



Linux / UNIX Command Fundamentals and the File System



Linux / UNIX Commands

So you want to start using the powerful Linux / UNIX command-line interface, but you don't know where to start. Here's a suggested set of things to know how to do. But first, just a quick note on nomenclature to avoid unneeded confusion — **UNIX is a general family of operating systems of which Linux is just one example. Learn the commands once, and you know them everywhere. *Everything on this page should be true for any vaguely UNIX-like operating system.*** This includes:

- *Commercial examples like Solaris, HP-UX, IRIX, AIX, Tru64, and so on*
- *Any distribution of Linux*
- *All other free UNIX-family operating systems like FreeBSD, OpenBSD, and NetBSD.*
- *Apple OS X*

Read this page from top to bottom, or jump to a topic:

Tweet Share

reddit this!

Select Language ▼

Powered by Google Translate

| |
|--|
| Getting Started |
| File Name Wildcards |
| Very Fundamental Directories: . and .. and ~ |
| Running Commands and Your PATH |
| Redirecting Input and Output |
| Editing Text Files With vi |
| Listing Files and Directories |
| Manipulating Files and Directories |
| Examining File Contents |
| Changing File Permissions |

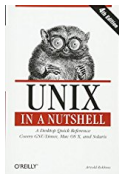
| |
|---------------------------------------|
| Who Am I and Where Am I? |
| Command History |
| Identifying and Controlling Processes |
| Network Commands |
| Controlling System State |
| Managing Users and Groups |
| Changing File Ownership and Group |
| Finding Files and Directories |
| Where The Pieces Go |

Getting Started and Learning More

This page is just a starting point, it mentions some of the most fundamental commands and provides examples of some simple and commonly needed tasks. The examples are just a glimpse of what they can accomplish, some of the commands have so many options that the overwhelming variety of possible command-line parameters could be considered a bug, as the command may be too complex to fully memorize.

This isn't really a problem! UNIX is designed around tools which each do one very specific thing well. Learn the main purposes of some core commands, and remember that you can always refresh your memory or learn more about options with the on-line manual pages.

The O'Reilly book **Unix in a Nutshell** is a good introductory text. Yes, there is also a *Linux in a Nutshell* title, but it uses a lot of its content showing you how to use fairly intuitive graphical interfaces. **Unix in a Nutshell** is much better for learning the command-line environment and powerful tools.



Unix in a Nutshell, Fourth Edition

By Arnold Robbins (Paperback - Nov 5, 2005)

\$20.09 ~~\$34.95~~

Rated 4.5 out of 5 by 103 reviewers on Amazon.com

[Buy Now](#)



Learning the vi and Vim Editors: Text Processing at Maximum Speed and Power

By Arnold Robbins, Elbert Hannah, Linda Lamb (Paperback - Jul 25, 2008)

\$23.90 ~~\$34.99~~

Rated 4.5 out of 5 by 49 reviewers on Amazon.com

[Buy Now](#)

Another good introduction is **The Linux Command Line: A Complete Introduction**, which is available as a free download [here](#).

Let's say that you want to **remove** a file but you do not remember the precise command. You can ask for a list of commands related to the topic. Ask for those one-line summaries of commands containing the string `remove`, where `-k` specifies *by keyword*:

```
$ man -k remove
```

You will see a *lot* of output, including commands to remove directories, to remove users, and arcane things for system administrators such as removing logical volume groups and named semaphores. But as you look down the alphabetical list of output you will see the line highlighted here:

```
[... lots of output ...]  
removef          (1)  - remove a file from software database  
remque [insque]   (3)  - insert/remove an item from a queue  
rm               (1)  - remove files or directories  
rmdir            (1)  - remove empty directories  
rmvq             (8)  - remove a message from a queue  
[... lots more output ...]
```

So you see that you probably need the command `rm` but you want to read a little about it. You need to verify that it's really the correct command and to see how to use it and possibly how to specify some optional behavior. Ask to see its detailed manual page:

```
$ man rm
```

Let's get started! Skilled Windows users will see that many features of the Windows command line interface were modeled directly on the UNIX command line interface — some things work exactly the same, and others are analogies.

File Name Wildcards

This is probably easiest to understand as a list of examples. You would use these literally on the command line, typing something like:

```
$ ls blah.*
```

or:

```
$ cp blah.[gjp]* backup-directory
```

You will see examples of these types of uses in later sections of this page.

*** matches any string, including nothing at all.**

`foo*` matches `foo`, `foot`, and `football`.

UNIX is stricter than Windows about matching. A Windows user would expect this command to remove all the files:

```
$ rm *.*
```

However, it does what you ask. *It only removes files with a "." in their name.*

```
$ ls
```

```
Index.html image.jpg logo.gif notes
```

```
$ rm *.*
```

```
$ ls
```

```
notes
```

The file `notes` has no "." in its name, so it was not removed.

? matches any single character.

`foo?` matches `food` and `foot`, but not `football`.

[*abc* . . .] matches one of any character in the list.

`foo[ldt]` matches `fool`, `food`, and `foot`.

You can specify ranges. `foo[a-c]` matches `fooa`, `foob`, and `fooc`.

`foo[a-cX-Z]` matches `fooa`, `foob`, `fooc`, `fooX`, `fooY`, and `fooZ`.

`foo[a-cmx-z]` matches `fooa`, `foob`, `fooc`, `foom`, `foox`, `fooy`, and `fooz`.

Be careful! `foo[d,t]ball` matches `foodball` and `football`, but it also matches `foo,ball` because that comma is treated as just another character.

[^ . . .] matches any single character *not* in the list.

`foo[^ldt]` matches everything `foo?` would match **except** for `fool`, `food`, and `foot`.

