

# An introduction to UNIX basics

Chris Brooks

## 1 Introduction

Unix is an extremely powerful and flexible operating system. The version you are most likely familiar with is Linux (sometimes referred to as GNU/Linux, or by the name of one of the many Linux vendors, such as RedHat, Debian, or Slackware), but there are many different flavors of Unix, including Irix, Solaris, FreeBSD, HP-UX, and OS X's shell. They all share the same basic commands, although occasionally the options or specifics of output will vary between flavors. We'll focus on RedHat Linux here. Check the man pages for other flavors as needed.

New arrivals to Unix are often a bit overwhelmed by the fact that interaction is done primarily via a command-line interface. Users coming from a Windows/Mac background, where they're used to being presented with a small number of pre-defined options, are usually a bit perplexed about how to interact with the Unix shell, which has a large number of commands, all of which can be strung together and combined to make new commands. After a bit, though, users realize that the command line frees you to get the results *you want*, rather than the results a GUI designer thought to include.

Unix also has the advantage of having its documentation included. This documentation is referred to as *man pages*. To find out what a command (say 'ls') does, type "man ls". There's an art to reading a man page; they're not written as a tutorial, but as a reference. It can be difficult to understand how a complex command works from a man page, but if you need to know what a particular option does, they're a great resource.

A man page begins with a synopsis describing how the command is invoked, followed by a short description of what the command does. This is typically followed by a description of the command's options and what each of them do. This is followed by usage information, the author, and any known bugs, plus related commands.

Sometimes, you may not know the name of the command you're interested in. (For example, you may want to know "How do I list the contents of a directory?") You can find this by doing `man -k string` or `apropos string`. For example,

```
[brooks@valis Mail-Sendmail-0.78] man -k list
...
lreplace          (n) - Replace elements in a list with new elements
```

```
ls                (1) - list directory contents
lsattr           (1) - list file attributes on a Linux second
extended file system
...
```

## 2 Files and Directories

The Unix filesystem uses the familiar metaphor of a tree-shaped directory structure, with files and directories (folders). If you've used DOS, many of these commands will look familiar, although there are some subtle differences, which we'll go into later.

### 2.1 Locating yourself and moving around

A user will always have a *current* or *working* directory. This is the directory in which the user is currently located, and uses the symbol '.' as a shorthand. For example, if you wanted to view the contents of a file 'foo' in the current directory using `cat`, you could do: `cat ./foo` or `cat foo` (since `cat` uses the current directory by default).

To find out what directory you're currently in, use `pwd` (for 'print working directory'). For example,

```
[brooks@valis brooks] pwd
/home/brooks
```

To change directories, use `cd` (for 'change directory'). You can either use an *absolute path*, which is a path beginning with the root directory, or a *relative path*, which indicates a new directory's location relative to the current working directory. Some examples of using an absolute path are:

```
cd /home/brooks/temp
cd ~brooks/temp - Note that here we use the shorthand '~' (called 'tilde')
to indicate the home directory of user 'brooks'.
```

```
cd ~/temp - the tilde by itself refers to the user's own home directory.
```

Some examples of a relative path are:

```
cd ./temp/images - change to the directory temp/images, which is a sub-
directory of the current working directory.
```

```
cd ../code/myprogram - '.' is a shorthand for the directory above the cur-
rent directory. this will move you up one directory, then down to code/myprogram.
```

```
cd .. - this will move you up one directory.
```

### 2.2 Listing a directory

Once you've moved to a directory, you'll want to know what's there. The command for listing a directory is `ls`. `ls` has a large number of options; I'll list a few of the most useful here:

- `-l` - long listing. Show the files' permissions, owners, size, and modification date.
- `-a` - show 'hidden' files, which are files whose names begin with `.` For example, `.cshrc`, `.emacs`, `.bashrc`.
- `-R` - recursively list subdirectories.
- `-S` - sort by file size.
- `-t` - sort by modification time.

There are many other options, check the man page for details. Multiple options can be combined. For example:

```
[brooks@valis code] ls -alt
total 14060
drwxr-xr-x  75 brooks  faculty      4096 Aug 22 12:42 ..
drwxr-xr-x   3 brooks  faculty      4096 Mar 25 16:43 aima
drwx-----  5 brooks  faculty      4096 Mar 25 16:43 .
drwx-----  3 brooks  faculty      4096 Oct 17  2002 clmusic
drwx-----  4 brooks  faculty      4096 Sep  3  2002 mult-producer
-rwx-----  1 brooks  faculty 14355930 Aug 28  2002 mult.tgz
```

```
[brooks@valis code] ls -lSa ~/mail/
total 24
-rw-----  1 brooks  faculty      6580 Aug 12 12:29 Trash
drwx-----  2 brooks  faculty      4096 Aug 12 12:29 .
drwxr-xr-x  75 brooks  faculty      4096 Aug 22 12:42 ..
-rw-----  1 brooks  faculty      2727 Aug 12 12:26 Sent
-rw-----  1 brooks  faculty       506 Jul 21 17:41 Drafts
```

## 2.3 Creating and removing directories and files

Making a new directory is done using `mkdir`. For example, `mkdir temp`.

