4th Edition
Covers Version 5.0

The Complete FreeBSD

Documentation from the Source

Greg Lehey

O'REILLY®
COMMUNITY PRESS

In this chapter:

- *Users and groups*
- *Gaining access*
- *The KDE desktop*
- *The fvwm2 window manager*
- *Changing the X display*
- *Getting a shell*
- *Files and file names*
- *Differences from Microsoft*
- *The Emacs editor*
- *Stopping the system*

7

The tools of the trade

So now you have installed FreeBSD, and it successfully boots from the hard disk. If you're new to FreeBSD, your first encounter with it can be rather puzzling. You probably didn't expect to see the same things you know from other platforms, but you might not have expected what you see either:

```
    FreeBSD (freebie.example.org) (ttyv0)

login:
```

If you have installed *xdm*, you'll at least get a graphical display, but it still asks you to log in and provide a password. Where do you go from here?

There isn't space in this book to explain everything there is about working with FreeBSD, but in the following few chapters I'd like to make the transition easier for people who have prior experience with Microsoft platforms or with other flavours of UNIX. You can find a lot more information about these topics in *UNIX for the Impatient*, by Paul W. Abrahams and Bruce R. Larson, *UNIX Power Tools*, by Jerry Peek, Tim O'Reilly, and Mike Loukides, and *UNIX System Administration Handbook*, by Evi Nemeth, Garth Snyder, Scott Seebass, and Trent R. Hein. The third edition of this book also covers FreeBSD Release 3.2. See Appendix A, *Bibliography*, for more information.

If you've come from Microsoft, you will notice a large number of differences between UNIX and Microsoft, but in fact the two systems have more in common than meets the eye. Indeed, back in the mid-80s, one of the stated goals of MS-DOS 2.0 was to make it more UNIX-like. You be the judge of how successful that attempt was, but if you know

the MS-DOS command-line interface, you'll notice some similarities in the following sections.

In this chapter, we'll look at FreeBSD from the perspective of somebody with computer experience, but with no UNIX background. If you *do* have a UNIX background, you may still find it interesting.

If you're coming from a Microsoft platform, you'll be used to doing just about everything with a graphical interface. In this book I recommend that you use X and possibly a desktop, but the way you use it is still very different. FreeBSD, like other UNIX-like systems, places much greater emphasis on the use of text. This may seem primitive, but in fact the opposite is true. It's easier to point and click than to type, but you can express yourself much more accurately and often more quickly with a text interface.

As a result, the two most important tools you will use with FreeBSD are the *shell* and the *editor*. Use the shell to issue direct commands to the system, and the editor to prepare texts. We'll look at these issues in more detail in this chapter. In Chapter 8, *Taking control*, we'll look at other aspects of the system. First, though, we need to get access to the system.

Users and groups

Probably the biggest difference between most PC operating systems and FreeBSD also takes the longest to get used to: FreeBSD is a multi-user, multi-tasking system. This means that many people can use the system at once, and each can do several things at the same time. You may think "Why would I want to do that?." Once you've got used to this idea, though, you'll never want to do without it again. If you use the X Window System, you'll find that all windows can be active at the same time—you don't have to select them. You can monitor some activity in the background in another window while writing a letter, testing a program, or playing a game.

Before you can access a FreeBSD system, you must be registered as a *user*. The registration defines a number of parameters:

- A *user name*, also often called *user ID*. This is a name that you use to identify yourself to the system.
- A *password*, a security device to ensure that other people don't abuse your user ID. To log in, you need to specify both your user ID and the correct password. When you type in the password, nothing appears on the screen, so that people looking over your shoulder can't read it.

  It might seem strange to go to such security measures on a system that you alone use. The incidence of Internet-related security problems in the last few years has shown that it's not strange at all, it's just common sense. Microsoft systems are still subject to a never-ending series of security exploits. FreeBSD systems are not.

- A *shell*, a program that reads in your commands and executes them. MS-DOS uses the program *COMMAND.COM* to perform this function. UNIX has a large choice of shells: the traditional UNIX shells are the Bourne shell *sh* and the C shell *csh*, but FreeBSD also supplies *bash*, *tcsh*, *zsh* and others. I personally use the *bash* shell, and the examples in this book are based on it.
- A *home directory*. The system can have multiple users, so each one needs a separate directory in which to store his private files. Typically, users have a directory */home/**username***, where ***username*** is the name they use to log in. When you log in to the system, the shell sets the current directory to your home directory. In it, you can do what you want, and normally it is protected from access by other users. Many shells, including the *bash* shell used in these examples, use the special notation ~ (tilde) to represent the name of the home directory.
- A *group* number. UNIX collects users into *groups* who have specific common access permissions. When you add a user, you need to make him a member of a specific group, which is entered in the password information. Your group number indirectly helps determine what you are allowed to do in the system. As we'll see on page 181, your user and group determine what access you have to the system. You can belong to more than one group.

  Group numbers generally have names associated with them. The group names and numbers are stored in the file */etc/group*. In addition, this file may contain user IDs of users who belong to another group, but who are allowed to belong to this group as well.

  If you find the concept of groups confusing, don't worry about them. You can get by quite happily without using them at all. You'll just see references to them when we come to discuss file permissions. For further information, look at the man page for *group(5)*.

By the time you get here, you should have defined a user name, as recommended on page 94. If you haven't, you'll have to log in as `root` and create one as described there.

Gaining access

Once you have a user name, you can log in to the system. Already you have a choice: FreeBSD offers both *virtual terminals* and the X Window System. The former displays plain text on the monitor, whereas the latter uses the system's graphics capabilities. Once running, you can switch from one to the other, but you have the choice of which interface you use first. If you don't do anything, you get a virtual terminal. If you run *xdm*, you get X.

It's still relatively uncommon to use *xdm*, and in many instances you may not want X at all, for example if you're running the system as a server. As a result, we'll look at the ''conventional'' login first.

If you're logging in on a virtual terminal, you'll see something like this:

```
login: grog
Password:                                   password doesn't show on the screen
Last login: Fri Apr 11 16:30:04 from canberra
Copyright (c) 1980, 1983, 1986, 1988, 1990, 1991, 1993, 1994
        The Regents of the University of California.  All rights reserved.

FreeBSD 5.0-RELEASE (FREEBIE) #0: Tue Dec 31 19:08:24 CST 2002

Welcome to FreeBSD!

You have mail.
erase ^H, kill ^U, intr ^C, status ^T
Niklaus Wirth has lamented that, whereas Europeans pronounce his name
correctly (Ni-klows Virt), Americans invariably mangle it into
(Nick-les Worth).  Which is to say that Europeans call him by name, but
Americans call him by value.
=== grog@freebie (/dev/ttyv0) ~ 1 ->
```

There's a lot of stuff here. It's worth looking at it in more detail:

- The program that asks you to log in on a terminal window is called *getty*. It reads in your user ID and starts a program called *login* and passes the user ID to it.
- *login* asks for the password and checks your user ID.
- If the user ID and password are correct, *login* starts your designated shell.
- While starting up, the shell looks at a number of files. See the man page for your particular shell for details of what they are for. In this case, though, we can see the results: one file contains the time you last logged in, another one contains the *Message of the day* (*/etc/motd*), and a third one informs you that you have mail. The shell prints out the message of the day verbatim—in this case, it contains information about the name of the kernel and a welcome message. The shell also prints information on last login time (in this case, from a remote system) and whether you have mail.
- The line ''`erase ^H, kill ^U, intr ^C, status ^T`'' looks strange. It's telling you the current editing control characters. We'll look at these on page 131. At this point, the shell changes the current directory to your *home directory*. There is no output on the screen to indicate this.
- The shell runs the *fortune* program, which prints out a random quotation from a database of ''fortune cookies.'' In this case, we get a message about Niklaus Wirth, the inventor of the Pascal programming language.
- Finally, the last line is a prompt, the information that tells you that the shell is ready for input.