Brace expansion is a powerful shortcut that can compress a series of similar commands into just one. Instead of typing this:

```
$ mkdir /home/cromwell/project/subdir1
```

```
$ mkdir /home/cromwell/project/subdir2
```

```
$ mkdir /home/cromwell/project/subdir3
```

you could simply type this:

```
$ mkdir /home/cromwell/project/{subdir1,subdir2,subdir3}
```

or, even shorter:

```
$ mkdir /home/cromwell/project/subdir{1,2,3}
```

Very Fundamental Directories: . and .. and ~

The directory named ".", that is, just one dot, is your current working directory. What is it *really*? It's wherever you are at the moment. As Buckaroo Banzai said, "*No matter where you go, there you are.*"

Related to that, the directory named ".." is the parent of your current working directory. If your current working directory is `/home/cromwell` then:

```
. is /home/cromwell and
.. is /home.
```

There is just one exception — when you are at the root of the file system there is no parent. That is, when your current working directory is `/` then you can't go up to its parent.

Actually, there *is* a directory named "." in the root of the file system, but it points to the same location as it has the same i-node. If you don't believe me, verify this for yourself:

```
$ ls -ldi /*
```

The special directory name `~` refers to *your* home directory, while `~julie` would be the home directory of the user `julie`. Home directories are specified in the file `/etc/passwd`, but you can simply use `~` and the shell automatically looks it up.

Running Commands and Your PATH

You do not need to be in any particular directory to run a program, so long as either of the following is true.

1: The program is in a directory listed in your PATH environment variable, *or*

2: You specify the full path to the program. This could be an absolute path:

```
$ /usr/bin/vim
```

or a relative path. In this example, `mydir` is a subdirectory of your current working directory and it contains a program named `myprogram`:

```
$ mydir/myprogram
```

That relative path could start with ".", to mean "Run the program that's *right here*, not a program by that name installed in a standard system binary area":

```
$ ./myprogram
```

You can see your current PATH environment variable setting with the following command. You will see that the components are separated by colons:



```
$ echo $PATH
```

When you just type a command, the shell scans through the elements of your PATH looking for the first match. The first match wins — when the shell finds an executable program by that name in a PATH component, it runs that program. If no match is found, an error is reported. The precise error depends on your command shell, but it will be something like:

```
foo: Command not found
```

A reasonable PATH value for users includes the following directories. Note that X might be installed under X11R6 as shown here, or just X11, or even just X. You will have to investigate and think just a little:

```
/bin  
/usr/bin  
/usr/X11R6/bin  
/usr/local/bin
```

A reasonable PATH value for root includes the above directories and adds the following:

```
/sbin  
/usr/sbin  
/usr/local/sbin
```

Notice that "." should *never* be in the path for root as that would allow a malicious user to lay a trap. That means that root needs to specify the path to a command in the local directory, for example, `./programname`.

You can run anything from anywhere if you provide the details. For example, let's say that your Apache web server saves its logs in a bunch of files in the directory `/var/www/logs` and you want to process them with some program named `/var/www/tools/log-analyzer`, a program that reads the files listed as its command-line parameters. You want to run that analysis and save the output in a file named `result` in your home directory. All you have to do is this:

```
$ /var/www/tools/log-analyzer /var/www/logs/* > ~/result
```

Redirecting I/O — Writing To and Reading From Arbitrary Files

Again, this is probably easiest to understand by example, once you realize that any process has at least three standard I/O streams: input, output, and error. They are sometimes called `stdin`, `stdout`, and `stderr`, especially by C/C++ programmers.

The standard output stream is general output created by the process — its contents may be the main intended result of the program. The standard error stream is available for a second parallel stream of output intended for use in reporting errors. As you will see below, they can be sent to the same place, which could be the terminal screen where you started

the program, or to a file, or into another program as its standard input. Or, the two output streams could be sent to different places. It's up to you, Unix gives you fine control!

| Command Syntax | What Happens |
|--|--|
| <code>\$ myprogram</code> | <p>Run the program <code>myprogram</code> and allow the standard output and standard error streams to print on the terminal screen.</p> <p>If <code>myprogram</code> expects to read some data from its standard input stream, it will just wait patiently until you type something because standard input, by default, is connected to the keyboard of the terminal where you ran this command.</p> <p>You didn't specify a full path to <code>myprogram</code> so it needs to be stored somewhere in your <code>PATH</code>.</p> |
| <code>\$ myprogram < datafile</code> | <p>Just like above, except that <code>myprogram</code> is now reading its standard input from the file named <code>datafile</code>.</p> |
| <code>\$ myprogram > saveit</code> | <p>Run the program <code>myprogram</code> and send its standard output stream into a file named <code>saveit</code>.</p> <p>If the file <code>saveit</code> already existed, this overwrites it!</p> <p>If the program prints any error messages, they will appear on the terminal screen as you only redirected standard output, not standard error.</p> |
| <code>\$ myprogram >> saveit</code> | <p>Run the program <code>myprogram</code> and append its standard output stream to the end of the file named <code>saveit</code>.</p> <p>Any data in <code>saveit</code> is preserved as the new output is appended to the end of the file.</p> <p>Again, any error messages will appear on the terminal screen.</p> |
| <code>\$ myprogram > saveit 2> oops</code> | <p>Send the standard output of <code>myprogram</code> to <code>saveit</code> and standard error (stream #2) to <code>oops</code>.</p> |

| | |
|---|---|
| <code>\$ myprogram > saveit 2>&1</code> | Send the standard output (I/O stream #1) to <code>saveit</code> , and send standard error (stream #2) to the same place as stream #1. That is, save both standard output and standard error to the file <code>saveit</code> . |
| <code>\$ myprogram otherprogram</code> | Build a pipeline — the standard output of <code>myprogram</code> is used as the standard input of <code>otherprogram</code> . |
| <code>\$ prog1 prog2 prog3</code> | Build a bigger pipeline — the standard output of <code>prog1</code> is used as the standard input of <code>prog2</code> , and its standard output becomes the standard input of <code>prog3</code> . |

Editing Text Files With `vi`


Now you're ready to run some programs! But those programs will need data, and the programs may be simple shell scripts that you create. This means that **you need to edit simple text files**.

This means that you really need to learn some standard Unix text editor, either `vi` or `emacs`. Sorry, that's just the way that it is.

