Basic commands

Brian Tiemann

A newcomer to FreeBSD will probably find himself served well by the KDE desktop environment that ships as a default part of PC-BSD, the desktop oriented version of FreeBSD, or that can run on FreeBSD just as easily as on Linux. KDE provides a full-featured desktop-style interface that gives you the same mouse-driven control, windowed application sharing, and icon-based file management that has become a part of all our lives over twenty years of Windows and Mac desktop computing.

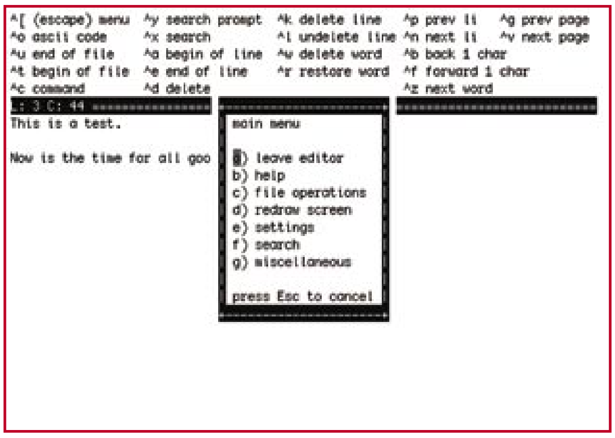
But FreeBSD isn't just about the desktop; indeed, more so than Linux, it's aimed at the server market, with far less attention paid to desktop users than to system administrators who want a stable and speedy platform for running their high-performance network services.

A server in that role has no use for a graphical desktop environment; it seldom even has a monitor. All the interaction a user generally has with a FreeBSD machine is through a textual command line accessed through a remote SSH connection. Instead of clicking icons and menus, users memorize and type commands, exchanging all communication with the system through a rectangle of plain text. It's the way it's been done for decades – and for Unix-heads, it's still the best way. But for a newbie to the world of Unix, accustomed to computers as being launching points for web browsers and digital movies and 3D video games, it's a lot to swallow.

Fortunately, anyone whose experience with alternative operating systems includes any time spent dabbling in Linux can count himself fortunate if confronted by a FreeBSD machine. This is because, as you can expect from any Unix, FreeBSD works in much the same way that Linux does, particularly to a brand-new user who wants to know what the basic commands are for getting around in the command-line interface.

Getting around

In a graphical operating system, the first thing the curious user does is double-click things on the desktop and start navigating around through the folders that are available. That impulse is no different in the command line interface (CLI) world; it's just that the tools and techniques you use to do it are a bit different, and require you to memorize a few short commands and their obscure suffixes (*flags* or *parameters*) rather than learn a few moves to use with your mouse. The first command, the one that does the equivalent of showing you a



```
^[ (escape) menu  ^y search prompt  ^k delete line   ^p prev li    ^g prev page
^o ascii code     ^x search         ^l undelete line ^n next li    ^v next page
^u end of file    ^a begin of line  ^w delete word   ^b back 1 char
^t begin of file  ^e end of line    ^r restore word  ^f forward 1 char
^c command        ^d delete                          ^z next word
L: 3 C: 44 ==================
This is a test.                  main menu

Now is the time for all goo      a) leave editor
                                 b) help
                                 c) file operations
                                 d) redraw screen
                                 e) settings
                                 f) search
                                 g) miscellaneous

                                 press Esc to cancel
```

Figure 1. *Editor you can launch from command line*

window with a list of the files in it, is the `ls` command, which stands for *list*:

```
% ls
Birthday.mov          essays ↲
   shopping_list.txt
My DVD List.txt       mozilla
Picture1.jpg          mp3
```

(if you're already a functional Linux or Unix user, none of this will be new to you; just bear with me here.) The textual output shown here consists of a line for the command you type, printed in bold, following the prompt set by your system ("`%`" in this case, but many systems use different prompts customized for their own users' purposes). This is followed by the output that appears after you press [*Enter*] or [*Return*].