The prompt illustrates a number of things about the UNIX environment. By default, *sh* and friends prompt with a `$`, and *csh* and friends prompt with a `%`. You can change it to just about anything you want with the UNIX shells. You don't have to like my particular version, but it's worth understanding what it's trying to say.

The first part, `===`, is just to make it easier to find in a large list on an X display. An *xterm* window on a high resolution X display can contain up to 120 lines, and searching for command prompts can be non-trivial.

Next, `grog@freebie` is my user ID and the name of system on which I am working, in the RFC 2822 format used for mail IDs. Multiple systems and multiple users can all be present on a single X display. This way, I can figure out which user I am and what system I am running on.

/dev/ttyv0 is the name of the terminal device. This can sometimes be useful.

~ is the name of the home directory. Most shells, but not all of them, support this symbolism.

`1` is the prompt number. Each time you enter a command, it is associated with this number, and the prompt number is incremented. One way to re-execute the command is to enter `!!1` (two exclamation marks and the number of the command). We'll look at more comfortable ones on page 131.

To start X from a virtual terminal shell, use the *startx* command:

```
$ startx
```

If you use *xdm*, you bypass the virtual terminals and go straight into X. Enter your user name and password to the login prompt or the *xdm* login screen, and press **Enter**. If you use the *xdm* login, you'll go straight into X.

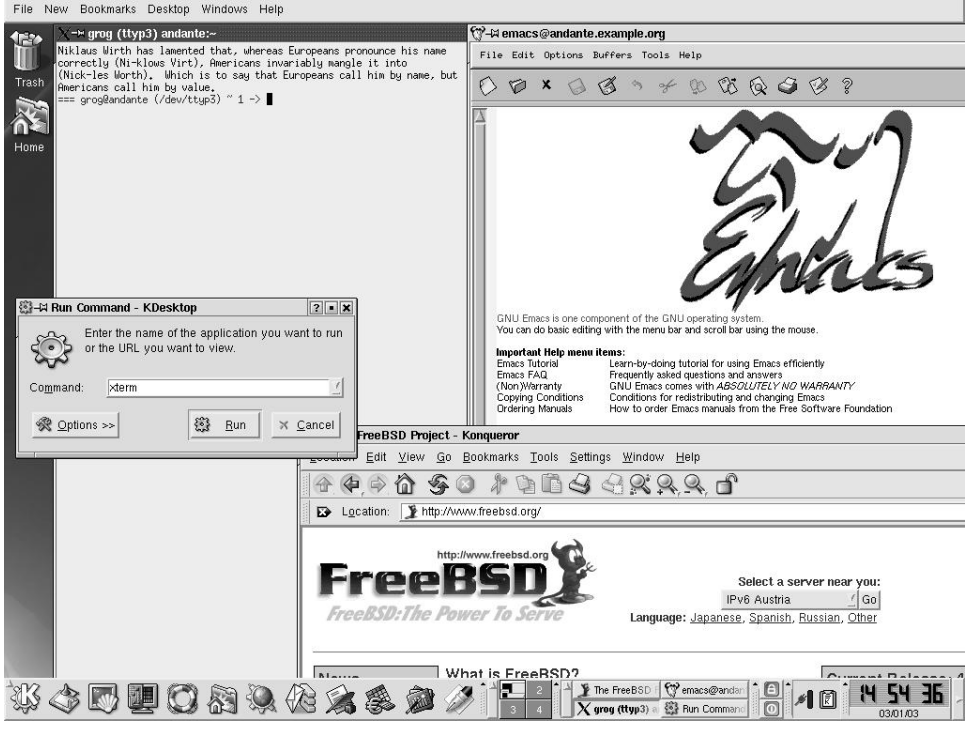


Figure 7-1: KDE display

Either way, assuming that you've installed and configured *kde*, you'll get a display similar to that in Figure 7-1. This example includes four windows that are not present on startup. On startup the central part of the screen is empty. We'll look at the windows further below.

The KDE desktop

KDE is a complicated system, and good documentation is available at *http://www.kde.org/documentation/*. Once you have KDE running, you can access the same information via the help icon on the panel at the bottom (the life ring icon). The following description gives a brief introduction.

The KDE display contains a number of distinct areas. At the top is an optional menu, at the bottom an almost optional *panel*, and the middle of the screen is reserved for windows.

The Desktop Menu

The *Desktop Menu* is at the very top of the screen. It provides functionality that is not specific to a particular application. Select the individual categories with the mouse. For example, the `New` menu looks like this:

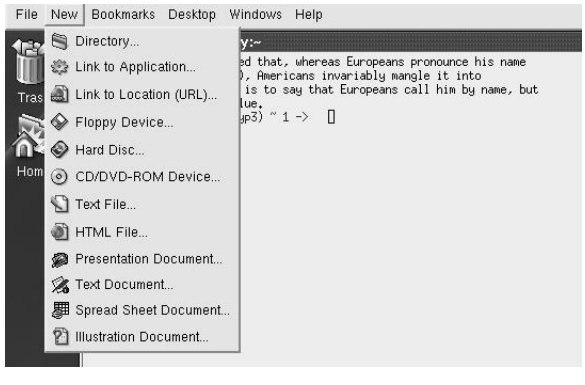


Figure 7-2: KDE desktop menu

As the menu indicates, you can use these menus to create new files.

The Panel

At the bottom of the screen is the panel, which consists of a number of fields. The left-hand section is used for starting applications.

The stylized letter K at the extreme left is the *Application Starter*. When you select it, a

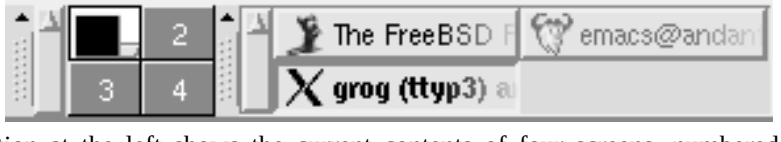
long vertical menu appears at the left of the screen and allows you to start programs (‘‘applications’’) or access just about any other function.

Next comes an icon called ‘‘show desktop.’’ This is a convenient way to iconify all the windows currently on the desktop.

The remaining icons on this part of the panel represent various applications.

- The *konsole* terminal emulator.
- The *command center*, which you use to configure KDE.
- The help system.
- Access to the home directory with the browser *konqueror*.
- Access to the Web, also with the browser *konqueror*.
- The *Kmail* MUA.
- The *KWord* word processor, which can understand Microsoft Word documents.
- The *Kspread* spreadsheet.
- The *Kpresenter* presentation package.
- The *Kate* editor.

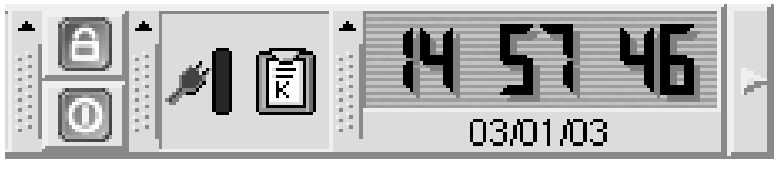
The next section of the panel contains some control buttons and information about the current desktop layout:



The section at the left shows the current contents of four screens, numbered 1 to 4. Screen 1 is the currently displayed screen; you can select one of the others by moving the cursor in the corresponding direction, or by selecting the field with the mouse.

To the right of that are icons for the currently active windows. The size expands and contracts depending on the number of different kinds of window active. If you select one of these icons with the left mouse button, it will iconify or deiconify (‘‘minimize’’ or ‘‘maximize’’) the window. If you have multiple *xterm*s active, you will only have one icon. In this case, if you select the icon, you will get another pop-up selection menu to allow you to choose the specific window.

The right part of the panel contains a further three fields:



- The first one shows a stylized padlock (for locking the session when you leave the machine; unlock by entering your password) and a stylized off switch, for logging out of the session.
- The next section shows a stylized power connector, which displays the current power status of the machine, and a clipboard.
- The right side shows a digital clock.