Note that `vi` may really be `vim`, which stands for "Vi Improved", on your system, or else both `vi` and `vim` may be available. If you use a variety of Unix systems, you might want to get into the habit of at least attempting to run `vim` and falling back to the slightly less friendly `vi` if necessary.

You can find a good on-line tutorial of `vi` at Yolinux.com, it goes deeper than this page does. Wikibooks has a book on-line, "[Learning the vi editor](#)". Other useful pages include vim.org and thomer.com.

You may also have the `vimtutor` program installed on your system, or you may be able to add it.

 AdChoices

Open Word File

Open Source



Learning the vi and Vim Editors: Text Processing at Maximum Speed and Power

By Arnold Robbins, Elbert Hannah, Linda Lamb (Paperback - Jul 25, 2008)

\$23.90 ~~\$34.99~~

Rated 4.5 out of 5 by 49 reviewers on Amazon.com

Buy Now

Crucial things to know about `vi` (and `vim`) include:

It has two modes: command and text insertion. When you're in text insertion mode, pressing keys just inserts those characters into the file at the current location of the cursor. When you're in command mode, pressing keys causes commands to happen.

Be careful, as always it is a bad idea to execute random commands. Typing away when you think you're inserting text but those characters are being interpreted as commands can have strange results at best, disastrous at worse.

To go from text insertion mode to command mode, just press the <Escape> key.

Now, when you're in command mode, there are *many* commands, but these are the vital ones plus a few more especially useful ones. The vital ones are those with *green* backgrounds. The good news is that you only need to know a handful of commands to be fairly useful. You will want to learn more so you can use the editor *much* more efficiently:

| Cursor Navigation | Meaning |
|-------------------|--|
| h j k l | Move one character to the left, town, up, and right, respectively. You must know how to navigate using these four keys. Some of you may have noticed that the arrow keys and the <Page Up> and <Page Down> keys work — at least they work <i>some</i> of the time, on <i>some</i> keyboards, <i>if</i> your environment fully understands and can handle your terminal emulator. Do not count on those special keys, know how to navigate with keys h, j, k, and l! |
| ^F ^B | Move f orward (control-F) and b ack (control-B) by one full screen. |
| ^U ^D | Move u p (control-U) and d own (control-D) by one-half screen. |
| w b | Move forward one w ord (w) or b ack one word (b). |
| e | Move to the e nd of the current word. |
| 0 \$ | Move to the beginning (0, zero) or end (\$) of the current line. |
| { } | Move to just before ({) or just after (}) the current paragraph or block of code. |

| | |
|--|---|
| H K | Move up to the first line (H) or down to the last line (L) currently visible on the screen. Notice how upper-case H and L are somewhat analogous to lower-case h and l. This is often the case with vi/vim commands. |
| Searching & Jumping | Explanation |
| /some-string<Enter> | When you type the / the cursor jumps down to a search and command line below the page of text shown on the screen. Then you type the string you want to search for, <i>some-string</i> in the example here. Then, when you press <Enter> the cursor will jump to the next instance of <i>some-string</i> , if it appears in the file. |
| n N | Pressing n jumps you to the next match for the most recent search string. N searches in the opposite direction. A message in the status line at the bottom of the screen will announce when the search wraps around from the end of the file to the beginning or vice-versa. |
| `` | Two back-quotes in a row jump the cursor back to where it was before the last search jump. |
| 123G<Enter> | This would make the cursor jump to the first text on line number 123. |
| ^G | Pressing <Ctrl-G> announces the current file name and line number of the cursor position in the status line at the bottom of the screen. You can configure vi to always display the word and line counts and positions in the status bar. This lets you check it manually. |
| Undoing & Re-doing | Meaning |
| . | Repeat the last deletion, addition, or modification at the current cursor position. |
| u | Undo the last deletion, addition, or modification. |
| Entering & Leaving Insertion Mode | Meaning |
| i a | Insert new text before the cursor position (i), or add new text after the cursor position (a). As with all commands that put you into insertion mode, the editor will insert everything you type until you leave insertion mode and get back into command mode. |
| I A | Insert new text at the beginning of the current line (I), or add new text to the end of the current line (A). |

| | |
|---------------------|--|
| o O | Open a new line below the current line (o), or before the current line (O). |
| <Escape> | Press the <Escape> key to leave insertion mode. |
| Deletion | Meaning |
| x | Delete the character currently under the cursor. |
| dw | Delete one w ord. |
| dd | Delete the current line. |
| D | Delete from the cursor through the end of this line. |
| And more... | Combine d (for "delete") with any motion character above to mean "delete from the current cursor position through the specified motion." That's really what dw means, but you can do things like delete the next half-screen with d^U or delete through the rest of the paragraph or block of code with d} . |
| Modification | Meaning |
| r | Replace the current character. Press x to issue this command, then press the new character. You will see the character change in place. |
| R | Replace from the cursor position through the end of the current line. Notice that this command and all the others after simple x throw you into text insertion mode. You can use R to replace the rest of this line with a few more words plus hundreds of lines of new text. |
| cw | Change the w ord. More precisely, change from the current cursor through the rest of this word. |
| cc | Change the current line. Compare cc for "change this line" to dd for "delete this line". |
| C | Change from the cursor through the end of this line. Compare c for "change the rest of this line" to D for "delete the rest of this line". |
| And more... | Just as with deletion, you can combine c (for "change") with any motion character above to mean "change from the current cursor position through the specified motion." |

