Shell

The Linux/Unix shell refers to a special program that allows you to interact with it by entering certain commands from the keyboard; the shell will execute the commands and display its output on the monitor. The environment of interaction is text-based (unlike the GUI-based interaction we have been using in the previous chapters) and since it is command-oriented this type of interface is termed Command Line interface or CLI. Before the advent of GUI-based computing environments, the CLI was the only way that one can interact and access a computer system.

Up until now, there was never a need to type commands into a shell; and with the modernization and creation of a lot of newer GUI-based tools, the shell is becoming increasingly un-required to perform many tasks. But that said, the shell is a very powerful place, and a lot is achieved through it. A lot of the front-end GUI methods of doing things have similar ways and means to get done with using the shell. Professional Linux and UNIX users find the shell very powerful, and an introduction to at least the basic shell usage is useful.

The bash shell

The *bash* shell is one of several shells available for Linux. It is also called the *B*ourne-*a*gain *sh*ell, after Stephen Bourne, the creator of an earlier shell (*/bin/sh*).
Bash is substantially compatible with sh, but provides many improvements in both function and programming capability. It incorporates features from the Korn shell (ksh) and C shell (csh), and is intended to be a POSIX-compliant shell.
Before we delve deeper into bash, recall that a *shell* is a program that accepts and executes commands. It also supports programming constructs allowing complex commands to be built from smaller parts. These complex commands, or *scripts*, can be saved as files to become new commands in their own right. Indeed, many commands on a typical Linux system **are** scripts.

Getting to a Shell

Since it is most likely that you are in the graphical desktop environment now, the underlying shell that is available is not displayed. To access a shell, try the following key combination,

Control + Alt + F1

Where F1 can be replaced by F2, F3, and so on. The graphical desktop tends to run in F7 or F8, so to go back to your graphical desktop screen, just hit Control + Alt + F7. These are virtual terminals.

Alternatively, you could get to a Terminal application, so you can have a shell while your in the graphical desktop environment (this is much preferred, and will be used throughout this Chapter). To do this, go to:

Main Menu --> System Tools --> Terminal

Or right-click on the desktop, and click on the Open Terminal option. This terminal is equivalent to the virtual terminals mentioned earlier, except now you don't have to switch screens - you can just minimize or maximize the terminal (or if you're done, you can close it).

Shells use three standard I/O *streams*:
• *stdin* is the *standard input stream*, which provides input to commands.
• *stdout* is the *standard output stream*, which displays output from commands.
• *stderr* is the *standard error stream*, which displays error output from commands.
Input streams provide input to programs, usually from terminal keystrokes. Output streams print text characters, usually to the terminal. The terminal was originally an ASCII typewriter or display terminal, but it is now more often a window on a graphical desktop. More detail on how to redirect these standard I/O streams is covered in a later section of this tutorial, Streams, pipes, and redirects. The rest of this section focuses on redirection at a high level.
If you are using a Linux system without a graphical desktop, or if you open a terminal window on a graphical desktop, you will be greeted by a prompt, perhaps like one shown in Listing 1.

Listing 1. Some typical user prompts
```
[db2inst1@echidna db2inst1]$
ian@lyrebird:~>
myname@hostname ~ $
```
If you log in as the root user (or superuser), your prompt may look like one shown in
Listing 2.

Listing 2. Superuser, or root, prompt examples
```
[root@echidna root]#
lyrebird:~ #
hostname username #
```
The root user has considerable power, so use it with caution. When you have root privileges, most prompts include a trailing pound sign (#). Ordinary user privileges are usually delineated by a different character, commonly a dollar sign ($). Your actual prompt may look different than

the examples in this tutorial. Your prompt may include your user name, hostname, current directory, date or time that the prompt was printed, and so on.

Commands and sequences

So now that you have a prompt, let's look at what you can do with it. The shell's main function is to interpret your commands so you can interact with your Linux system. On Linux (and UNIX) systems, commands have a *command name*, and then *options* and *parameters*. Some commands have neither options nor parameters, and some have options but no parameters, while others have no options but do have parameters.
If a line contains a # character, then all remaining characters on the line are ignored. So a # character may indicate a comment as well as a root prompt.

Echo

The echo command prints (or echoes) its arguments to the terminal as shown in Listing 3.

Listing 3. Echo examples

```
[ian@echidna ian]$ echo Word
Word
[ian@echidna ian]$ echo A phrase
A phrase
[ian@echidna ian]$ echo Where are my spaces?
Where are my spaces?
[ian@echidna ian]$ echo "Here are my spaces." # plus comment
Here are my spaces.
```

In third example of Listing 3, all the extra spaces were compressed down to single spaces in the output. To avoid this, you need to *quote* strings, using either double quotes (") or single quotes ('). Bash uses *white space*, such as blanks, tabs, and new line characters, to separate your input line into *tokens*, which are then passed to your command. Quoting strings preserves additional white space and makes the whole string a single token. In the example above, each token after the command name is a parameter, so we have respectively 1, 2, 4, and 1 parameters.