Probably the most useful part of this section of the panel is not very obvious: the right-pointing arrow allows you to remove the panel if you find it's in the way. The entire panel is replaced by a single left-pointing arrow at the extreme right of the display.

Using the mouse

By default, *kde* only uses the left and the right mouse buttons. In general, the left button is used to select a particular button, and the right button is used for an auxiliary menu.

Manipulating windows

You'll notice that each window has a frame around it with a number of features. In X terminology, they're called *decorations*. Specifically:

- There's a *title bar* with the name of the program. If you select the bar itself, you raise the window above all others. If you hold down the button on the title bar, you can move the window.
- At the left of the title bar there is an X logo. If you select this logo, you get a menu of window operations.
- At the right of the title bar, there are three buttons that you can select. The left one iconifies the window, the middle one *maximizes* the window, making it take up the entire screen, and the one on the right kills the application. If the window is already maximized, the middle button restores it to its previous size.
- You can select any corner of the window, or any of the other edges, to change the size of the window.

The fvwm2 window manager

If you come from a conventional PC background, you shouldn't have much difficulty with KDE. It's a relatively complete, integrated environment. But it isn't really UNIX. If you come from a UNIX environment, you may find it too all-encompassing. You may also find that there are significant delays when you start new applications.

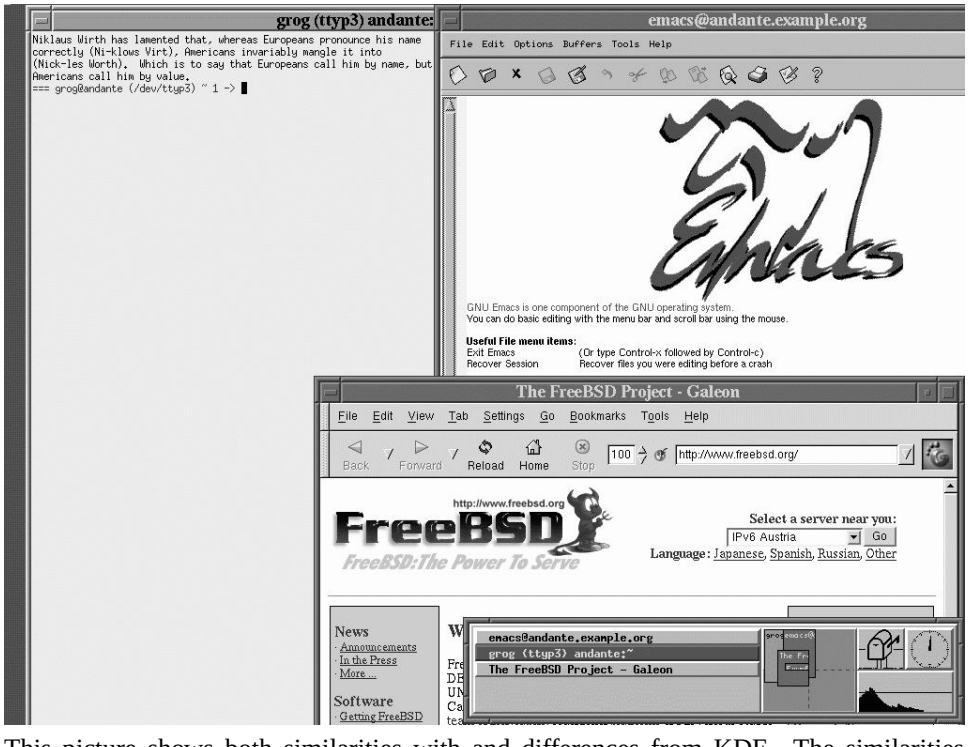
UNIX has a very different approach to windows. There is no desktop, just a window manager. It takes up less disk space, less processor time, and less screen real estate. By default, XFree86 comes with the *twm* window manager, but that's really a little primitive. With modern machines, there's no reason to choose such a basic window manager. You may, however, find that *fvwm2* is more your style than KDE.

Starting fvwm2

Like KDE, you install *fvwm2* from the Ports Collection. It's not designed to work completely correctly out of the box, though it does work. As with KDE, the first thing you need to do is to create a *.xsession* or *.xinitrc* file, depending on whether you're running *xdm*. It must contain at least the line:

```
fvwm2
```

Start X the same way you did for KDE. This time you see, after starting the same applications as before:

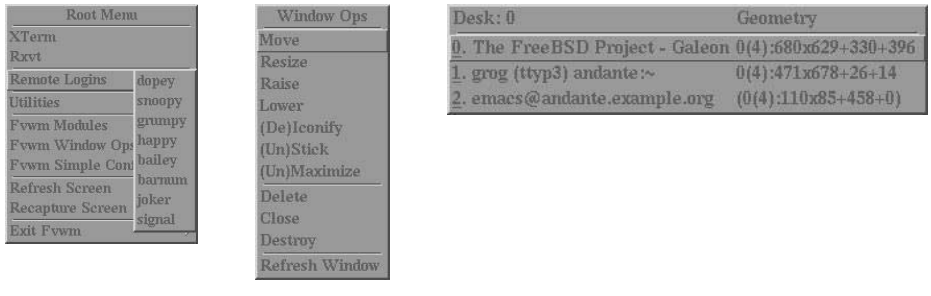


This picture shows both similarities with and differences from KDE. The similarities include:

- Each window has a frame and a title. The exact form of the decorations is different, but the purpose is the same. There is no ''close application'' button: for most UNIX applications, you should get the program to exit rather than killing it.
- There is a task bar at the bottom right, taking up only half the width of the screen. The currently active window (the *xterm* at the left in this example) is highlighted.
- The default *fvwm2* display also has four screens, and the task bar shows the position of the windows on the task bar.

Still, there are a number of differences as well:

- Unless you have a top-end machine, it's *much* faster in what it does.
- The background (the *root window*) doesn't have any pattern; it's just a grey cross-hatch.
- You can move from one screen to the other using the cursor, and windows can overlap. In this example, the *galeon* web browser window goes down to the screen below, and the Emacs window goes over all four screens, as the display on the task bar shows. With KDE, the only way to display the rest of these windows is to move the window.
- Paradoxically, you can do a lot more with the mouse. On the root window, the left mouse button gives you a menu for starting various programs, both locally and remotely, and also various window utilities. The middle button gives you direct access to the window manipulation utilities, and the right button gives a drop-down list to select any of the currently active windows:



The menus above show one of the problems: look at those system names on the left submenu (*dopey*, *snoopy* and friends). They don't exist on our sample network, and the chance of them existing on your network are pretty low as well. They're hard-coded in the sample configuration file, */usr/X11R6/etc/system.fvwm2rc*. To use *fvwm2* effectively, you'll have to modify the configuration file. The best thing to do is to make a copy of */usr/X11R6/etc/system.fvwm2rc* in your own directory, as *~/.fvwm2/.fvwm2rc*. Then you can have lots of fun tweaking the file to do exactly what you want it to do. Clearly, KDE is easier to set up.

Changing the X display

When you set up your *XF86Config* file, you may have specified more than one resolution. For example, on page 107 we selected the additional resolution 1024x768 pixels. When you start X, it automatically selects the first resolution, in this case 640x480. You can change to the previous resolution (the one to the left in the list) by pressing the **Ctrl**-**Alt**-**Keypad -** key, and to the following resolution (the one to the right in the list) with **Ctrl**-**Alt**-**Keypad +**. *Keypad* + and *Keypad* - refer to the **+** and **-** symbols on the numeric keypad at the right of the keyboard; you can't use the **+** and **-** symbols on

the main keyboard for this purpose. The lists wrap around: in our example, if your current resolution is 640x480, and you press **Ctrl**-**Alt**-**Keypad -**, the display changes to 1024x768. It's a very good idea to keep the default resolution at 640x480 until you have debugged your *XF86Config* parameters: 640x480 almost always works, so if your display is messed up, you can just switch back to a known good display with a single keystroke.