| Copy & Paste | Meaning |
|---|--|
| Background... | You may know it as "copy & paste" because of some recent upstart programs that made up new terminology as they went along. But <code>vi</code> dates from at least the early 1980s, and for over a decade it was using the terms "yank & push" to refer to <i>yanking</i> some data off the screen and into a buffer, and then <i>pushing</i> it back out somewhere else. |
| <code>yy</code> | Yank the entire line into the buffer. |
| <code>p</code> | Push the data back out after the cursor position. If it's an entire line or more, it will appear below the current line. If it's a single character through a partial line, it will appear after the cursor position on this same line. |
| <code>P</code> | Just like <code>p</code> except it pushes it before the current cursor position, not after. |
| And more... | <p>This should come as no surprise, but you can combine <code>y</code> with a motion command character to yank a word (<code>yw</code>), yank the rest of the line (<code>y\$</code>), yank the rest of the current paragraph or block of code (<code>y}</code>), and so on.</p> <p>Another important thing is that whatever you last deleted with one of the deletion character commands is in the buffer, so you can move a block from here to there by deleting it with one command, moving to the new position, and pushing it back out.</p> |
| ed-style Commands | Meaning |
| Background... | Pressing ":" puts you into single-command mode, with the cursor jumped down to a ":" prompt below the editing area. This lets you describe changes to be made through the entire file, or through a range of lines. One somewhat (but not terribly!) complex command can do the work of a huge sequence of moves and changes. |
| <code>:1,22s/oldstring/newstring</code> | On lines 1 through 22, do a substitution replacing the first instance of <code>oldstring</code> on each line with <code>newstring</code> . |
| <code>:1,\$s/oldstring/newstring</code> <code>:%s/oldstring/newstring</code> | The same thing, except on <i>all</i> lines. Note that you can specify the line range as <code>1, \$</code> , "from line #1 to the end", or the short-cut <code>%</code> . |
| <code>:1,\$s/oldstring/newstring/g</code> <code>:%s/oldstring/newstring/g</code> | The same thing, except make it a global change — change <i>every</i> instance of <code>oldstring</code> on each line to <code>newstring</code> . |

| <code>:%s@/bin@/usr/bin@g</code> | This shows how to handle search or replacement strings containing "/" — use a different delimiter character. This would change every instance of <code>"/bin"</code> to <code>"/usr/bin"</code> . |
|-------------------------------------|--|
| Saving (or not) and Exiting | Meaning |
| <code>:w</code> | Write out the file as it exists now in the editor buffer, but stay within the editor session. |
| <code>:q</code> | Quit the editor session. If you have made changes that haven't been written, maybe you want to write them out first. But if you made some changes you didn't want to make... |
| <code>:q!</code> | Quit without writing the file. You started making changes but you weren't happy with the way the editor session was going. |
| <code>:wq</code> <code>ZZ</code> | Write the file to disk <i>and</i> quit the editor session. Classes usually teach <code>:wq</code> as the combination of <code>:w</code> and <code>:q</code> . Eventually you learn that <code>ZZ</code> does the same thing. |
| <code>:w!</code> | Write the file to disk, even though you don't have write permission, and stay within the editor session. This will only work if you own the file or you're <code>root</code> . (You need to have the permission to change the permissions to grant yourself write access, make the write, and then change the permissions back.) |
| <code>:wq!</code> | As above, but also quit . |

Listing Files and Directories

The Swiss Army Chainsaw is the `ls` command. It has *many* options, to the point that the BUGS section of the GNU manual page has at times referred to the large number of options and the inability of users to accurately remember and use most of them as a bug. It would properly be a design flaw, but there's a point in there somewhere.

Useful applications of `ls` include:

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

| | |
|--------------------|--|
| <code>ls</code> | <p>List the names of the files, or at least those with names not starting with a "." character. Remember that directories are just a special type of file.</p> <p>With no arguments, <code>ls</code> reports on the current directory.</p> <p>With one or more arguments, <code>ls</code> reports for each of those. If any argument is a directory, <code>ls</code> reports on its contents, not on the directory itself.</p> |
| <code>ls -a</code> | Add <code>-a</code> to report on all of the files, even those with names starting with a "." character. |
| <code>ls -d</code> | If any argument passed on the command line happens to be a d irectory, then report just that d irectory itself and not its contents. |
| <code>ls -l</code> | Produce a long listing with details as described below. |
| <code>ls -F</code> | <p>"Decorate" some of the reported file names with single characters indicating the file type, including:</p> <ul style="list-style-type: none"> / = directory * = executable @ = symbolic link = = socket <p>Note that the GNU version of <code>ls</code>, which might be simply <code>/bin/ls</code> on your system or perhaps something like <code>gls</code> or <code>colorls</code> on a BSD system, can also use color (if the terminal emulator supports it) to indicate data types for plain old files.</p> |
| <code>ls -R</code> | Produce a r ecursive listing, listing the specified file(s) and then their contents for any directories in the list, and <i>their</i> contents, and so on. |
| <code>ls -t</code> | Sort the output in t ime order, newest to oldest. |
| <code>ls -r</code> | <p>Reverse the order of the output. So, to see the files in reverse time order, oldest to newest, use:</p> <p><code>lt -tr</code></p> |

| | |
|--------------|---|
| Combinations | <p>If you like lots of details, you could issue a command like the following for a recursive (R) list of all (a) files, even dot-named ones, in the current directory, decorating the file names to distinguish directories, files, symbolic links, sockets, and so on, use:</p> <pre>ls -RaF</pre> <p>The order does not matter so you could have used any of these to get the same result:</p> <pre>ls -aFR or ls -aRF or ls -FaR or ls -FRa or ls -RFa</pre> |
|--------------|---|