To remove a file, use `rm`. For example, `rm foo`. Note that unix does not have a 'trashcan' or an undelete function. If you delete something, it is gone, so be careful. Some people use the `-i` option with `rm` for this reason. In this instance, you will be prompted before files are deleted.

You can remove a directory using `rmdir`. For example, `rmdir ./temp`. `rmdir` will only work on a directory that is empty. If a directory has files in it, there are two ways to remove it.

- 1) First remove all the files using `rm`, then use `rmdir` to remove the directory.
- 2) The `-r` option to `rm` will recursively delete the directory, along with all files and subdirectories in it. Be careful when using this. For example:

```

brooks@valis temp] ls
foo (the current dir has a subdirectory called foo)
[brooks@valis temp] ls foo
example (foo contains a file called 'example')
[brooks@valis temp] rm -r foo (we remove foo and everything below it)
[brooks@valis temp] ls
[brooks@valis temp] (the dir is now empty)

```

## 2.4 File Permissions

In talking about `ls`, we mentioned file permissions. In Unix, there are three types of file permissions: read permission, write permission, and execute permission. In addition, Unix allows us to specify who can have each of these permissions. There are three types of users: user (the person who owns the file), group (other users in the same group as the user) and other (everyone else).

We haven't mentioned groups yet. In unix, your sysadmin can create groups of users (for example, people all working on the same project). To find out what groups you're in, just type `groups` at the prompt.

So three types of permissions times three types of users gives nine different permissions, which are typically represented as a string, like so: `rw-r-xr-x` This says that the owner of the file (the first three characters) can read, write and execute the file, and group and other users can read and execute (but not write) the file.

There are two ways to change a file's permissions, both using the `chmod` command. The first allows you to explicitly specify a user and permission. For example, `chmod u+w foo` adds write permission of the user/owner of the file, and `chmod g-r foo` removes read permission for group members on `foo`.

the second method allows you to compactly specify the entire permission string. First, note that each user's permissions consist of three bits which are either on or off. This allows us to map permissions onto the octal representation for the numbers 0-7, as follows:

```

0 000 ---
1 001 --x
2 010 -w-
3 011 -wx
4 100 r--
5 101 r-x
6 110 rw-
7 111 rwx

```

So, if we wanted the user to have all permissions, group members to have read permissions, and others to have execute permissions, we could do: `chmod 741 foo`. Most commonly, you'll do `chmod 755 foo` (to let yourself do anything,

and let others read) or `chmod 700 foo` (to let yourself do anything, and deny all access to others). Note that `chmod` also has a `-R` option, which will recursively change the permissions of all subdirectories and their contents.

## 2.5 Viewing files

Unix has several utilities available for viewing the contents of files. Some of the most useful are:

- **cat** - dump the contents of a file to the screen. By itself, this is not that useful, but when combined with pipes and redirection (discussed below) it is quite helpful.
- **less** - Display a file one page at a time. You can also back up (hit 'b'), search ('/'), or jump to a particular line number (`;``num``;`), among other things. Check the man page for more details. (Why is it called 'less'? It's a replacement for an earlier program called 'more'. Computer humor ...)
- **head** - display the first `n` lines of a file (by default 20). `head foo` will display the first 20 lines of `foo`, and `head -n 5 foo` will display the first 5 lines of `foo`.
- **tail** Display the last `n` lines of `foo`, in the same manner as `head`. Another useful option of `tail` is the `-f` option - this allows you to watch the bottom of a file as it changes - very useful for log files. Try creating a file called 'foo' and doing `tail -f foo`. Then, in another window, do `echo 'Hello'`  
> `foo`.
- **wc** Word count. Displays the number of characters, words and lines in a file. Useful for telling how much code you've written or how long that essay is.

## 3 Pipes and Redirection

The paradigm for developing unix tools and programs is that of a toolbox. Unlike Windows and Mac programs, which tend to be large and do lots of things, the thinking behind unix tools is to build something that does one task well, and then compose these tools to make larger, more complex tools using *pipes* and *redirection*.