Bash shell metacharacters and control operators

Bash has several *metacharacters*, which, when not quoted, also serve to separate input into words. Besides a blank, these are '|', '&', ';', '(', ')', '<', and '>'. We will discuss some of these in more detail in other sections of this tutorial. For now, note that if you want to include a metacharacter as part of your text, it must be either quoted or escaped using a backslash (\) as shown in Listing 4.

Listing 4. using echo with metacarachters

```
[ian@echidna ian]$ echo me \& my sister
me & my sister
```

The new line and certain metacharacters or pairs of metacharacters also serve as *control operators*. These are '||', '&&', '&', ';', ';;', '|" '(', and ')'. Some of these control operators allow you to create *sequences* or *lists* of commands.
The simplest command sequence is just two commands separated by a semicolon (;). Each command is executed in sequence. In any programmable environment, commands return an indication of success or failure; Linux commands usually return a zero value for success and a non-zero in the event of failure. You can introduce some conditional processing into your list using the && and || control operators. If you separate two commands with the control operator && then the second is executed if and only if the first returns an exit value of zero. If you separate the commands with ||, then the second one is executed only if the first one returns a non-zero exit code. Listing 5 shows some command sequences using the echo command. These aren't very exciting since echo returns 0, but you will see more examples later when we have a few more commands to use.

Listing 5. Command sequences

```
[ian@echidna ian]$ echo line 1;echo line 2; echo line 3
line 1
line 2
line 3
[ian@echidna ian]$ echo line 1&&echo line 2&&echo line 3
line 1
line 2
line 3
[ian@echidna ian]$ echo line 1||echo line 2; echo line 3
line 1
line 3
```

Exit

You can terminate a shell using the exit command. You may optionally give an exit code as a parameter. If you are running your shell in a terminal window on a graphical desktop, your window will close. Similarly, if you have connected to a remote system using ssh or telnet (for example), your connection will end. In the bash shell, you can also hold the **Ctrl** key and press the **d** key to exit.
Let's look at another control operator. If you enclose a command or a command list in parentheses, then the command or sequence is executed in a sub shell, so the exit command exits the sub shell rather than exiting the shell you are working in.

Environment variables

When you are running in a bash shell, many things constitute your *environment*, such as the form of your prompt, your home directory, your working directory, the name of your shell, files that you have opened, functions that you have defined, and so on. Your environment includes

many *variables* that may have been set by bash or by you. The bash shell also allows you to have *shell variables*, which you may *export* to your environment for use by other processes running in the shell or by other shells that you may from the current shell.
Both environment variables and shell variables have a *name*. You reference the value of a variable by prefixing its name with '$'. Some of the common bash environment variables that you will encounter are shown in Table 4.

Name	**Function**
USER	The name of the logged-in user
UID	The numeric user id of the logged-in user
HOME	The user's home directory
PWD	The current working directory
SHELL	The name of the shell
$	The process id (or *PID* of the running bash shell (or other)process

Listing 7 shows what you might see in some of these common bash variables.

Listing 7. Environment and shell variables

```
[ian@echidna ian]$ echo $USER $UID
ian 500
[ian@echidna ian]$ echo $SHELL $HOME $PWD
/bin/bash /home/ian /home/ian
```

Writing a Simple Script

1. Create a new file with your text editor for example nano.
2. Give the file the extension of .sh or .bash (this is not necessary but it's good to perform to let others know it’s a bash script).
3. Write you Commands in the file and save it.
4. while you are in the same directory as the script itself you can run it using the following commands:
   ./scriptname.sh
   bash scriptname.sh
   sh scriptname.sh

Command history

If you are typing in commands as you read, you may notice that you often use a command many times, either exactly the same, or with slight changes. The good news is that the bash shell can maintain a *history* of your commands. By default, history is on. You can turn it off using the command **set +o history** and turn it back on using **set -o history**. An environment variable called HISTSIZE tells bash how many history lines to keep. A number of other settings control how history works and is managed. See the bash man pages for full details.
Some of the commands that you can use with the history facility are:

history

Displays the entire history

history *N*

Displays the last *N* lines of your history

history -d *N*
Deletes line *N* form your history; you might do this if the line contains a password, for example

which

which [*options*] [**--**] [*commands*]

List the full pathnames of the files that would be executed if the named *commands* had been run. **which** searches the user's PATH environment variable.

Man pages

Our final topic in this section of the tutorial tells you how to get documentation for Linux commands through manual pages and other sources of documentation.