Selecting pixel depth

You can configure most display boards to display a number of different pixel depths (a different number of bits per pixel, which translates to a different number of colours). When you start X, however, it defaults to 8 bits per pixel (256 colours), which is a very poor rendition. To start it with a different number, specify the number of planes. For example, to start with 32 bits per pixel (4,294,967,296 colours), enter:

```
$ startx -- -bpp 32
```

With older display boards, which had relatively limited display memory, there was a tradeoff between maximum resolution and maximum pixel depth. Modern display cards no longer have this limitation. We'll look at this issue in more detail on page 522.

Getting a shell

As we saw at the beginning of the chapter, your main tools are the shell and the editor, and that's what we saw on the sample screens. But when you start X, they're not there: you need to start them.

In KDE, you have two ways to start a terminal window:

- You can select the icon showing a monitor with a shell in front of it, third from the left at the bottom of the example above. This starts the *konsole* terminal emulator.
- You can start an *xterm* by pressing **Alt-F2**. You see a window like the one in the centre left of Figure 7-1, enter the text `xterm` (as shown) and press `Run` or the **Enter** key.

Obviously the first is the intended approach, and it's easier. Nevertheless, I recommend using *xterm* at least until you're sure you want to stick with *kde*: there are some subtle differences, and *konsole* is intended to work with *kde* only. If you do stick with KDE, you should change the configuration of the *konsole* button to start *xterm* instead; that's relatively straightforward.

In *fvwm2*, you start an *xterm* from the left mouse menu, as shown above.

Shell basics

The most basic thing you can do with the shell is to start a program. Consider program names to be commands: like you might ask somebody to ‘‘wash the dishes’’ or ‘‘mow the lawn,’’ you can tell the shell to ‘‘remove those files’’:

```
$ rm file1 file2 file3
```

This starts a program called *rm* (remove), and gives it a list of three file names to remove.

If you’re removing a whole lot of files, this could take a while. Consider removing the entire directory hierarchy */usr/obj*, which is created when building a new version of the system (see page 595). This directory hierarchy contains about 15,000 files and directories, and it’ll take a while to remove it. You can do this with the following command:

```
# rm -rf /usr/obj &
```

In this example, we have a couple of options led in by a hyphen (`-`) and also the character `&` at the end of the line.

- The `r` option tells *rm* to *recursively* descend into subdirectories. If you didn’t specify this, it would remove all files in the directory */usr/obj* and then exit, complaining that it can’t delete directories.
- The `f` (*force*) option tells *rm* to continue on error; otherwise if anything goes wrong, it will stop.
- The `&` character at the end of the line tells the shell (not *rm*) to continue after starting the program. It can run for some time, and there’s no need to wait for it.

Options

In the previous example, we saw a couple of options. By convention, they come between the command name and other parameters, and they’re identified because they start with a hyphen character (`-`). There’s a lot of variation, though, depending on the individual program.

- Sometimes, as in the previous example, options consist of a single letter and can often be joined together.
- Some programs, like *tar* and *ps*, don’t insist on the hyphen lead-in. In Chapter 8, we’ll see the command:

  ```
  # ps waux
  ```

This command could equally well be written:

```
# ps -waux
```

You may also come across programs that refuse to accept the hyphen at all.

- Sometimes options can have values. For example, in Chapter 23 we'll see:

  ```
  # tcpdump -i ppp0 host hub.freebsd.org
  ```

  Here, `ppp0` is an argument to the `i` option. In some cases, it must be written with a space; in others, it must be written without a space; and in others again, it can be written either way. Pay attention to this detail when reading man pages.

- In other cases, they can be keywords, in which case they need to be written separately. The GNU project is particularly fond of this kind of option. For example, when building the system you may see compiler invocations like these:

  ```
  cc -O -pipe -Dinline=rpcgen_inline -Wall -Wno-format-y2k -Wno-uninitialized \
  -D__FBSDID=__RCSID -c /usr/src/usr.bin/rpcgen/rpc_main.c
  ```

  With the exception of the last parameter, all of these texts are options, as the hyphen suggests.

- Options are specific to particular commands, though often several commands attempt to use the same letters to mean the same sort of thing. Typical ones are `v` for verbose output, `q` for quiet output (i.e. less than normal).

- Sometimes you can run into problems when you supply a parameter that looks like an option. For example, how do you remove a file called *-rf*? There are a number of solutions for this problem. In this example, you could write:

  ```
  $ rm ./-rf
  ```

  This is an alternative file naming convention that we'll look at again on page 126.

Shell parameters

When you invoke a program with the shell, it first *parses* the input line before passing it to the program: it turns the line into a number of parameters (called *arguments* in the C programming language). Normally the parameters are separated by *white space*, either a space or a tab character. For example, consider the previous example:

```
$ rm file1 file2 file3
```

the program receives four arguments, numbered 0 to 3:

Table 7-1: Program arguments

Argument	Value
0	rm
1	file1
2	file2
3	file3

What happens if you want to pass a name with a space? For example, you might want to look for the text "`Mail rejected`" in a log file. UNIX has a standard program for looking for text, called *grep*. The syntax is:

```
grep expression files
```

Argument 1 is the expression; all additional arguments are the names of files to search. We could write:

```
$ grep Mail rejected /var/log/maillog
```

but that would try to look for the text `Mail` in the files *rejected* (probably causing an error message that the file did not exist) and */var/log/maillog* (where just about every line contains the text `Mail`). That's not what we want. Instead, we do pretty much what I wrote above:

```
$ grep "Mail rejected" /var/log/maillog
```

In other words, if we put quote characters `""` around a group of words, the shell will interpret them as a single parameter. The first parameter that is passed to *grep* is `Mail rejected`, not `"Mail rejected"`.

This behaviour of the shell is a very good reason not to use file names with spaces in them. It's perfectly legitimate to embed spaces into UNIX file names, but it's a pain to use. If you want to create a file name that contains several words, for example *All files updated since last week*, consider changing the spaces to underscores: *All_files_updated_since_last_week*.

It's even more interesting to see what happens when you pass a globbing character to a program, for example:

```
$ cc -o foo *.c
```

This invocation compiles all C source files (*.*c*) and creates a program *foo*. If you do this with Microsoft, the C compiler gets four parameters, and it has to find the C source files itself. In UNIX, the shell expands the text *.*c* and replaces it with the names of the source files. If there are thirty source files in the directory, it will pass a total of 33 parameters to the compiler.

Fields that can contain spaces

The solution to the ‘‘Mail rejected’’ problem isn’t ideal, but it works well enough as long as you don’t have to handle fields with blanks in them too often. In many cases, though, particularly in configuration files, fields with blanks are relatively common. As a result, a number of system configuration files use a colon (`:`) as a delimiter. This looks very confusing at first, but it turns out not to be as bad as the alternatives. We’ll see some examples in the `PATH` environment variable on page 130, in the password file on page 144, and in the login class file on page 570.

Files and file names

Both UNIX and Microsoft environments store disk data in *files*, which in turn are placed in *directories*. A file may be a directory: that is, it may contain other files. The differences between UNIX and Microsoft start with *file names*. Traditional Microsoft file names are rigid: a file name consists of eight characters, possibly followed by a period and another three characters (the so-called *file name extension*). There are significant restrictions on which characters may be used to form a file name, and upper and lower case letters have the same meaning (internally, Microsoft converts the names to UPPER CASE). Directory members are selected with a backslash (\), which conflicts with other meanings in the C programming language—see page 138 for more details.