What's happening here is that a hypothetical user typed the `ls` command, requesting a listing of all the files in the current directory (a *directory* is what you might be used to thinking of as a *folder*, an entity that contains files and other folders in a hierarchy). The system responds by printing out a table of file names. There aren't any pretty icons to go with them, and you can't easily tell which are files and which are folders, but aside from that there isn't that much to separate this conceptually from double-clicking a folder name to get a window full of file icons.

Suppose, though, that you wanted to get a little bit more out of your file listing. File names are just text, after all, and text is what a CLI has in spades; you ought to be able to coax a little more meaning out of this listing. Well, you're in luck: this is a perfect opportunity to demonstrate how parameters work.

```
% ls -F
Birthday.mov      essays/    ↲
      shopping_list.txt
My DVD List.txt       mozilla*
Picture1.jpg@         mp3/
```

Adding the "`-F`" flag (after a space, don't forget the space!) modifies the output of `ls` by adding symbols that indicate what kind of file they are. They're not application-based icons like the ones you might be used to, distinguishing one file type from another; rather, this system distinguishes objects much more broadly, by using a slash ("`/`") to denote directories, the "`@`" sign for symbolic links (which you can think of as Unix versions of shortcuts or aliases), and asterisks ("`*`") for executable files (for example programs). For example, in this sample output you can tell that `mp3` is a folder (probably containing music files), *Picture1.jpg* is really an alias for an image file somewhere else in the system, and `mozilla` is an executable program, which you can launch like this:

```
% ./mozilla
```

The "`./`" prefix here tells the system that the program file `mozilla` is in the current directory, denoted by a single dot (the parent directory is two dots, "`..`"). Why isn't this obvious, you ask? Shouldn't the system be able to figure that out? Well, yes - except that being able to run a command that refers to a file in your current directory is generally thought of as poor style and bordering on bad security practice (in the Unix world, everything you do takes place under an invisible mental banner that says CONSIDER THE SECURITY IMPLICATIONS – or at least it ought to).

In FreeBSD, as in Linux and every other CLI-driven Unix, your command-line environment comes preloaded with several *paths* to directories in the system where executable programs might be found. These paths might include */bin*, */usr/sbin*, */usr/local/bin*, and a few others. (The `ls` command, for example, lives in */bin*.) But suppose you had a program called *top* that was sitting in your personal directory. If you typed *top* from within that directory, that personal program might launch instead of the one installed at the system level. It's possible that that file might have been put there maliciously by someone who gained access to your account; and if it's not really the *top* program, but a piece of malware designed to exploit your user privileges somehow to gain control of the system or otherwise mess things up, you could find yourself unwittingly triggering a destructive event by running what you think is a routine program.

This might seem far-fetched, as can the idea that forcing you to specify "`./`" to indicate a local executable file in the current directory might make much of a difference; but you'll soon learn not to be surprised at the ingenuity of mischief-makers, or the benefits of even a small inconvenience standing in their way. If it means memorizing a little more of this arcane CLI knowledge for the benefit of increased compliance with accepted best practices, you're better off swallowing that pill than trying to find a way to avoid it (for example by customizing your shell's environment variables to add "`./`" to your path, which you can do, but you really shouldn't).

While we're on the subject of the *current directory*, though, you might notice that in a command-line environment, there's not really any indication of where in the filesystem you are.

Whereas in a GUI like KDE or Windows you can see each folder graphically represented as a window or an icon in a clear hierarchy, at the command line all you see is a list of files.

Some systems are set up to display your location right in the prompt, like this:

```
~ btiemann/mp3 %
```

But if yours isn't configured that way, you can still find out your location using the `pwd` command (it stands for *present working directory*):

```
% pwd
/home/btiemann/mp3
```

This tells me that I'm in the "`mp3`" subdirectory of my home directory. I can move around using the `cd` (*change directory*) command, like this:

```
% cd /home/btiemann
```

This would move me into my home directory.