Manual pages and its sections
The primary (and traditional) source of documentation is the *manual pages*, which you can access using the man command. Figure 1 illustrates the manual page for the man command itself. Use the command man man to display this information.
• A heading with the name of the command followed by its section number in parentheses
• The name of the command and any related commands that are described on the same man page
• A synopsis of the options and parameters applicable to the command
• A short description of the command
• Detailed information on each of the options
You might find other sections on usage, how to report bugs, author information, and a list of any related commands. For example, the man page for man tells us that
related commands (and their manual sections) are:
apropos(1), whatis(1), less(1), groff(1), and man.conf(5).
You may have noticed that man has many options for you to explore on your own. For now, let's take a quick look at some of the "See also" commands related to man.
See also
Two important commands related to man are whatis and apropos. The whatis command searches man pages for the name you give and displays the name information from the appropriate manual pages. The apropos command does a keyword search of manual pages and lists ones containing your keyword. Listing 6 illustrates these commands.
Listing 6. Whatis and apropos examples
[ian@lyrebird ian]$ whatis man
man (1) - format and display the on-line manual pages
man (7) - macros to format man pages

man [manpath] (1) - format and display the on-line manual pages
man.conf [man] (5) - configuration data for man
[ian@lyrebird ian]$ whatis mkdir
mkdir (1) - make directories
mkdir (2) - create a directory
[ian@lyrebird ian]$ apropos mkdir
mkdir (1) - make directories
mkdir (2) - create a directory
mkdirhier (1x) - makes a directory hierarchy

The man command pages output onto your display using a paging program. On most Linux systems, this is likely to be the less program. Another choice might be the older more program. If you wish to print the page, specify the -t option to format the page for printing using the groff or troff program.
The less pager has several commands that help you search for strings within the displayed output. Use man less to find out more about / (search forwards),?(search backwards), and *n* (repeat last search), among many other commands.

4. Input/Output Redirection and Pipes

Running a command by itself with a lot of output doesn't seem all that useful. For instance, if there are many files in a directory, running a command to list the directory like,

$ ls /usr/bin

will result in about 2100 lines being displayed on the screen! To actually get any useful information out of it, you might want to dump the output of the ls command to a file; or maybe use a utility like less to view it. All this is possible thanks to input/output redirection and pipes.

Input redirection is performed using < or <<, while output redirection is done via > or >>. A point to note is that when using >, it just recreates the file, even if the same filename exists, while >> concatenates the output to the same file, causing it to possibly be double in size (if its the same output).

A pipe ("|") is used to pass the output of the command not to a file, or to the screen, but to the next utility.

Pipes can be nested, so you can pass the data through several utilities before you can get the useful information that you want. Let's dive into some examples!

1. $ ls /usr/bin >> usr.bin

2. $ wc -l usr.bin
3. 2171 usr.bin
4. $ ls /usr/bin >> usr.bin
5. $ wc -l usr.bin
6. 4342 usr.bin
7. $ ls /usr/bin > usr.bin
8. $ wc -l usr.bin
9. 2171 usr.bin

Note: the line numbers are added for clarity, and are not included in the shell output!

In line 1, the output of the directory listing of /usr/bin gets placed in a file called usr.bin. On line 2, a new utility called 'wc' is used (this is used to print the number of lines in the file (as it gets passed the -l option) - its output is at line 3. The same command is then repeated on line 4, and now, the file is double the size as per line 6! That is because the >> output redirection was used, which has concatenated the two outputs together. Notice that in line 7, a single > is used, and in line 9, it shows that the file has been over-written with the new contents.

$ ls /usr/bin | grep cancel

cancel

cancel.cups

The above is an example of how a pipe is used. After listing the files, the output is passed on to a utility called grep (which basically searches for a pattern, and prints the output) and the string being searched for is "cancel". It comes back with two matches. Similarly, a command like:

$ ls /usr/bin | less

Will place the output of the directory listing into the less pager so that it can be scrolled through easily. And for another example as to how pipes can be nested, issuing:

$ ping 192.168.1.254 > pingresult

$ more pingresult | grep time|wc -l

19

sends the output of the directory listing of pingresult to grep, which then searches for the string "time", and then wc prints how many times it occurs in lines.

Wc, head, and tail

Cat and tac display the whole file. That's fine for small files like our examples, but suppose you have a large file. Well, first you might want to use the wc (*W*ord *C*ount) command to see how big the file is. The wc command displays the number of lines, words, and bytes in a file. You can also find the number of bytes by using ls -l. Listing 7 shows the long format directory listing for our two text files, as well as the output from wc.

Listing 7. Using wc with text files

```
[ian@echidna lpi103]$ ls -l text*
-rw-rw-r-- 1 ian ian 24 Sep 23 12:27 text1
-rw-rw-r-- 1 ian ian 25 Sep 23 13:39 text2
[ian@echidna lpi103]$ wc text*
3 6 24 text1
3 6 25 text2
6 12 49 total
```

Options allow you to control the output from wc or to display other information such as maximum line length. See the man page for details.

Two commands allow you to display either the first part (*head*) or last part (*tail*).