FreeBSD has a very flexible method of naming files. File names can contain any character except /, and they can be up to 255 characters long. They are *case-sensitive*: the names *FOO*, *Foo* and *foo* are three different names. This may seem silly at first, but any alternative means that the names must be associated with a specific character set. How do you upshift the German name *ungleichmäßig*? What if the same characters appear in a Russian name? Do they still shift the same? The exception is because the / character represents directories. For example, the name */home/fred/longtext-with-a-long-name* represents:

1. First character is a /, representing the *root file system*.
2. *home* is the name of a directory in the root file system.
3. *fred* is the name of a directory in */home*.
4. The name suggests that *longtext-with-a-long-name* is probably a file, not a directory, though you can’t tell from the name.

As a result, you can’t use / in a file name. In addition, binary 0s (the ASCII `NUL` character) can confuse a lot of programs. It’s almost impossible to get a binary 0 into a file name anyway: that character is used to represent the end of a string in the C programming language, and it’s difficult to input it from the keyboard.

Case sensitivity no longer seems as strange as it once did: web browsers have made UNIX file names more popular with *Uniform Resource Indicators* or *URIs*, which are derived from UNIX names.

File names and extensions

The Microsoft naming convention (name, period and extension) seems similar to that of UNIX. UNIX also uses extensions to represent specific kinds of files. The difference is that these extensions (and their lengths) are implemented by convention, not by the file system. In Microsoft, the period between the name and the extension is a typographical feature that only exists at the display level: it's not part of the name. In UNIX, the period is part of the name, and names like *foo.bar.bazzot* are perfectly valid file names. The system doesn't assign any particular meaning to file name extensions; instead, it looks for *magic numbers*, specific values in specific places in the file.

Relative paths

Every directory contains two directory entries, . and .. (one and two periods). These are *relative directory entries*: . is an alternative way to refer to the current directory, and .. refers to the parent directory. For example, in */home/fred*, . refers to */home/fred*, and .. refers to */home*. The root directory doesn't have a parent directory, so in this directory only, .. refers to the same directory. We'll see a number of cases where this is useful.[1]

Globbing characters

Most systems have a method of representing groups of file names and other names, usually by using special characters for representing an abstraction. The most common in UNIX are the characters `*`, `?` and the square brackets `[]`. UNIX calls these characters *globbing characters*. The Microsoft usage comes from UNIX, but the underlying file name representation makes for big differences. Table 7-2 gives some examples.

Table 7-2: Globbing examples

Name	Microsoft meaning	UNIX meaning
`CONFIG.*`	All files with the name *CONFIG*, no matter what their extension.	All files whose name starts with *CONFIG.*, no matter what the rest is. Note that the name contains a period.
`CONFIG.BA?`	All files with the name *CONFIG* and an extension that starts with *BA*, no matter what the last character.	All files that start with *CONFIG.BA* and have one more character in their name.
`*`	Depending on the Microsoft version, all files without an extension, or all files.	All files.
`*.*`	All files with an extension.	All files that have a period in the middle of their name.

1. Interestingly, the Microsoft file systems also have this feature.

`foo[127]`	In older versions, invalid. In newer versions with long file name support, the file with the name *foo[127]*.	The three files *foo1*, *foo2* and *foo7*.

Input and output

Most programs either read input data or write output data. To make it easier, the shell usually starts programs with at least three open files:

- *Standard input*, often abbreviated to *stdin*, is the file that most programs read to get input data.
- *Standard output*, or *stdout*, is the normal place for programs to write output data.
- *Standard error output*, or *stderr*, is a separate file for programs to write error messages.

With an interactive shell (one that works on a terminal screen, like we're seeing here), all three files are the same device, in this case the terminal you're working on.

Why two output files? Well, you may be collecting something important, like a backup of all the files on your system. If something goes wrong, you want to know about it, but you don't want to mess up the backup with the message.

Redirecting input and output

But of course, even if you're running an interactive shell, you don't want to back up your system to the screen. You need to change *stdout* to be a file. Many programs can do this themselves; for example, you might make a backup of your home directory like this:

```
$ tar -cf /var/tmp/backup ~
```

This creates (option `c`) a file (option `f`) called */var/tmp/backup*, and includes all the files in your home directory (`~`). Any error messages still appear on the terminal, as *stderr* hasn't been changed.

This syntax is specific to *tar*. The shell provides a more general syntax for redirecting input and output streams. For example, if you want to create a list of the files in your current directory, you might enter:

```
$ ls -l
  drwxr-xr-x   2 root  wheel    512 Dec 20 14:36 CVS
  -rw-r--r--   1 root  wheel   7928 Oct 23 12:01 Makefile
  -rw-r--r--   5 root  wheel    209 Jul 26 07:11 amd.map
  -rw-r--r--   5 root  wheel   1163 Jan 31  2002 apmd.conf
  -rw-r--r--   5 root  wheel    271 Jan 31  2002 auth.conf
  -rw-r--r--   1 root  wheel    741 Feb 19  2001 crontab
  -rw-r--r--   5 root  wheel    108 Jan 31  2002 csh.cshrc
  -rw-r--r--   5 root  wheel    482 Jan 31  2002 csh.login
(etc)
```

You can redirect this output to a file with the command:

```
$ ls -l > /var/tmp/etclist
```

This puts the list in the file */var/tmp/etclist*. The symbol > tells the shell to redirect *stdout* to the file whose name follows. Similarly, you could use the < to redirect *stdin* to that file, for example when using *grep* to look for specific texts in the file:

```
$ grep csh < /var/tmp/etclist
  -rw-r--r--   5 root  wheel    108 Jan 31  2002 csh.cshrc
  -rw-r--r--   5 root  wheel    482 Jan 31  2002 csh.login
  -rw-r--r--   5 grog  lemis    110 Jan 31  2002 csh.logout
```

In fact, though, there's a better way to do that: what we're doing here is feeding the output of a program into the input of another program. That happens so often that there's a special method of doing it, called *pipes*:

```
$ ls -l | grep csh
  -rw-r--r--   5 root  wheel    108 Jan 31  2002 csh.cshrc
  -rw-r--r--   5 root  wheel    482 Jan 31  2002 csh.login
  -rw-r--r--   5 grog  lemis    110 Jan 31  2002 csh.logout
```

The | symbol causes the shell to start two programs. The first has a special file, a *pipe*, as the output, and the second has the same pipe as input. Nothing gets written to disk, and the result is much faster.

A typical use of pipes are to handle quantities of output data in excess of a screenful. You can pipe to the *less*[1] program, which enables you to page backward and forward:

```
$ ls -l | less
```

Another use is to sort arbitrary data:

```
$ ps aux | sort -n +1
```

This command takes the output of the *ps* command and sorts it by the numerical (`-n`) value of its *second* column (`+1`). The first column is numbered 0. We'll look at *ps* on page 148.

Environment variables

The UNIX programming model includes a concept called *environment variables*. This rather unusual sounding name is simply a handy method of passing relatively long-lived information of a general nature from one program to another. It's easier to demonstrate the use than to describe. Table 7-3 takes a look at some typical environment variables. To set environment variables from Bourne-style shells, enter:

1. Why *less*? Originally there was a program called *more*, but it isn't as powerful. *less* is a new program with additional features, which proves beyond doubt that *less* is *more*.

```
$ export TERM=xterm
```

This sets the value of the `TERM` variable to `xterm`. The word `export` tells the shell to pass this information to any program it starts. Once it's exported, it stays exported. If the variable isn't exported, only the shell can use it.

Alternatively, if you want to set the variable only once when running a program, and then forget it, you can set it at the beginning of a command line:

```
$ TERM=xterm-color mutt
```

This starts the *mutt* mail reader (see page 474) with *xterm*'s colour features enabled.