Now, to read that long listing:


```
$ ls -lFd [Dp]* /tmp/file10jOkm /dev/ttyS0 /dev/sda1 /usr/bin/passwd /usr/bin/man
brw-rw----. 1 root      disk      8,  1 Apr 04 20:43 /dev/sda1
crw-rw----. 1 cromwell uucp       4, 64 Apr 04 20:43 /dev/ttyS0
srwxrwxr-x. 1 cromwell wheel      0 Apr 06 11:41 /tmp/file10jOkm=
-rwxr-sr-x. 1 root      mail     47360 Dec 14 17:55 /usr/bin/mail*
-r-s--x--x. 1 root      shadow   22104 Dec 14 13:56 /usr/bin/passwd*
drwxr-x---. 12 cromwell wheel  126976 Apr 07 14:14 Documents/
lrwxrwxrwx. 1 cromwell cromwell   18 Jan 28 14:40 pete@ -> purdue.jpeg
-rw-r--r--. 1 cromwell wheel  314943 Jan 22 12:16 purdue.jpeg
```

```
^ \_____/ ^ ^ ^ ^ \_____/ ^
| | | | | | | | | | |
| | | | | | | | | | | File name
| | | | | | | | | | |
| | | | | | | | | | | User Group | Time stamp
| | | | | | | | | | |
| | | | | | | | | | | Size in bytes for regular files.
| | | | | | | | | | | Major,minor device numbers for
| | | | | | | | | | | device special files.
| | | | | | | | | | |
| | | | | | | | | | | Number of links pointing to this in the file system. "Documents" is
| | | | | | | | | | | a directory with 11 subdirectories. There is one link in the current
| | | | | | | | | | | directory pointing to it by the name "Documents" plus one more link
| | | | | | | | | | | in each of those 11 subdirectories pointing to it by the name "..".
| | | | | | | | | | |
| | | | | | | | | | | Permission mask: 1st three for owner r = read
| | | | | | | | | | | 2nd three for group w = write
| | | | | | | | | | | 3rd three for others x = execute for files
| | | | | | | | | | | search for directories
| | | | | | | | | | | s = setuid/setgid, set user ID and
| | | | | | | | | | | set group ID
| | | | | | | | | | | - = permission not granted
| | | | | | | | | | | If there is a final "." on the permission mask, as seen here, it means that
| | | | | | | | | | | additional NSA Security-Enhanced Linux ACL (Access Control List) restrictions
| | | | | | | | | | | apply. See the SELinux ACLs with: ls --context
| | | | | | | | | | | Similarly, on BSD and Solaris a "@" or "+" character indicates extended ACLs.
| | | | | | | | | | | See them with: getfacl
| | | | | | | | | | |
| | | | | | | | | | | So, "Documents" has permissions rwxr-x---. cromwell wheel
| | | | | | | | | | | Subject to unexpected restrictions by the extended SELinux ACLs:
| | | | | | | | | | | Owner cromwell can read, write and execute (search)
```

```
|   Group wheel members can read and execute (search)
|   Others have no permissions
|
|   "/usr/bin/passwd" has permissions r-s--x--x.  root  shadow:
|   Subject to unexpected restrictions by the extended SELinux ACLs:
|   Anyone can execute it, but that one process will have the effective user
|   ID of "root" so it can read and write /etc/shadow.
|
|   "/usr/bin/mail" has permissions rwxr-sr-x.  root  mail
|   Subject to unexpected restrictions by the extended SELinux ACLs:
|   Anyone can execute it, but that one process will have the effective group
|   ID of "mail" so it can manipulate the directory /var/spool/mail.
|
File type:  -  =  regular file
             d  =  directory
             l  =  symbolic link
             s  =  socket
             b  =  block (buffered) special device
             c  =  character (raw) special device
```

Manipulating Files and Directories

| Command | Meaning |
|----------------------------|--|
| <code>cp foo bar</code> | Copy file <code>foo</code> to <code>bar</code> . If <code>bar</code> is a directory, this creates the file <code>bar/foo</code> . Otherwise, <code>bar</code> will be a file that is a copy of <code>foo</code> . |
| <code>mv foo bar</code> | Move file <code>foo</code> to <code>bar</code> . If <code>bar</code> is a directory, this creates the file <code>bar/foo</code> . Otherwise, <code>foo</code> will have its name changed to <code>bar</code> . |
| <code>mkdir foo</code> | Create a directory named <code>foo</code> . |
| <code>rmdir foo</code> | Remove the directory named <code>foo</code> . This fails with an error report if <code>foo</code> is not empty. |
| <code>touch foo</code> | If <code>foo</code> exists, change its modification timestamp . If <code>foo</code> does not exist, create it as an empty file . |
| <code>ln -s foo bar</code> | Create a symbolic link named <code>bar</code> pointing to <code>foo</code> . |
| <code>rm foo</code> | Remove <code>foo</code> . |
| <code>rm -r foo</code> | Recursively remove <code>foo</code> and its contents. |

| | |
|-------------------------|---|
| <code>rm -rf foo</code> | Recursively remove <code>foo</code> and its contents and force it to happen with no error or warning messages. Be careful using this one! |
|-------------------------|---|

Examining File Contents

| Command | Meaning |
|---|---|
| <code>more foo</code> | View the contents of <code>foo</code> one screen at a time. Press <code><Spacebar></code> to move forward one screen, <code>b</code> to move back one screen, and <code>q</code> to quit. Bonus: Press <code>v</code> to jump into a <code>vi</code> session at that point in the file. |
| <code>less foo</code> <code>my-program less</code> | The command <code>less</code> is like <code>more</code> except it's more capable. Those GNU folks love their puns and self-references.... One of the most useful differences is that <code>less</code> can back up when reading a stream, while <code>more</code> can only back up when reading files. |
| <code>cat foo</code> | Print all the contents of <code>foo</code> to the screen. |
| <code>cmp foo bar</code> | Compare the files <code>foo</code> and <code>bar</code> , answering the question <i>"Do <code>foo</code> and <code>bar</code> have identical contents?"</i> |
| <code>diff foo bar</code> | Show the differences between the contents of <code>foo</code> and <code>bar</code> . |
| <code>grep blah foo</code> | Print just those lines of <code>foo</code> containing the literal string <code>blah</code> ... |
| <code>grep -i blah foo</code> | ... except ignore the case of the letters in <code>blah</code> |
| <code>grep -w blah foo</code> | ... only when <code>blah</code> appears as an isolated word |
| <code>grep -iw blah foo</code> | ... ignoring case <i>and</i> only when <code>blah</code> appears as an isolated word |
| <code>grep -v blah foo</code> | ... except reversing the sense of the search, only the lines that do <i>not</i> contain the string <code>blah</code> |
| <code>egrep 'blah fnord' foo</code> | Notice that this uses <code>egrep</code> , not <code>grep</code> , "e" for "extended", to search for those lines that contain either the literal string <code>blah</code> <i>or</i> the literal string <code>fnord</code> . |

```
egrep -ivw 'blah|fnord' foo
```

Output only those lines that contain *neither* the isolated word `blah` *nor* the isolated word `fnord`, ignoring upper versus lower case. The options can appear in any order, `-ivw`, `-iwv`, `-viw`, `-vwi`, `-wiv`, or `-wvi`.