Unix programs all work with two basic *streams*, the input stream (called STDIN) and the output stream (called STDOUT). (there's also the error stream, STDERR, but we'll skip over that.) When you did the command `less ./foo`, the input stream was the file 'foo', and the output stream was the screen. You can redirect this output to a file using the '>' character. (think of it as an arrow pointing in the direction the data will travel.) For example, `ls -l > data` will execute the command 'ls -l' and redirect the contents into the file 'data'.

Anything that was previously in 'data' will be overwritten. If you wanted to append the result of 'ls -l' to 'data', you would instead do: `ls -l >> data`.

Similarly, you can redirect the input of a command from a file instead of from the keyboard by using the '<' character. For example, if the file 'fname' contained the string 'data', you could do `ls -l < fname` to list the current directory's contents. (Admittedly, this is a pretty dull example. We'll see more interesting examples when we get to `grep` and `sort`.)

Even more useful is connecting two programs so that the output of one serves as the input for another. This is done using a pipe (`|`). This connects the `STDOUT` of the first command to the `STDIN` of the second command. For example, suppose you wanted to know how many files were in the current directory. One way to do this is: `ls | wc`. In this case, the output of `ls` (a list of files) serves as the input for `wc`.

You can also chain multiple pipes together. We'll see examples of this below.

## 4 Searching and Filtering

Now that we've got some basic tools, let's look at some useful commands for generating output, and we'll use pipes to help filter this output.

### 4.1 `grep`

One of the most useful commands is called `grep`, which stands for *generalized regular expression parser*. `grep` allows you to look through a file (or more generally, any input stream, and select lines that contain some sort of pattern. For example, the following command prints all lines in `/etc/passwd` that contain the string 'root'.

```
brooks@valis temp] grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
```

`grep` is often used with the pipe to filter the output of other commands. For example,

```
[brooks@valis brooks] ls -l | grep mail
drwx----- 2 brooks faculty 4096 Aug 12 12:29 mail
drwx----- 2 brooks faculty 4096 Sep 19 2002 nsmail
```

This way, we can limit the output to those file we're actually interested in.

`grep` can also deal with regular expressions as search strings. Regular expressions consist of strings, plus a set of *wildcard* characters that can match any of a number of characters. Some commonly used wildcards are:

- `*` - matches 0 or more instances of the previous character (any). For example, `cat*` would match `cat`, `catch`, `caterpillar`, `cats`, `cate`, and so on.

`cat*dog` would match `catdog`, `catabcdedog`, `cattdog`, and so on. Anything can occur after the `*`, so `cat *` will apply `cat` to every file in the current directory.

- `+` matches one or more occurrences of the previous character. So `cat+dog` would match `cattdog` and `catttttdog`, but not `cadog`.
- `?` matches 0 or 1 occurrence of the previous character. `cat?dog` matches `catdog` and `cadog`, but not `cattdog`.
- Brackets are used to specify ranges of letters or numbers. For example, `cat temp[0-9]` would apply `cat` to `temp0`, `temp1`, ..., `temp9`, but not `temp10`. You can also apply the wildcards to bracketed ranges, so `ls [a-f]+` would list all files whose names consisted only of the letters `a,b,c,d,e,f`.

This is only a partial description of regular expression syntax. The man page for `grep` gives a complete description.

`Grep` is useful when you're trying to find a string in a file. Oftentimes, you'll know that one of the files in the current directory has some information, but you're not sure which one. `Grep` and wildcards can help here. For example, `grep stuff-to-find * | less` looks for the string 'stuff-to-find' in the current directory and prints each line that matches this string. The results of this are sent to `less`, so that we can see them one page at a time.

## 4.2 find

Sometimes, you know the name of the file, but not where it is. In this case, the `find` command is very helpful. The syntax is: `find [starting dir] -name [file] -print`. `[starting dir]` is the directory you want to look in; `find` will search this directory and all subdirectories (assuming you have access) and print any matches. For example, `find . -name core -print` finds all `core` files at or below the current directory. You can also use wildcards with `find`, although you have to put your search string in quotes. For example, `find . -name 'core*' -print` finds all files beginning with `core`.

There are lots of other options to `find`. For example, you can search by things other than `-name`. `-atime -n` looks for files accessed less than `n` days ago, `-size +nk` finds files larger than `n` kilobytes, and `-user` finds files belonging to a particular user. You can also choose actions other than printing. The most common thing to do is to execute a command. This is done with the following (slightly obscure) syntax:

```
find . -name core -exec rm {} \;
```

This will find all files named 'core' and remove them. The `\;` is needed by the shell to execute each 'rm' command accurately - just take it as something that has to be there. (Note: be careful when trying this out with 'rm' - remember that unix doesn't have an undo.)

`Find` has *lots* of other features - check the man page for details.