For *csh* and *tcsh*, set environment variables with:

```
% setenv TERM xterm
```

To start a process with these variables, enter:

```
% env xterm-color mutt
```

Table 7-3: Common environment variables

Name	Purpose
`BLOCKSIZE`	The size of blocks that programs like *df* count. The default is 512 bytes, but it's often more convenient to use 1024 or even 1048576 (1 MB).
`DISPLAY`	When running X, the name of the X server. For a local system, this is typically `unix:0`. For remote systems, it's in the form *system-name*:*server-number*.*screen-number*. For the system *bumble.example.org*, you would probably write `bumble.example.org:0`.
`EDITOR`	The name of your favourite editor. Various programs that start editors look at this variable to know which editor to start.
`HOME`	The name of your home directory.
`LANG`	The *locale* that you use. This should be the name of a directory in */usr/share/locale*.
`MAIL`	Some programs use this variable to find your incoming mail file.
`MANPATH`	A list of path names, separated by colons (`:`), that specifies where the *man* program should look for man pages. A typical string might be `/usr/share/man:/usr/local/man`, and specifies that there are man pages in each of the directories */usr/share/man* and */usr/local/man*.
`NTAPE`	The name of the non-rewinding tape device. See page 252 for more details.

Name	Purpose
PATH	A list of path names, separated by colons (:), that specifies where the shell should look for executable programs if you specify just the program name.
PS1	In Bourne-style shells, this is the prompt string. It's usually set to **$**, but can be changed. See page 114 for a discussion of a possible prompt for *bash*.
PS2	In Bourne-style shells, this is the prompt string for continuation lines. It's usually set to **>**.
SHELL	The name of the shell. Some programs use this for starting a shell.
TAPE	The name of the rewinding tape device. See page 252 for more details.
TERM	The type of terminal emulation you are using. This is very important: there is no other way for an application to know what the terminal is, and if you set it to the wrong value, full-screen programs will behave incorrectly.
TZ	Time zone. This is the name of a file in */usr/share/zoneinfo* that describes the local time zone. See the section on timekeeping on page 155 for more details.

Note particularly the PATH variable. One of the most popular questions in the FreeBSD-questions mailing list is ''I have compiled a program, and I can see it in my directory, but when I try to run it, I get the message ''command not found.'' This is usually because PATH does not include the current directory.

> It's good practice *not* to have your current directory or your home directory in the PATH: if you do, you can be subject to security compromises. For example, somebody could install a program called *ps* in the directory */var/tmp*. Despite the name, the program might do something else, for example remove all files in your home directory. If you change directory to */var/tmp* and run *ps*, you will remove all files in your home directory. Obviously much more subtle compromises are possible.
>
> Instead, run the program like this:
>
> ```
> $ ./program
> ```

You should set your PATH variable to point to the most common executable directories. Add something like this to your *.profile* file (for Bourne-style shells):

```
PATH=/usr/bin:/usr/local/bin:/usr/sbin:/bin:/sbin:/usr/X11R6/bin
export PATH
```

This variable is of great importance: one of the leading problems that beginners have is to have an incorrect PATH variable.

Printing out environment variables

So you can't start a program, and you're wondering whether your `PATH` environment variable is set correctly. You can find out with the *echo* command:

```
$ echo $PATH
/bin:/usr/bin
```

The `$` at the beginning of `$PATH` tells the shell to substitute the value of the environment variable for its name. Without this, the shell has no way of knowing that it's an environment variable, so it passes the text `PATH` to *echo*, which just prints it out.

If you want to print out all the environment variables, use the *printenv* command:

```
$ printenv | sort
BLOCKSIZE=1048576
CLASSPATH=/usr/local/java/lib:/usr/local/java/lib/classes.zip:/home/grog/netscape/
CVSROOT=/home/ncvs
DISPLAY=freebie:0
EDITOR=emacs
HOME=/home/grog
PAGER=less
PATH=.:/usr/bin:/usr/sbin:/bin:/sbin:/usr/X11R6/bin:/usr/local/bin:/usr/local/sbin
XAUTHORITY=/home/grog/.Xauthority
```

This example sorts the variables to make it easier to find them. In all probability, you'll find many more variables.

Command line editing

Typing is a pain. If you're anything like me, you're continually making mistakes, and you may spend more time correcting typing errors than doing the typing in the first place. It's particularly frustrating when you enter something like:

```
$ groff -rex=7.5 -r$$ -rL -rW -rN2 -mpic tmac.M unixerf.mm
troff: fatal error: can't open 'unixerf.mm': No such file or directory
```

This command *should* create the PostScript version of this chapter, but unfortunately I messed up the name of the chapter: it should have been *unixref.mm*, and I typed *unixerf.mm*.

> Yes, I know this looks terrible. In fact, UNIX has ways to ensure you almost never need to write commands like this. The command I really use to format this chapter is "`make unixref`".

It would be particularly frustrating if I had to type the whole command in again. UNIX offers a number of ways to make life easier. The most obvious one is so obvious that you tend to take it for granted: the **Backspace** key erases the last character you entered. Well, most of the time. What if you're running on a machine without a **Backspace** key? You won't have that problem with a PC, of course, but a lot of workstations have a **DEL** key instead of a **Backspace** key. UNIX lets you specify what key to use to erase the last character entered. By default, the erase character really is **DEL**, but the shell startup changes it and prints out a message saying what it has done:

```
erase ^H, kill ^U, intr ^C, status ^T
```

in the example on page 113. **^H** (**Ctrl-H**) is an alternative representation for **Backspace**.

The three other functions `kill`, `intr`, and `status` perform similar editing functions. `kill` erases the whole line, and `intr` stops a running program.

> More correctly, `intr` sends a *signal* called `SIGINT` to the process. This normally causes a program to stop.

You'll notice that it is set to **Ctrl-C**, so its function is very similar to that of the MS-DOS **Break** key. `status` is an oddball function: it doesn't change the input, it just displays a statistics message. *bash* doesn't in fact use it: it has a better use for **Ctrl-T**.

In fact, these control characters are just a few of a large number of control characters that you can set. Table 7-4 gives an overview of the more common control characters. For a complete list, see the man page *stty(1)*.

Table 7-4: Terminal control characters

Name	Default	Function
`CR`	`\r`	Go to beginning of line. Normally, this also terminates input (in other words, it returns the complete line to the program, which then acts on the input).
`NL`	`\n`	End line. Normally, this also terminates input.
`INTR`	`Ctrl-C`	Generate a `SIGINT` signal. This normally causes the process to terminate.
`QUIT`	`Ctrl-|`	Generate a `SIGQUIT` signal. This normally causes the process to terminate and *core dump*, to save a copy of its memory to disk for later analysis.
`ERASE`	`DEL`	Erase last character. FreeBSD sets this to **Backspace** on login, but under some unusual circumstances you might find it still set to **DEL**.
`KILL`	`Ctrl-U`	Erase current input line.
`EOF`	`Ctrl-D`	Return end-of-file indication. Most programs stop when they receive an EOF.
`STOP`	`Ctrl-S`	Stop output. Use this to examine text that is scrolling faster than you can read.
`START`	`Ctrl-Q`	Resume output after stop.
`SUSP`	`Ctrl-Z`	Suspend process. This key generates a `SIGTSTP` signal when typed. This normally causes a program to be suspended. To restart, use the *fg* command.

Name	Default	Function
DSUSP	Ctrl-Y	Delayed suspend. Generate a SIGTSTP signal when the character is read. Otherwise, this is the same as SUSP.
REPRINT	Ctrl-R	Redisplay all characters in the input queue (in other words, characters that have been input but not yet read by any process). The term "print" recalls the days of harcopy terminals. Many shells disable this function.
DISCARD	Ctrl-O	Discard all terminal output until another DISCARD character arrives, more input is typed or the program clears the condition.

To set these characters, use the *stty* program. For example, if you're used to erasing the complete input line with **Ctrl-X**, and specifying an end-of-file condition with **Ctrl-Z**, you could enter:

```
$ stty susp \377 kill ^X eof ^Z
```

You need to set SUSP to something else first, because by default it is Ctrl-Z, so the system wouldn't know which function to perform if you press ^Z.

> The combination \377 represents the character octal 377 (this notation comes from the C programming language, and its origin is lost in the mists of time, back in the days when UNIX ran on PDP-11s). This character is the "null" character that turns off the corresponding function. System V uses the character \0 for the same purpose.