Changing File Permissions

Permissions can be specified in octal. Yes, octal, base 8. It will eventually become second nature, but to the uninitiated this is like Cypher in *The Matrix* when he waves his hand at the screens of cascading code and says he no longer sees the character patterns but instead what they represent. *"All I see now is blonde, brunette, redhead."* Only six patterns are really useful:

```
rw- -> 110 = 6
r-x -> 101 = 5
r-- -> 100 = 4
--x -> 001 = 1
--- -> 000 = 0
```

▶ AdChoices

Word Documents

Windows Software



Once you learn this, you will see `rw-r-x---` and read it as 750. Really. And you will immediately understand it as "Full access for the owner, all but writing for the group, nothing else."

Permissions can be specified explicitly in octal, as above, or symbolically:

```
          [ugo] [+ -] [rwx]
            ^   ^   ^
            |   |   |
+-----+ +-----+ +-----+
|         |         |         |
|         |         |         |
u = user (owner)  + = add      r = read
g = group          - = remove  w = write
o = other (world)  = = equal    x = execute (or search, for directories)
                                   X = search if it's a directory
                                   s = set-UID or set-GID
```

So, with all that background we're ready for examples:

| Command | Meaning |
|---------------------------------------|---|
| <code>chmod 760 something</code> | Change the permissions of the file <code>something</code> to mode 760, <code>rw-rw----</code> . |
| <code>chmod o+r something</code> | Add the permission for others to read the file <code>something</code> , but leave the other permissions alone. |
| <code>chmod -R o+r something</code> | Apply that change recursively to the directory <code>something</code> , its contents, and so on. |
| <code>chmod -R go=rX something</code> | Recursively give the group and others permission to read, and if it's a directory, also permission to search, and take away their permission to write, within the directory <code>something</code> , its contents, and so on. Leave the user permissions as they are. |

Who Am I and Where Am I?

| Command | Meaning |
|---------------------|---|
| <code>id</code> | List your credentials: your user ID, your primary group ID, and any other groups to which you belong. |
| <code>pwd</code> | Print the current working directory. |
| <code>echo ~</code> | If you were to run the command <code>cd</code> with no argument, so you change to your home directory, where would that be? |

Command History

| | |
|----------------------|--|
| <code>history</code> | Print your command history. The first column is the command number in the sequence. In some shells like <code>tcsh</code> , the second column shows the time at which that command was run. |
| <code>!!</code> | Re-run the last command in your history. |
| <code>!395</code> | Re-run command #395 in your history. |
| <code>!vi</code> | Re-run the most recent command starting with "vi". This doesn't have to be the command <code>vi</code> itself, just the most recent command starting with those two letters. It could be <code>vi</code> or <code>vim</code> or <code>view</code> or <code>vimtutor</code> or <code>visudo</code> or ... |
| <code>!v:p</code> | You are not certain, but you think that the most recent command starting with "v" <i>might</i> be the one you want. The ":p" means only <i>print</i> what would be run, do not actually run it. |

| | |
|---------------|---|
| !v:s/jpg/conf | <p>Let's say that the last time you edited a file with <code>vi</code> or <code>vim</code> you ran this command:</p> <pre>vim /some/long/path/to/whatever.jpg</pre> <p>when you really should have run this:</p> <pre>vim /some/long/path/to/whatever.conf</pre> <p>This history modifier means "Run the most recent command starting with 'v' <i>except</i> change the first instance of 'jpg' to 'conf'."</p> |
| ^jpg^conf | This does the above replacement of "jpg" for "conf" but applies to <i>the most recent command only</i> . |
| !c:gs/123/124 | <p>Let's say that you recently ran this command:</p> <pre>cp /path/to/dscf0123.jpg ~/Pictures/image-0123.jpg</pre> <p>Now you want to do the same thing for the next sequentially numbered image file name. This history modifier means "Run the most recent command starting with 'c' <i>except</i> change <i>every</i> instance of '123' to '124'."</p> |

Identifying and Controlling Processes

| Command | Meaning |
|------------------|--|
| ps axuw | List many details about all running processes. On Solaris you will need to use the full path to the much more useful BSD version of the command, so you don't get the fairly lame SVR4 one in <code>/bin</code> : <code>/usr/ucb/ps axuw</code> |
| top | Simply put, this lists the processes using the most of the CPU cycles, by default in decreasing order of CPU use percentage. See your local manual page, as <code>top</code> formats its output differently on different Unix implementations. There is usually a way to change the column by which the output is sorted, typically with the left-arrow and right-arrow keys, if they work correctly. There will probably be some way to change the update timing and specify which processes are displayed in terms of PID and/or user. |
| kill -HUP 1234 | Send a HUP signal to the process with PID 1234. Many daemons interpret a HUP signal to mean " <i>Keep running, but re-read your configuration file.</i> " |
| kill -TERM 1234 | Send a TERM signal to the process with PID 1234. A well-written program will use this opportunity to shut itself down cleanly. |
| kill -KILL 1234 | You tried a reasonably polite and gentle TERM signal but that did not work. Use a bigger hammer that won't shut it down cleanly, but it <i>will</i> kill the process. |
| pkill -HUP named | Like <code>kill</code> , but you can specify processes by names without looking up their numeric PID. |

Network Commands

For far more details and explanations of how to read some sample output, see my page with network commands for various operating systems. Here is a short version. Some of these will require you to have `/sbin` and maybe `/usr/sbin` in your `PATH`. Why? Because with a lot of other commands in those directories, they are administrator and power user tasks but not the sort of the thing the average user does. See the above section on `PATH` for details on this.

