts for commands with an alias in your shell. Quite often this is the *wrong* way to do things as you won't be able to access the alias in another shell or in a Perl script. Other options to consider are symlinks (to make going to other directories quicker) and scripts (often put in your own personal `/home/username/bin` directory so you get them automatically). You can put aliases in your ∼`/.mytcshrc` file to get them the next time you start a shell. The format for an alias definition in tcsh is `alias new "old"`. For example

```
alias rm "rm -i"   <-- ask for confirmation
```

Makes every `rm` command you type in become `rm -i`. Only commands, not files are aliased. For instance `rm rubbish*` becomes `rm -i rubbish*`. This can get quite confusing if used too often or inappropriately (don't try alias ls as 'rm'). You can also do things like

```
alias lsl "ls -lrt"
```

which makes `lsl` show your files in long time order, latest last.

13 Enviroment variables

The shell you are using, `tcsh`, has a list of settings and other information stored as *variables* and *environment variables*. Unfortunately `tcsh` has both of these types of variables, rather than a single set. `bash/sh` only has environment variables.

Variables are used within scripts and they can also affect the behaviour of the shell (e.g. the shell prompt is stored in the `prompt` variable). Environment variables are passed to programs run from the shell — they can be useful for sending information to a program. Special environment variables can affect standard Unix behaviour (e.g. PATH).

In the `tcsh` shell you can list the variables using the `set` command without any arguments:

```
> set               <-- list all variables
...
uid     914
user    jss
version tcsh 6.17.00 (Astron) 2009-07-10 (x86_64-u...
> set name=fred     <-- set variable name to fred
> echo $name        <-- print out contents of name variable
> echo ${name}_foo  <-- prints fred_foo
```

The `$name` or `${name}` syntax is called expansion of a variable. The shell replaces it with the contents of the variable before running the command.

Environment variables are used similarly. They are typically given upper-case names which helps to distinguish them from variables.

```
> setenv              <-- list all environment variables
...
> setenv VAR something <-- set an environment variable
> echo $VAR           <-- use an environment variable
```

Special environment variables include `PATH` which is a list of directories to search for commands in, separated by colons. For instance you can add a directory to the list using

```
> setenv PATH /home/username/bin:${PATH}
```

which makes `/home/username/bin` searched before all the other directories on the PATH. Another environment variable you may meet is `LD_LIBRARY_PATH` which tells the system where to look for shared libraries when running programs.

14 Quoting

You cannot pass certain characters easily to commands because they are special to the shell (e.g. round brackets, dollars, semicolons, spaces, ampersands). If you want to use these characters they must be *quoted* or *escaped*. Many characters can be preceeded by a backslash character to be ignored by the shell, e.g.

```
> echo ; echo    <-- semicolon separates commands


> echo \; echo   <-- semicolon is quoted
; echo
```

Another way to quote special characters is to use single or double quotes around them. The difference between double and single quotes is that environment variables are not expanded inside single quotes.

```
> echo "Hello $name"
Hello fred
> echo 'Hello $name'
Hello $name
```