In this particular case, ^X really does mean the character ^ followed by the letter X, and not **Ctrl-X**, the single character created by holding down the **Control** character and pressing **X** at the same time.

Command history and other editing functions

Nowadays, most shells supply a *command history* function and additional functionality for editing it. We'll take a brief look at these features here—for more details, see the man pages for your shell.

Shell command line editing has been through a number of evolutionary phases. The original Bourne shell supplied no command line editing at all, though the version supplied with FreeBSD gives you many of the editing features of more modern shells. Still, it's unlikely that you'll want to use the Bourne shell as your shell: *bash*, *ksh*, and *zsh* are all compatible with the Bourne shell, but they also supply better command line editing.

The next phase of command line editing was introduced with the C shell, *csh*. By modern standards, it's also rather pitiful. It's described in the *csh* man page if you really want to know. About the only part that is still useful is the ability to repeat a previous command with the !! construct. Modern shells supply command line editing that resembles the editors *vi* or *Emacs*. In *bash*, *sh*, *ksh*, and *zsh* you can make the choice by entering:

```
$ set -o emacs                  for Emacs-style editing
$ set -o vi                     for vi-style editing
```

In *tcsh*, the corresponding commands are:

```
% bind emacs
% bind vi
```

Normally you put one of these commands in your startup file.

In *Emacs* mode, you enter the commands simply by typing them in. In *vi* mode, you have to press **ESC** first. Table 7-5 shows an overview of the more typical Emacs-style commands in *bash*. Many other shells supply similar editing support.

As the name suggests, the *Emacs* editor understands the same editing characters. It also understands many more commands than are shown here. In addition, many X-based commands, including web browsers, understand some of these characters.

Table 7-5: Emacs editing characters

Key	Function
Ctrl-A	Move to the beginning of the line.
LeftArrow	Move to previous character on line.
Ctrl-B	Move to previous character on line (alternative).
Ctrl-D	Delete the character under the cursor. Be careful with this character: it's also the shell's end-of-file character, so if you enter it on an empty line, it stops your shell and logs you out.
Ctrl-E	Move to the end of the line.
RightArrow	Move to next character on line.
Ctrl-F	Move to next character on line (alternative).
Ctrl-K	Erase the rest of the line. The contents are saved to a *ring buffer* of erased text and can be restored, possibly elsewhere, with **Ctrl-Y**.
Ctrl-L	Erase screen contents (shell) or redraw window (*Emacs*).
DownArrow	Move to next input line.
Ctrl-N	Move to next input line (alternative).
UpArrow	Move to previous input line.
Ctrl-P	Move to previous input line (alternative).
Ctrl-R	Incremental search backward for text.
Ctrl-S	Incremental search forward for text.
Ctrl-T	Transpose the character under the cursor with the character before the cursor.
Ctrl-Y	Insert previously erased with **Ctrl-K** or **Alt-D**.
Ctrl-_	Undo the last command.
Alt-C	Capitalize the following word.
Alt-D	Delete the following word.
Alt-F	Move forward one word.
Alt-L	Convert the following word to lower case.

Key	Function
Alt-T	Transpose the word before the cursor with the one after it.
Alt-U	Convert the following word to upper case.
Ctrl-X Ctrl-S	Save file (*Emacs* only).
Ctrl-X Ctrl-C	Exit the *Emacs* editor.

You'll note a number of alternatives to the cursor keys. There are two reasons for them: firstly, the shell and *Emacs* must work on systems without arrow keys on the keyboard. The second reason is not immediately obvious: if you're a touch-typer, it's easier to type **Ctrl-P** than take your hands away from the main keyboard and look for the arrow key. The arrows are good for beginners, but if you get used to the control keys, you'll never miss the arrow keys.

File name completion

As we have seen, UNIX file names can be much longer than traditional Microsoft names, and it becomes a problem to type them correctly. To address this problem, newer shells provide *file name completion*. In *Emacs* mode, you typically type in part of the name, then press the **Tab** key. The shell checks which file names begin with the characters you typed. If there is only one, it puts in the missing characters for you. If there are none, it beeps (rings the ''terminal bell''). If there are more than one, it puts in as many letters as are common to all the file names, and then beeps. For example, if I have a directory *docco* in my home directory, I might enter:

```
=== grog@freebie (/dev/ttyp4) ~ 14 -> cd docco/
=== grog@freebie (/dev/ttyp4) ~/docco 15 -> ls
freebsd.faq  freebsd.fbc  freeware
=== grog@freebie (/dev/ttyp4) ~/docco 16 -> emacs freebeepbsd.fbeepaq
```

Remember that my input is in **constant width bold** font, and the shell's output is in constant width font. On the first line, I entered the characters `cd doc` followed by a **Tab** character, and the shell completed with the text `co/`. On the last line, I entered the characters `emacs f` and a **Tab**. In this case, the shell determined that there was more than one file name that started like this, so it added the letters `ree` and rang the bell. I entered the letter `b` and pressed **Tab** again, and the shell added the letters `sd.f` and beeped again. Finally, I added the letters `aq` to complete the file name *freebsd.faq*.

Command line completion in *vi* mode is similar: instead of pressing **Tab**, you press **ESC** twice.

Shell startup files

As we saw above, there are a lot of ways to customize your shell. It would be inconvenient to have to set them every time, so all shells provide a means to set them automatically when you log in. Nearly every shell has its own startup file. Table 7-6 gives an overview.

Table 7-6: Shell startup files

Shell	startup file
bash	*.profile*, then *.bashrc*
csh	*.login* on login, always *.cshrc*
sh	*.profile*
tcsh	*.login* on login, always *.tcshc*, *.cshrc* if *.tcshrc* not found

These files are shell scripts—in other words, straight shell commands. Figure 7-3 shows a typical *.bashrc* file to set the environment variables we discussed.

```
umask 022
export BLOCKSIZE=1024   # for df
export CVSROOT=/src/ncvs
export EDITOR=/opt/bin/emacs
export MANPATH=/usr/share/man:/usr/local/man
export MOZILLA_HOME=/usr/local/netscape
export PAGER=less
export PATH=/usr/bin:/usr/local/bin:/usr/sbin:/bin:/sbin:/usr/X11R6/bin
PS1="=== \u@\h (`tty`) \w \# -> "
PS2="\u@\h \w \! ++ "
export SHELL=/usr/local/bin/bash
export TAPE=/dev/nsa0   # note non-rewinding as standard
if [ "$TERM" = "" ]; then
  export TERM=xterm
fi
if [ "$DISPLAY" = "" ]; then
  export DISPLAY=:0
fi
/usr/games/fortune              # print a fortune cookie
```

Figure 7-3: Minimal .bashrc file

It would be tedious for every user to put settings in their private initialization files, so the shells also read a system-wide default file. For the Bourne shell family, it is */etc/profile*, while the C shell family has three files: */etc/csh.login* to be executed on login, */etc/csh.cshrc* to be executed when a new shell is started after you log in, and */etc/csh.logout* to be executed when you stop a shell. The start files are executed before the corresponding individual files.

In addition, login classes (page 570) offer another method of setting environment variables at a global level.

Changing your shell

The FreeBSD installation gives `root` a C shell, *csh*. This is the traditional BSD shell, but it has a number of disadvantages: command line editing is very primitive, and the script language is significantly different from that of the Bourne shell, which is the *de facto* standard for shell scripts: if you stay with the C shell, you may still need to understand the Bourne shell. The latest version of the Bourne shell *sh* also includes some command line editing. See page 133 for details of how to enable it.

If you want to stay with a *csh*-like shell, you can get better command line editing with *tcsh*, which is also in the base system. You can get both better command line editing and Bourne shell syntax with *bash*, in the Ports Collection.