Network
Commands

| Command | Meaning |
|-----------------------------------|--|
| <code>hostname</code> | Show the hostname, probably the fully-qualified domain name. That is, <code>www.cromwell-intl.com</code> instead of just <code>www</code> . |
| <code>ip addr</code> | Show the IP address(es) and subnet mask for each interface. <i>Warning</i> — This <u>used</u> to be done with the <code>ifconfig</code> command. That command cannot be trusted to work as expected on a modern system! |
| <code>ip route</code> | Show the routing table. <i>Warning</i> — This <u>used</u> to be done with the <code>netstat -r</code> command. That command cannot be trusted to work as expected on a modern system! |
| <code>cat /etc/resolv.conf</code> | Show the DNS configuration: DNS domain and DNS name server(s). |

Controlling System State

For Linux with `systemd`, meaning most any Linux distribution after 2013–2014:

| Command | Meaning |
|---------------------------------|--|
| <code>systemctl poweroff</code> | Shut Linux down cleanly, halt the OS, and power off the system. |
| <code>systemctl reboot</code> | Reboot Linux cleanly. |
| <code>systemctl rescue</code> | Take the system to rescue mode, which means being <code>root</code> on the console. Depending on the configuration, you may need to type the <code>root</code> password. |

For UNIX family in general and Linux before `systemd`:

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

| | |
|---|--|
| <pre>init 0</pre> <pre>halt</pre> <pre>shutdown -h -t 0 now</pre> | <p>Shut UNIX down cleanly and halt the OS. You may be able to turn off the power, but to accomplish that you <i>may</i> need to use:</p> <pre>halt -p</pre> <p>Note that in some forms of UNIX (e.g., Solaris) these may behave rather differently and <code>halt</code> may be inappropriate.</p> |
| <pre>init 6</pre> <pre>reboot</pre> <pre>shutdown -r -t 0 now</pre> | <p>Shut UNIX down cleanly and reboot.</p> <p>Again note that in some forms of UNIX (e.g., Solaris) these may behave rather differently.</p> |
| <pre>init 1</pre> | <p>Go to single-user mode: maintenance mode, <code>root</code> on the console. On BSD you will need to use:</p> <pre>kill -s TERM 1</pre> |

Managing Users and Groups

All these commands must be run as `root`:

| Command | Meaning |
|------------------------------------|---|
| <code>useradd -m julie</code> | Create a user named <code>julie</code> and create the home directory <code>/home/julie</code> based on the template in <code>/etc/skel</code> . |
| <code>userdel julie</code> | Delete the user <code>julie</code> , the home directory and files will be left on the system. |
| <code>passwd julie</code> | Assign a password to <code>julie</code> . |
| <code>groupadd -g 100 staff</code> | Create a group named <code>staff</code> with group ID 100. |

Here are the files defining users:

| File | Defines |
|------|---------|
|------|---------|

| | |
|--------------------------|---|
| <code>/etc/passwd</code> | <p>This specifies most of the details about a user. The file must be world-readable so any user could use <code>~username</code> and have that resolved to a full path. The fields are:</p> <ul style="list-style-type: none"> login UID primary GID real name (or "GECOS field", for historical reasons) home directory command interpreter shell |
| <code>/etc/shadow</code> | <p>This specifies the password and related details about a user. Therefore it must be readable only by <code>root</code> and the operating system itself. The fields are:</p> <ul style="list-style-type: none"> login hashed password and salt account and password aging controls |
| <code>/etc/groups</code> | <p>The groups are defined here, one line per group. If a user belongs to more than one group, their primary group is defined in <code>/etc/passwd</code> and the other groups here.</p> <p>Let's say that <code>/etc/passwd</code> contains:</p> <pre>cromwell:567:101:....</pre> <p>and <code>/etc/group</code> contains:</p> <pre>wheel::10:cromwell users::101:</pre> <p>So, the user <code>cromwell</code> has a primary group ID of 101, meaning <code>users</code>, and also belongs to the group 10 or <code>wheel</code>. So, when that user creates a new file it will belong to the group <code>users</code> by default, and the user can optionally change that file to group <code>wheel</code> by running a command shown below.</p> |

Changing File Ownership and Group

With one exception, `chgrp`, all these commands must be run as `root`:

| Command | Meaning |
|----------------------------------|--|
| <code>chown julie bigfile</code> | Change the owner of the file <code>bigfile</code> to the user <code>julie</code> . |

| | |
|-------------------------------------|--|
| <code>chgrp wheel bigfile</code> | Change the group of the file <code>bigfile</code> to the group <code>wheel</code> . The owner of the file can do this as long as they belong to the target group. |
| <code>chown root:sys bigfile</code> | Change the owner of the file <code>bigfile</code> to the user <code>root</code> , and the group <code>bigfile</code> to <code>wheel</code> . |
| <code>chown -R root:sys data</code> | Apply that change recursively to the directory <code>data</code> and its contents. |

Finding Files and Directories

Most people seem to find the `find` command the most confusing, with a manual page that may not be terribly helpful. Here is an attempt to present some useful examples.