These commands are the head and tail commands. They can be used as filters, or they can take a file name as an augument. By default they display the first (or last) 10 lines of the file or stream. Listing 8 uses the dmesg command to display bootup messages, in conjunction with wc, tail, and head to discover that there are 177 messages, then to display the last 10 of these, and finally to display the six messages starting 15 from the end. Some lines have been truncated in this output.

Listing 8. Using wc, head, and tail to display boot messages

```
[ian@echidna lpi103]$ dmesg | wc
177 1164 8366
[ian@echidna lpi103]$ dmesg | tail
i810: Intel ICH2 found at IO 0x1880 and 0x1c00, MEM 0x0000 and ...
i810_audio: Audio Controller supports 6 channels.
i810_audio: Defaulting to base 2 channel mode.
i810_audio: Resetting connection 0
ac97_codec: AC97 Audio codec, id: ADS98 (Unknown)
i810_audio: AC'97 codec 0 Unable to map surround DAC's (or ...
i810_audio: setting clocking to 41319
Attached scsi CD-ROM sr0 at scsi0, channel 0, id 0, lun 0
sr0: scsi3-mmc drive: 0x/32x writer cd/rw xa/form2 cdda tray
Uniform CD-ROM driver Revision: 3.12
[ian@echidna lpi103]$ dmesg | tail -n15 | head -n 6
agpgart: Maximum main memory to use for agp memory: 941M
agpgart: Detected Intel i845 chipset
agpgart: AGP aperture is 64M @ 0xf4000000
Intel 810 + AC97 Audio, version 0.24, 13:01:43 Dec 18 2003
PCI: Setting latency timer of device 00:1f.5 to 64
```

Another common use of tail is to *follow* a file using the -f option, usually with a line count of 1. You might use this when you have a background process that is generating output in a file and you want to check in and see how it is doing. In this mode, tail will run until you cancel it (using **Ctrl-c**), displaying lines as they are written to the file.

Sort and uniq

The sort command sorts text files

Listing 9 illustrates using the sort command to sort our text file. Since the sort order is by character, you might be surprised at the results. Fortunately, the sort command can sort by numeric values or by character values.

Listing 9. Character and numeric sorting

```
[ian@echidna lpi103]$ sort text1 > sorted| more sorted
1 apple
1 apple
2 pear
3 banana
3 banana
9 plum
```

Notice that we still have two lines containing the fruit "apple". Another command called uniq gives us additional control over the elimination of duplicate lines. The uniq command normally operates on sorted files. Listing 10 applies the uniq command on the file that was sorted in listing 9

Listing 34. Using uniq

```
[ian@echidna lpi103]$ uniq sorted > nodups |more nodups
1 apple
3 banana
2 pear
9 plum
```

Our sort was by collating sequence, so uniq gives us only one of the "1 apple" lines.

Wildcards and Globbing

Often, you may need to perform a single operation on many filesystem objects, without operating on the entire tree as we have just done with recursive operations.

For example, you might want to find the modification times of all the text files, without listing the split files. Although this is easy with our small directory, it is much harder in a large filesystem.

To solve this problem, use the wildcard support that is built in to the bash shell. This support, also called "globbing" (because it was originally implemented as a program called /etc/glob), lets you specify multiple files using wildcard pattern.

A string containing any of the characters '?', '*' or '[', is a *wildcard pattern*. Globbing is the process by which the shell (or possibly another program) expands these patterns into a list of pathnames matching the pattern. The matching is done as follows.
? matches any single character.
***** matches any string, including an empty string.
[introduces a *character class*. A character class is a non-empty string, terminated by a ']'. A match means matching any single character enclosed by the brackets. There are a few special considerations.
• The '*' and '?' characters match themselves. If you use these in filenames, you will need to be really careful about appropriate quoting or escaping.
• Since the string must be non-empty and terminated by ']', you must put ']' **first** in the string if you want to match it.
• The '-' character between two others represents a range which includes the two other characters and all between in the collating sequence. For example, [0-9a-fA-F] represents any upper or lower case hexadecimal digit. You can match a '-' by putting it either first or last within a range.
• The '!' character specified as the first character of a range complements the range so that it matches any character except the remaining characters. For example [!0-9] means any character except the digits 0 through 9. A '!' in any position other than the first matches itself. Remember that '!' is also used with the shell history function, so you need to be careful to properly escape it.
Globbing is applied separately to each component of a path name. You cannot match a '/', nor include one in a range. You can use it anywhere that you might specify multiple file or directory names, for example in the ls, cp, mv or rm commands. In Listing 11, we first create a couple of oddly named files and then use the ls and rm commands with wildcard patterns.
Listing 11. Wildcard pattern examples

```
[ian@echidna lpi103]$ ls
backup text1 text2 text3 text4 text5 text6
 [ian@echidna lpi103]$ ls text[2-4]
text2 text3 text4
[ian@echidna lpi103]$ ls text[!2-4]
text1 text5 text6
[ian@echidna lpi103]$ echo text*>text10
[ian@echidna lpi103]$ ls
backup text1 text10 text2 text3 text4 text5 text6
[ian@echidna lpi103]$ rm text1*
[ian@echidna lpi103]$ ls
backup text2 text3 text4 text5 text6
[ian@echidna lpi103]$ ls backup/*2
backup/text1.bkp.2
[ian@echidna lpi103]$ ls -d .*
. ..
```

Notes:
1. The pattern '*[!2-4]' matches the longest part of a name that does not have 2, 3, or 4 following it.
2. If a filename starts with a period (.) then that character must be matched explicitly. Notice that only the last ls command listed the two special directory entries (. and ..).