A few other usages of `cd` would be to take me to a subdirectory, specified with a relative path (for example one that does not begin with the slash, `"/"`, that denotes the root, or topmost, directory of the entire system):

```
% cd essays
```

Or to change to the parent directory:

```
% cd ..
```

Or to move two levels up the tree:

```
% cd ../..
```

Or to return to my home directory using the shortcut of omitting the path parameter altogether:

```
% cd
```

Working with Files

Now that you know the basics of command-line navigation, you can put it to use by moving some files around. The first thing to do might be to create a text file. You can use the built-in `"ee"` editor for this:

```
% ee test.txt
```

When you've typed some text, press [*Escape*] to bring up the menu, then use the *Up* and *Down* arrows to select *Leave editor*. Choose *Save changes* in the next screen, and you'll exit the editor having created a new file. You can get a closer look at it now using the ls command in another new way:

```
% ls -l test.txt
-rw-r--r--  1 btiemann users ↲
   561 Mar 31  2008 test.txt
```

Here, the `"test.txt"` argument makes `ls` show only the information on the specified file, and the `"-l"` flag makes it print its output in `"long"` form, meaning to show information on the file's permissions, ownership, last-modification date, and size (in bytes).

You can now move the file to some other location. This is done using the `mv` command (many core Unix commands, as you can see, are abbreviated to an almost comical degree - but for hard-core users who use these commands hundreds of times a day, the less typing they can get away with, the better):

```
% mv test.txt essays
```

This moves the file into the `"essays"` subdirectory. Just a with `cd`, you can also specify a destination such as the parent directory:

```
% mv test.txt ..
```

Or, if the file you want to move is somewhere other than your current directory, you can use a relative path to refer to it:

```
% mv ../test.txt essays
```

Or an absolute path:

```
% mv /home/btiemann/test.txt ↲
   essays
```

What happens if you move a file into a directory where there's already a file with the same name as the one you're moving? Well, if the file's permissions allow it, the old file gets overwritten, without so much as a warning. This is one of the pitfalls – or, depending on how you see it, the blessings – of an austere CLI environment: it really knows how to get out of your way and let you do your work... or shoot yourself in the foot.

Indeed, the `mv` command has many potentially destructive uses. One of the most commonly surprising ones, to newcomers to Unix, is that the `mv` command is what you use to rename files.

Rather than having a dedicated command for `"rename"`, the developers of Unix decided that if you're changing a file's name, what you're really doing is `"moving"` the old file to a new name (which makes sense if you think of names as being trivial little identifiers that merely point to the really interesting stuff, the data):

```
% mv test.txt my_essay.txt
```

There is, however, a totally separate command for *copying* files: `cp`. This command works just like `mv`, except that it duplicates the original file and leaves it where it is:

```
% cp test.txt my_essay.txt
```

Finally, deleting a file is done using the `rm` command, which stands for *remove*:

```
% rm test.txt
```

Suppose you want to make a new directory to store this file and others. You'd do that with the `mkdir` (*make directory*) command:

```
% mkdir essays/history
```

This command creates a subdirectory called `"history"` inside the `"essays"` directory. As with most of the other commands you've seen, the arguments can be bare filenames (to refer to a target in the current directory), absolute paths (beginning with the `"/"` directory and specifying each subsequent step down the tree), or relative paths (as you just saw).

Deleting a directory is a little trickier. The command for deleting a directory is `rmdir`, or *remove directory*; but it only works on a directory that's empty:

```
% rmdir essays
rmdir:  essays:  Directory  ↲
   not empty
```

Now, you can go through the contents of your directory and painstakingly delete every single file in it using the `rm` command (or, for more convenience and more risk of things going badly wrong, `rm *`, which matches all filenames in the current directory); or, if you're in a hurry or just want to get things over with, you can use the `rm -rf` command to delete a directory and everything inside it all at once:

```
% rm -rf essays
```

Note, though, that doing it this way means you won't get any warnings or second chances this way (the `-f` flag suppresses them); and you'd better be sure that everything inside all the subdirectories of the directory you're deleting is really safe to delete. FreeBSD has no *Trash* or *Recycle Bin* equivalent - an `rm` command is forever.