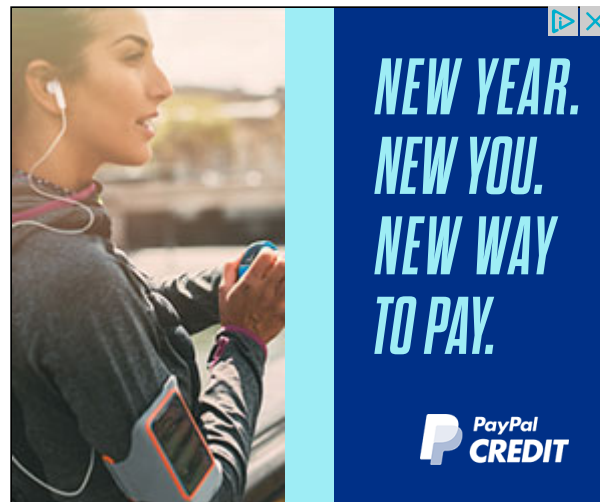
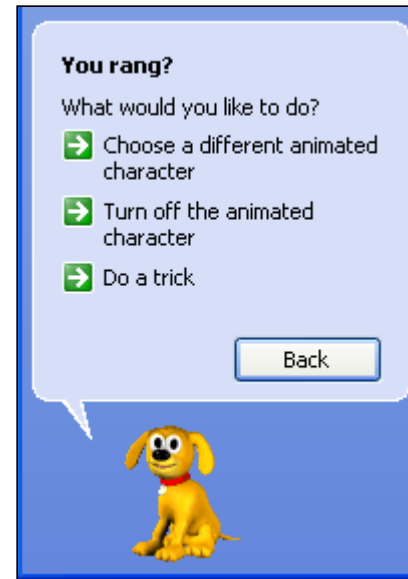
| Command | Meaning |
|--|--|
| <code>find /home -name trouble</code> | Find, anywhere under the <code>/home</code> directory, all files named <code>trouble</code> . |
| <code>find /home -name 'trouble*'</code> | Find, anywhere under the <code>/home</code> directory, all files with names starting with <code>trouble</code> . Notice that you have to hide the wild card character from the shell so it does not interpret it in terms of the files in your current working directory. Used as seen here, the wildcard is passed to <code>find</code> so it can do the search you want. |
| <code>find / -name 'trouble*'</code> | The same search, except look through the entire file system. |
| <code>find /tmp /var -name 'trouble*'</code> | The same search, except look under both <code>/tmp</code> and <code>/var</code> but nowhere else. |
| <code>find /home -type l</code> | Do any users have symbolic links under their home directories? |
| <pre>for DIR in /home/* > do > echo \$DIR \$(find \$DIR -type f wc -l) > done</pre> | <p>What are the user logins, and how many files does each one own?</p> <p>That command within backquotes, <code>\$(...)</code>, is executed first. Its output, a number in this case, is substituted into the outer command, the <code>echo</code>.</p> |

```
find /home -type l -name '*missing'
```

Do any users have symbolic links with names containing the string "missing" under their home directories?

Do any users own a file containing the string "mystery", ignoring case?

```
find /home -type f -exec grep -i mystery {} /dev/null \;
```



Notice the bizarre syntax of `find -exec` — the command you want to execute for every file matching the search criteria (in this case, under `/home` and a regular file) must appear between `-exec` and the delimiter formed by an escaped semicolon. The command will be run once for every file found, with `{}` replaced by the name of that file.

The example here always passes a second file name argument, `/dev/null`. The `grep` command can read that empty file in no time at all, but the presence of a second file name means that any output will be prefaced with the file name where the string was found. Compare the output of the two `grep` commands in this sequence:

```
echo foo > /tmp/bar
$ grep foo /tmp/bar
foo
$ grep foo /tmp/bar /dev/null
/tmp/bar:foo
```

There is usually a way to tell your `grep` command to include that file name even with a single file name on the command line, but the precise way to do that varies from one implementation to the other, and it isn't always possible at all.

Where The Pieces Go

This is hard to predict in detail, but the following table provides generally useful guidance.

See my page on [file system design for performance and security](#) for more detail, and suggestions of how to design a partitioning scheme.

| Directory | Contents |
|---|---|
| <code>/etc</code> | Most of the system configuration goes in <code>/etc/</code> and its subdirectories. |
| <code>/etc/rc*</code> <code>/etc/init.d</code> | Boot scripts, on older systems. <code>Systemd</code> replaces all the boot scripts with compiled binaries, and <i>completely</i> changes the way the system is started once the kernel has been loaded and found the root file system. Click here for the details of Linux booting. <code>Systemd</code> components go in <code>/lib/systemd</code> . |
| <code>/home</code> | User home directories |
| <code>/bin</code> <code>/usr/bin</code> <code>/usr/local/bin</code> | Programs generally useful to users |
| <code>/sbin</code> <code>/usr/sbin</code> <code>/usr/local/sbin</code> | Programs useful to boot and maintain the system, but not frequently used by users. However, some of the commands mentioned on this page, <code>ifconfig</code> and <code>ip</code> are prominent examples, are located in one of these. |

| | |
|---|---|
| <code>/usr/ucb</code> | On Solaris only, this is where the good and true and proper BSD versions of tools exist, as opposed to the less useful SVR4 versions in <code>/bin</code> or wherever. The most prominent example is <code>ps</code> but there are others where the UCB behavior may be more useful. |
| <code>/var</code> | All sorts of vital things that users never notice. Log files are in <code>/var/log</code> and/or <code>/var/adm</code> , mail and printing may use <code>/var/spool</code> or similar, and so on. |
| <code>/lib /usr/lib</code> <code>/lib64</code> <code>/usr/lib64</code> <code>/usr/local/lib</code> | Shared libraries critical to most of the executables on the system. Seriously. Do <i>not</i> mess these up. |
| <code>/dev</code> | Device-special files mostly ignored by users except for <code>null</code> , <code>zero</code> , <code>random</code> , <code>urandom</code> , and maybe a few others. |
| <code>/boot</code> | On a Linux system, the kernel itself and some support files are stored in here. |
| <code>/proc</code> | A user-accessible file system, or at least what appears to be a file system. Really it's a collection of kernel data structures. On Linux and some BSD versions, this is directly useful. On Solaris and most other commercial Unixes, this is really useful only for interpretation by <code>ps</code> and debuggers. That is, <i>if</i> you consider running a debugger on a running kernel "useful". |
| <code>/usr/share</code> | Application configuration files, collections of error messages and their translations into various languages, manual pages under <code>/usr/share/man</code> , further documentation under <code>/usr/share/doc</code> , and who knows what else your vendor or distribution assembler has stuffed into this area. |

Commands for Linux system administration

Hardening default installations of Linux and BSD

Other Linux / UNIX Topics

Cybersecurity

You May Like These