The best way to understand all the various shell interactions is by practice, so try these wildcards out whenever you have a chance. Remember to try ls to check your wildcard pattern before committing it to whatever cp, mv or worse, rm might unexpectedly do for you.

touch

touch [*options*] *files*

For one or more *files*, update the access time and modification time (and dates) to the current time and date. **touch** is useful in forcing other commands to handle files a certain way; for example, the operation of **make**, and sometimes **find**, relies on a file's access and modification time. If a file doesn't exist, **touch** creates it with a file size of 0.

You may find it useful to create an empty file as a placeholder for future content. The touch command can be used without any options to create an empty file as follows:

```
$ touch /home/tclark/touch1.fil        #create file touch1.fil
$ touch touch2.fil     #create file touch2.fil

$ ls –l list the files
total 4
drwxrwxr-x 2 tclark tclark 4096 Jan 13 17:48 examples
-rw-rw-r-- 1 tclark tclark 0 Jan 13 19:13 touch1.fil
-rw-rw-r-- 1 tclark tclark 0 Jan 13 19:14 touch2.fil
```

The touch command also has an option that allows you to change the files modification date to current date and time.

```
$touch touch1.fill
$ ls -l
total 4
drwxrwxr-x 2 tclark tclark 4096 Jan 13 17:48 examples
-rw-rw-r-- 1 tclark tclark 0 Jan 10 23:30 touch1.fil
-rw-rw-r-- 1 tclark tclark 0 Jan 13 19:14 touch2.fil
```

Finding files

In the final topic for this part of the tutorial, we will look at the find command which is used to find files in one or more directory trees, based on criteria such as name, timestamp, or size.

find

The find command will search for files or directories using all or part of the name, or by other search criteria, such as size, type, file owner, creation date, or last access date. The most basic find is a search by name or part of a name. Listing 12 shows an example where we first search for all files that have either a '1' or a 'k' in their name, then perform some path searches that we will explain in the notes below.

Listing 12. Finding files by name

```
[ian@echidna lpi103]$ find -name "*[1k]*"
./text1
./f1
./backup
./backup/text1.bkp.2
./backup/text1.bkp.1
./f1a
```

Notes:

1. The patterns that you may use are shell wildcard patterns like those we saw earlier when we discussed under Wildcards and globbing.
2. If you want to find a file or directory whose name begins with a dot, such as .bashrc or the current directory (.), then you **must** specify a leading dot as part of the pattern. Otherwise, name searches will ignore these files or directories.

We can also search by file size, in Listing 13 we find all files with size 0, Note that specifying -empty instead of -size 0 also finds empty files.

Listing 13. Finding files by size

```
[ian@echidna lpi103]$ find  -size 0
./f2
./f3
./f4
./f5
./f6
./f7
```

Create, monitor, and kill processes

In this section, you learn about the following topics:

- Foreground and background jobs
- Monitoring and displaying processes
- Identifying and killing processes

If you stop and reflect for a moment, it is pretty obvious that lots of things are running on your computer, other than the terminal programs we have been running. Indeed, if you are using a graphical desktop, you may have opened more than one terminal window at one time, or

perhaps have opened a file browser, internet browser, game, spreadsheet, or other application. So far, our examples have entered commands at a terminal window. The command runs and we wait for it to complete before we do anything else. If you type the ps command which displays process status and you see that each process has a Process ID (PID).

```
$ ps

  PID TTY          TIME CMD
 3511 pts/1    00:00:00 bash
 3514 pts/1    00:00:00 ps
```

Foreground and background jobs

When you run a command in your terminal window, such as we have done to this point, you are running it in the *foreground*. Our commands to date have run quickly, but suppose we are running a graphical desktop and would like a digital clock displayed on the desktop. For now, let's ignore the fact that most desktops already have one; we're just using this as an example.

If you have the X Window System installed, you probably also have some utilities such as xclock or xeyes. Either works for this exercise, but we'll use xclock. The man page explains that you can launch a digital clock on your graphical desktop using the command

```
xclock -d -update 1
```

The -update 1 part requests updates every second, otherwise the clock updates only every minute. So let's run this in a terminal window.

```
[ian@echidna ian]$ xclock -d -update 1
```

Unfortunately, your terminal window no longer has a prompt, so we really need to get control back. Fortunately, the Bash shell has a *suspend* key, Ctrl-z. Pressing this key combination gets you a terminal prompt again as shown in Listing 14.