### 4.3 sort

Sort's function is pretty straightforward; it sorts input. It's particularly helpful in filtering the output of a previous command. For example, `ls | sort` will print an alphabetical list of files. Some of the useful options to sort are:

- `-n` sort numeric data as numbers, rather than alphanumeric characters.
- `-r` print output in reverse order.
- `-k` POS sort based on column pos. For example, `ls -l | sort -n -k 5` will sort the result of `ls -l` according to increasing size (the 5th column). Note that `-n` is needed to sort the sizes as numbers, rather than strings.

### 4.4 which and whereis, and the path

When you execute a command in Unix, such as `ls`, what you're actually doing is invoking a separate program named `ls`. It's located in `/usr/bin`, so you could type `/usr/bin/ls` and get the same result.

This may lead you to wonder, how does the OS know where these programs are located? The answer is that every user has an environment variable called `PATH` that tells the OS where to search for programs. To see your path, just type: `echo $PATH`. You should see a list of directories, separated by colons. When you type a command, the OS searches your path (from left to right) for an executable file that matches that command.

You can modify your path. For example, let's say you have some programs in `/home/brooks/bin` that you want to run without having to type out the whole path every single time. To add `/home/brooks/bin` to your path, do:

```
[brooks@valis brooks]$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/brooks/bin:
/usr/local/mpi/bin:/usr/local/apache-ant-1.5.2/bin
[brooks@valis brooks]$ PATH=$PATH:/home/brooks/bin; export PATH
/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/brooks/bin:
/usr/local/mpi/bin:/usr/local/apache-ant-1.5.2/bin:/home/brooks/bin
brooks]$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/brooks/bin:
/usr/local/mpi/bin:/usr/local/apache-ant-1.5.2/bin:/home/brooks/bin
```

Note that we extend the path by concatenating the new directory to the existing one, by doing `PATH=$PATH:/home/brooks/bin`. Then, we use `export`. This ensures that the value of this new `PATH` variable can be seen by other processes. Otherwise, the shell would 'forget' the new value.

Sometimes, you'll try to execute a command and get unexpected behavior. Either the command won't be found, or you'll get the output from some other command. In this case, `which` is very useful - it tells you which version of the command is being executed. For example,

```
[brooks@valis brooks]$ which chmod
/bin/chmod
```

`whereis` is similar, but also lists the location of man pages.

## 4.5 diff

Often, you want to compare two files (say two versions of a program) and find out how they're different. The way to do this is with `diff`. `diff file1 file2` will compare `file1` and `file2` and print the lines on which they differ. Usually, you'll want to pipe this output into another program, such as `less` (to see one page at a time) or `grep` (to search for particular lines).

## 5 Processes and resource usage

Unix is a multiprocess, multiuser system. This means that a user can have many programs running at once, and many users are sharing the same system. Often, it's nice to know what your different programs (or *processes*) are doing.

To see all the processes associated with your username, use the `ps` command. For example:

```
brooks@valis brooks]$ ps
  PID TTY          TIME CMD
28078 pts/3    00:00:00 bash
22033 pts/3    01:40:44 squeak
 3634 pts/3    00:00:00 ps
```

Here we see the process ID (a unique identifier), the terminal the process was started on, how long it has been running, and the name of the process (note that `ps` itself is in the list). `ps` has several useful options including:

- `-a` list all users' processes
- `-u` long listing

Sometimes you need to terminate a process - perhaps it's hung, or needs to be restarted. You can do this with `kill`, followed by the process ID. For example, `kill 22033` would kill the `squeak` process in the example above. An exception to this is if a process has other processes that depend on it. In the above example, the `bash` process spawned the `squeak` process, and so if `bash` was killed, `squeak` would also die. To kill `bash` and all its children, use the `-9` option, like so: `kill -9 28078`.

`top` is a command that will tell you about the current resource usage on your machine. At the top of the screen, it tells how long the machine has been up, the number of users, and the system load, along with memory usage statistics. Following this is a list of the `n` largest processes, the amount of memory and CPU they are using, and their status (R is running, S is sleeping, SW is swapped

out, T is stopped). This can be very useful in figuring out why your machine is running slowly.

Often, you'll want to diagnose not your machine, but its connection to the network. Two useful commands for doing this are `ping` and `traceroute`. `ping` sends a packet from your computer to the other hostname given on the command line (for example `ping www.google.com`), who then returns the packet. This will tell you if there's a path to another machine. It's very useful for telling if a problem is related to your machine or a remote computer.

Even more helpful is `traceroute`. (`traceroute` may not be in your path; use `which` to find out. Sometimes it's in `/usr/sbin`) Traceroute will tell you the path that a packet takes on the way to a host, along with some timing statistics. This can help you determine if there's problems somewhere in your network.