Viewing Files

A graphical operating system in the modern day is designed to let you do just about anything with a click or two of the mouse. You're probably well accustomed to viewing the contents of a text document or a picture file by double-clicking on it; this causes the operating system to look up the installed application that's registered to be responsible for opening files with the specified type (generally defined by the three- or four-letter extension at the end of the file's name, such as .txt or .jpg or .html).

Unix, however, does not have this facility built into its architecture; whereas KDE behaves much like Windows or Mac OS X when it comes to file type associations and opener applications, in the CLI environment you have to use other means to see inside your files.

To show the contents of a plain text file, use the `cat` command (whose name stands for *catenate*):

```
% cat my_essay.txt
```

This is a test.

Now is the time for all good men to come to.

This is great for short files; but many of the text files you'll encounter are many kilobytes or even megabytes long (think log files). There's a variety of other commands that let you manage these larger files much more efficiently, such as `more`:

```
% more /var/log/maillog
```

The `more` program is a *pager*, which is a full-screen program that lets you move through the contents of a text file page by page, using the space bar to move forward and the W key to move back. Skip to the end by pressing [*Shift*+] (the > symbol), or the beginning with <). You can search for a string of characters by typing / followed by the string, then pressing [*Return*] or [*Enter*]. Press [*Q*] to quit.

Working within the textual shell, particularly through a remote terminal connection (for example over SSH), you're really only going to be able to look at plain text files; images, movies, and even text documents that are stored in a binary format (such as Microsoft Word files) are going to be unviewable unless you're working within a graphical environment like KDE.

If you are using KDE, however (KDE does include a terminal program called Konsole that gives you access to a FreeBSD desktop machine's CLI layer), you can open images, movies, and other media file types in external programs, provided you've got them installed. For example, the `xv` program lets you view pictures:

```
% xv Picture1.jpg
```

Movies can be viewed using the MPlayer program:

```
% mplayer Birthday.mov
```

Word documents can be opened using OpenOffice.org, a Microsoft Office-compatible suite available for free at the website of the same name. Indeed, just about any popular or generic file format has a program that can open it under Linux; and if a program exists for Linux, chances are that it can be compiled and installed on FreeBSD as well, if there isn't a binary package for it already. And if there isn't, FreeBSD's Linux compatibility layer will allow it to run Linux programs natively.

Getting more information

You've now got a pretty good idea of the equivalent commands that will allow you to move around the system and work with files just as you would on a desktop system with a mouse-driven GUI. Inevitably, though, an article this long can only scratch the surface of what's possible at the Unix command line. Whole textbook chapters can be written (as I can personally attest) on subjects like file permissions alone. You're going to need to continue your research on your own, preferably with a FreeBSD machine handy that you can learn on.

There are dozens of websites and books available on Unix and FreeBSD, and I would be remiss if I did not plug my own book, *FreeBSD Unleashed, Third Edition* (Sams Publishing, 2006).

Still, though, for the impatient there's no beating the online FreeBSD Handbook (*http://www.freebsd.org/handbook/basics.html*) for command-line theory and tutorials with details specific to the tcsh-centric, GNU-phobic world of FreeBSD (as opposed to the bash-centric, GNU-heavy world of Linux).

Also remember that the `man` command is available for any command in the FreeBSD system. Just type `man` *command* to read the documentation in the more pager that you saw earlier:

```
% man ls
```

Finally, check out the FreeBSD mailing lists (available at *http://www.freebsd.org/community/mailinglists.html*). Subscribing to one of these lists will give you a tap into the vibrant FreeBSD user community, letting you ask questions and hear others working their way through the same problems you're facing.

In the open-source world, there's no better problem-solving tool than the community itself, which is arguably the whole movement's primary strength. And if you're a part of the community, you might as well use it to your advantage!