Listing 14. Suspending xclock with Ctrl-z

```
[ian@echidna ian]$ xclock -d -update 1
[1]+ Stopped xclock -d -update 1
[ian@echidna ian]$
```

The clock is still on your desktop, but it has stopped running. Suspending it did exactly that. In fact, if you drag another window over part of it, that part won't even redraw. You also see a terminal output message indicating "[1]+ Stopped". The 1 in this message is a *job number*. You can restart the clock by typing fg %1. You could also use the command name or part of it by using fg %xclock or fg %?clo.

Finally, if you just use fg with no parameters, you can restart the most recently stopped job, job 1 in this case. Restarting it with fg also brings the job right back to the foreground, and you no longer have a shell prompt. What you need to do is place the job in the *background*; a bg command takes the same type of job specification as the fg command

and does exactly that. Listing 15 shows how to bring the xclock job back to the foreground and suspend it using two forms of the fg command.. You can suspend it again and place it in the background; the clock continues to run while you do other work at your terminal.

Listing 15

```
[ian@echidna ian]$ fg %1
xclock -d -update 1
[1]+ Stopped xclock -d -update 1
[ian@echidna ian]$ fg %?clo
xclock -d -update 1
[1]+ Stopped xclock -d -update 1
[ian@echidna ian]$ bg
[1]+ xclock -d -update 1 &
[ian@echidna ian]$
```

Using "&"

You may have noticed that when we placed the xclock job in the background, the message no longer said "Stopped" and that it was terminated with an ampersand (&). In fact, you don't need to suspend the process to place it in the background at all. You can simply append an ampersand to the command and the shell will start the command (or command list) in the background. Let's start an analog clock with a wheat background and red hands using this method.

```
[ian@echidna ian]$ xclock -bg wheat -hd red -update 1&
[2] 5659
```

Notice that the message is slightly different this time. It represents a job number and a process id (PID). We will cover PIDs and more about status in a moment. For now, let's use the jobs command to find out what jobs are running. Add the -l option to list PIDs, and you see that job 2 indeed has PID 5659 as shown in Listing 16. Note also that job 2 has a plus sign (+) beside the job number, indicating that it is the *current job*. This job will come to the foreground if no job specification is given with the fg command.

Listing 16. Displaying job and process information

```
[ian@echidna ian]$ jobs -l
[1]- 4234 Running xclock -d -update 1 &
[2]+ 5659 Running xclock -bg wheat -hd red -update 1 &
```

When we finally get the fg command entered, bash displays the command that is now running in our shell, namely, the command list, which is still happily printing the time every two seconds.

Once we succeed in getting the job into the foreground, we can either terminate (or *kill*), or take some other action, In this case, we use Ctrl-c to terminate our 'clock'.

To stop a process while its in the background use:

$ Kill [pid]

PID is the process ID of the process you want to stop.

Standard IO and background processes

What happens to a process if it needs input from stdin?

The terminal process under which we start a background application is called the *controlling terminal*. Unless redirected elsewhere, the stdout and stderr streams from the background process are directed to the controlling terminal. Similarly, the background task expects input from the controlling terminal, but the controlling terminal has no way of directing characters you type to the stdin of a background process. In such a case, the Bash shell suspends the process, so that it is no longer executing. You may bring it to the foreground and supply the necessary input. Listing 17 illustrates a simple case where you can put a command list in the background.

After a moment, press **Enter** and the process stops. Bring it to the foreground and provide a line of input followed by Ctrl-d to signal end of input file. The command list completes and we display the file we created.

Listing 17. Waiting for stdin

```
[ian@echidna ian]$ cat  >bginput.txt&
[1] 18648
[ian@echidna ian]$ fg
cat >ginput.txt
input data
[ian@echidna ian]$ cat bginput.txt
input data
```

Process status

In the previous part of this section, we had a brief introduction to the jobs command and saw how to use it to list the Process IDs (or PIDs) of our jobs.

ps

There is another command, the ps command, which we use to display various pieces of process status information. Remember "ps" as an acronym for "process status". The ps command accepts zero or more PIDs as argument and displays the associated process status. Use of the ps command is shown in Listing 18.

Listing 18. Status of background processes

```
[ian@echidna ian]$ jobs
[1]- Running xclock –update 1 &
[2]+ Running xclock –update 1 &
[ian@echidna ian]$ ps
PID TTY STAT TIME COMMAND
21709 pts/3 SN 0:00 xclock
21719 pts/3 SN 0:00 xclock
```

This command will show all processes running in your computer:

```
$ ps ux
USER       PID %CPU %MEM    VSZ  RSS TTY      STAT START   TIME COMMAND
heyne      691  0.0  2.4 19272 9576 ?        S    13:35   0:00 kdeinit: kded
heyne      700  0.1  1.0  5880 3944 ?        S    13:35   0:01 artsd -F 10 -S 40
heyne      710  0.0  2.8 21876 11072 ?       S    13:35   0:00 kdeinit: knotify
```

```
heyne     711  0.0  0.0  1344  352 ?        S    13:35   0:00 kwrapper ksmserve
heyne     713  0.0  2.4 18900 9304 ?        S    13:35   0:00 kdeinit: ksmserve
heyne     714  0.0  2.9 21548 11528 ?       S    13:35   0:00 kdeinit: kwin -se
heyne     715  0.3  4.8 31096 18820 ?       S    13:35   0:03 /usr/lib/mozilla/
heyne     719  0.1  3.5 22548 13680 ?       S    13:35   0:01 kdeinit: kdesktop
heyne     721  0.5  3.6 23904 14028 ?       S    13:35   0:05 kdeinit: kicker
heyne     722  0.0  2.0 18504 7824 ?        S    13:35   0:00 kdeinit: kio_file
heyne     723  0.0  4.8 31096 18820 ?       S    13:36   0:00 /usr/lib/mozilla/
heyne     724  0.0  4.8 31096 18820 ?       S    13:36   0:00 /usr/lib/mozilla/
heyne     725  0.0  4.8 31096 18820 ?       S    13:36   0:00 /usr/lib/mozilla/
heyne     729  0.0  2.4 19408 9404 ?        S    13:36   0:00 kdeinit: klaptopd
heyne     730  0.0  2.3 17044 9152 ?        S    13:36   0:00 kteatime -session
heyne     731  0.0  3.0 21236 11848 ?       S    13:36   0:00 kdeinit: kmix -se
heyne     735  0.0  4.8 31096 18820 ?       S    13:36   0:00 /usr/lib/mozilla/
heyne     736  0.0  2.7 20492 10600 ?       S    13:36   0:00 korgac --miniicon
heyne     745  0.0  2.4 19232 9528 ?        S    13:36   0:00 kalarmd --login
heyne     753  0.0  0.3  2108 1160 pts/0    S    13:36   0:00 bash
heyne     787  0.7  1.7  9520 6784 ?        S    13:50   0:00 emacs
heyne     789  0.2  0.3  2112 1164 pts/1    S    13:51   0:00 bash
heyne     794  0.0  0.4  3560 1576 pts/1    R    13:51   0:00 ps ux
```

top

If you run ps several times in a row, to see what is changing, you probably need the top command instead. It displays a continuously updated process list, along with useful summary information. See the man pages for top for full details on options, including how to sort by memory usage or other criteria.

Priorities

As we have seen in the previous section, Linux, like most modern operating systems can run multiple processes. It does this by sharing the CPU and other resources among the processes. If some process can use 100% of the CPU, then other processes may become unresponsive. When we looked at Process status in the previous section, we saw that the top command's default output was to list processes in descending order of CPU usage. Your system may have many commands that are capable of using lots of CPU.

Displaying and setting priorities

If we had a long running job such as this, we might find that it interfered with our ability (or the ability of other users) to do other work on our system. Linux and UNIX systems use a priority system with 40 priorities, ranging from -20 (highest priority) to 19 (lowest priority).

nice

Processes started by regular users usually have priority 0. The nice command displays our default priority. The ps command can also display the priority (nice, or NI, level), for example using the -l option. We illustrate this in Listing 19, where we have highlighted the nice value of 0.

Listing 19. Displaying priority information

```
[ian@echidna ian]$ nice
```

0

```
[ian@echidna ian]$ ps -l
F    S   UID     PID     PPID   C      PRI     NI     ADDR   SZ     WCHAN    TTY    TIME CMD
000 S 500       7283    7282   0      70      0      - 1103 wait4    pts/2    00:00:00    bash
000 R 500        9578    7283   0      72      0       - 784          pts/2    00:00:00     ps
```

The nice command can also be used to start a process with a different priority. You use the -n or (--adjustment) option with a positive value to increase the priority value and a negative value to decrease it. Remember that processes with the lowest priority value run at highest scheduling priority, so think of increasing the priority value as being *nice* to other processes. Note that you usually need to be the superuser (root) to specify negative priority adjustments. In other words, regular users can usually only make their processes nicer.

Listing 20. Using nice to set priorities

```
[ian@echidna ian]$ nice –n 19 xclock –update 1 &
```

Changing priorities

renice

If you happen to start a process and realize that it should run at a different priority, there is a way to change it after it has started, using the renice command. You specify an absolute priority (and not an adjustment) for the process or processes to be changed as shown in Listing 21.

Listing 21. Using renice to change priorities

```
[ian@echidna ian]$ renice +10 2244
```

While 2244 is the process ID of the process you want to change its priority.

You can find more information on nice and renice in the man pages.

File editing with vi

Using vi

The vi editor is almost certainly on every Linux and UNIX system. In fact, if a system has just one editor, it's probably vi, so it's worth knowing your way around in vi. This section introduces you to some basic vi editing commands, but for a full vi tutorial, check out or consult the man pages or one of the many excellent books that are available.

Starting vi

Most Linux distributions now ship with the vim (for **V**i **IM**proved) editor rather than classic vi. Vim is upward compatible with vi and has a graphical mode available (gvim) as well as the standard vi text mode interface. The vi command is usually an alias or symbolic link to the vim program. To start vi type vi followed by a filename.

vi modes

The vi editor has two modes of operation:

Command mode

In command mode, you move around the file and perform editing operations such as searching for text, deleting text, changing text, and so on. You usually start in command mode.

Insert mode

In insert mode, you type new text into the file at the insertion point. To return to command mode, press the **Esc** (Escape) key.
These two modes determine way the editor behaves. Vi dates from the time when not all terminal keyboards had cursor movement keys, so everything you can do in vi can be done with the keys typically found on a standard typewriter plus a couple of keys such as **Esc** and **Insert**. However, you can configure vi to use additional keys if they are available; most of the keys on your keyboard do something useful in vi.
Because of this legacy and the slow nature of early terminal connections, vi has a well-deserved reputation for using very brief and cryptic commands.

Getting out of vi

One of the first things I like to learn about a new editor is how to get out of it before I do anything I shouldn't have done. The following ways to get out of vi include saving or abandoning your changes, or restarting from the beginning. If these commands don't seem to work for you, you may be in insert mode, so press **Esc** to leave insert mode and return to command mode.

:q!

Quit editing the file and abandon all changes. This is a very common idiom for getting out of trouble.

:w!

Write the file (whether modified or not). Attempt to overwrite existing files or read-only or other unwriteable files. You may give a filename as a parameter, and that file will be written instead of the one your started with. It's generally safer to omit the ! unless you know what you're doing here.

ZZ

Write the file if it has been modified. Then exit. This is a very common idiom for normal vi exit.

:e!

Edit the current disk copy of the file. This will reload the file, abandoning changes you have made. You may also use this if the disk copy has changed for some other reason and you want the latest version.

:!

Run a shell command. Type the command and press **Enter**. When the command completes, you will see the output and a prompt to return to vi editing.

Notes:

1. When you type the colon (:), your cursor will move to the bottom line of your screen where you can type in the command and any parameters.
2. If you omit the exclamation point from the above commands, you may receive an error message such as one saying changes have not been saved, or the output file cannot be written (for example, you are editing a read-only file).
3. The : commands have longer forms (:quit, :write, :edit), but the longer forms are seldom used.

Moving around
The following commands help you move around in a file:
h
Move left one character on the current line
j
Move down to the next line
k
Move up to the previous line
l
Move right one character on the current line
w
Move to the next word on the current line
e
Move to the next end of word on the current line
b
Move to the previous beginning of the word on the current line
Ctrl-f
Scroll forward one page
Ctrl-b
Scroll backward one page
If you type a number before any of these commands, then the command will be executed that many times. This number is called a *repetition count* or simply *count*.
For example, 5h will move left five characters. You can use repetition counts with many vi commands.
Moving to lines
The following commands help you move to specific lines in your file:
G
Moves to a specific line in your file. For example, 3G moves to line 3. With no parameter, G moves to the last line of the file.
H
Moves relative to the top line on the screen. For example, 3H moves to the line currently 3rd from the top of your screen.
L
Is like H, except that movement is relative to the last line on screen, Thus, 2L moves to the second to last line on your screen.
Searching
You can search for text in your file using regular expressions:
/
Use / followed by a regular expression to search forward in your file.
?
Use ? followed by a regular expression to search backward in your file.
n
Use n to repeat the last search in either direction.
You may precede any of the above search commands with a number indicating a repetition count. So 3/x will find the third occurrence of x from the current point, as will /x followed by 2n.

Modifying text

Use the following commands if you need to insert, delete, or modify text:

i

Enter insert mode before the character at the current position. Type your text and press **Esc** to return to command mode. Use I to insert at the beginning of the current line.

a

Enter insert mode after the character at the current position. Type your text and press **Esc** to return to command mode. Use A to insert at the end of the current line.

c

Use c to change the current character and enter insert mode to type replacement characters.

o

Open a new line for text insertion below the current line. Use O to open a line above the current line.

cw

Delete the remainder of the current word and enter insert mode to replace it.

Use a repetition count to replace multiple words. Use c$ to replace to end of line.

dw

As for cw (and c$) above, except that insert mode is not entered.

dd

Delete the current line. Use a repetition count to delete multiple lines.

x

Delete the character at the cursor position. Use a repetition count to delete multiple characters.

p

Put the last deleted text after the current character. Use P to put it before the current character.

xp

This combination of x and p is a useful idiom. This swaps the character at the cursor position with the one on its right.