If you have `root` access, you can use *vipw* to change your shell, but there's a more general way: use *chsh* (*Change Shell*). Simply run the program. It starts your favourite editor (as defined by the `EDITOR` environment variable). Here's an example before:

```
#Changing user database information for velte.
Shell: /bin/csh
Full Name: Jack Velte
Location:
Office Phone:
Home Phone:
```

You can change anything after the colons. For example, you might change this to:

```
#Changing user database information for velte.
Shell: /usr/local/bin/bash
Full Name: Jack Velte
Location: On the road
Office Phone: +1-408-555-1999
Home Phone:
```

chsh checks and updates the password files when you save the modifications and exit the editor. The next time you log in, you get the new shell. *chsh* tries to ensure you don't make any mistakes—for example, it won't let you enter the name of a shell that isn't mentioned in the file */etc/shells*—but it's a *very* good idea to check the shell before logging out. You can try this with *su*, which you normally use to become super user:

```
bumble# su velte
Password:
su-2.00$                                    note the new prompt
```

You might hear objections to using *bash* as a root shell. The argument goes something like this: *bash* is installed in */usr/local/bin*, so it's not available if you boot into single-user mode, where only the root file system is available. Even if you copy it to, say, */bin*, you can't run it in single-user mode because it needs libraries in */usr/lib*.

In fact, this isn't a problem. If you install the system the way I recommend in Chapter 5, */usr is* on the root file system. Even if it isn't, though, you don't have to use *bash* in single-user mode. When you boot to single-user mode, you get a prompt asking you which shell to start, and suggesting */bin/sh*.

Differences from Microsoft

If you're coming from a Microsoft background, there are a few gotchas that you might trip over.

Slashes: backward and forward

/ (slash) and \ (backslash) are confusing. As we've seen, UNIX uses / to delimit directories. The backslash \ is called an *escape character*. It has several purposes:

- You can put it in front of another special character to say ''don't interpret this character in any special way.'' We've seen that the shell interprets a space character as the end of a parameter. In the previous example we changed `Mail rejected` to `"Mail rejected"` to stop the shell from interpreting it. We could also have written: `Mail\ rejected`.

  A more common use for this *quoting* is to tell the shell to ignore the end of a line. If a command line in a shell script gets too long, you might like to split it up into several lines; but the shell sees the end of a line as a go-ahead to perform the command. Stop it from doing so by putting a backslash *immediately* before the end of the line:

  ```
  $ grep \
    "Mail rejected" \
     /var/log/maillog
  ```

  Don't put any spaces between the \ and the end of the line; otherwise the shell will interpret the first space as a parameter by itself, and then it will interpret the end of line as the end of the command.

- In the C programming language, the backslash is used to represent several *control characters*. For example, `\n` means ''new line.'' This usage appears in many other places as well.

- Using \ as an escape character causes problems: how do we put a \ character on a line? The answer: quote it. Write \\ when you mean \. This causes particular problems when interfacing with Microsoft: if you give a Microsoft path name to a shell, it needs the doubled backslashes: *C:\\WINDOWS*.

Tab characters

We've seen that the shell treats ''white space,'' either spaces or tab characters, as the same. Unfortunately, some other programs do not. *make*, *sendmail* and *syslogd* make a distinction between the two kinds of characters, and they all require tabs (not spaces) in certain places. This is a *real* nuisance, because hardly any editor makes a distinction between them.

Carriage control characters

In the olden days, the standard computer terminal was a Teletype, a kind of computer-controlled electric typewriter. When the carriage, which contained the print head, got to the end of a line, it required two mechanical operations to move to the beginning of the next line: the *carriage return* control character told it to move the carriage back to the beginning of the line, and the *line feed* character told it turn the platen to the next line.

Generations of computer systems emulated this behaviour by putting both characters at the end of each text line. This makes it more difficult to recognize the end of line, it uses up more storage space, and normally it doesn't buy you much. The implementors of UNIX decided instead to use a single character, which it calls the *new line* character. For some reason, they chose the line feed to represent new line, though the character generated by **Enter** is a carriage return. As we saw above, the C programming language represents it as `\n`.

This causes problems transferring data between FreeBSD and Microsoft, and also when printing to printers that still expect both characters. We'll look at the file transfer issues on page 260 and the printer issues on page 267.

The Emacs editor

Apart from the shell, your second most important tool is the *editor*, a program that creates and changes texts. Another divergence of concept between UNIX and Microsoft environments is that UNIX gives you a choice of editors in just about anything you do. Microsoft products frequently try to redefine the whole environment, so if you change mailers, you may also have to change the editor you use to write mail. This has a profound effect on the way you work. In particular, the Microsoft way makes it uninteresting to write a really good editor, because you can't use it all the time.

The standard BSD editor is *vi*, about which people speak with a mixture of admiration, awe and horror. *vi* is one of the oldest parts of BSD. It is a very powerful editor, but nobody would say that it is easy to learn. There are two reasons to use *vi*:

1. If you're already an experienced *vi* hacker, you probably won't want to change.
2. If you do a lot of work on different UNIX systems, you can rely on *vi* being there. It's about the only one on which you can rely.

If, on the other hand, you don't know *vi*, and you only work on systems whose software you can control, you probably shouldn't use *vi*. *Emacs* is much easier to learn, and it is more powerful than *vi*.

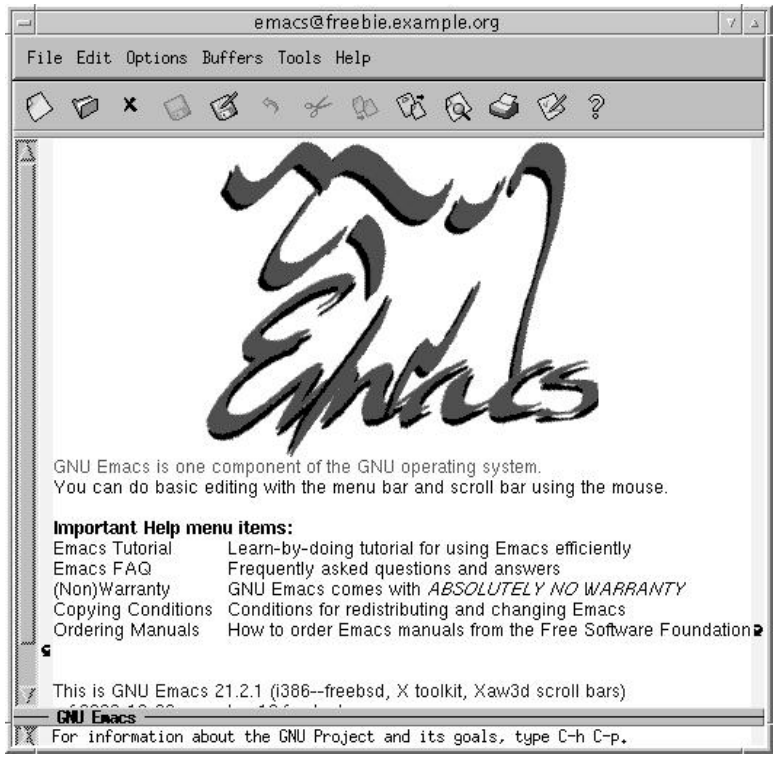


Figure 7-4: Emacs main menu

When running under X, *Emacs* displays its own window (*vi* uses an *xterm* under these circumstances). As a result, if you start *Emacs* from an *xterm*, you should use the & character to start it in the background:

```
$ emacs &
```

Figure 7-4 shows the resulting display. As you can see, the first thing that *Emacs* offers you is a tutorial. You should take it. You'll also notice the menu bars at the top. Although they look primitive compared to graphics toolbars, they offer all the functionality of graphics-oriented menus. In addition, they will tell you the keystrokes that you can use to invoke the same functions. Figure 7-5 gives an example of the *Files* menu.

There is a lot of documentation for *Emacs*, much of it on line. The complete *Emacs* handbook is available via the *info* mode of *Emacs*, which is described in the tutorial. If that's not enough, read *Learning GNU Emacs*, by Debra Cameron, Bill Rosenblatt and Eric Raymond.

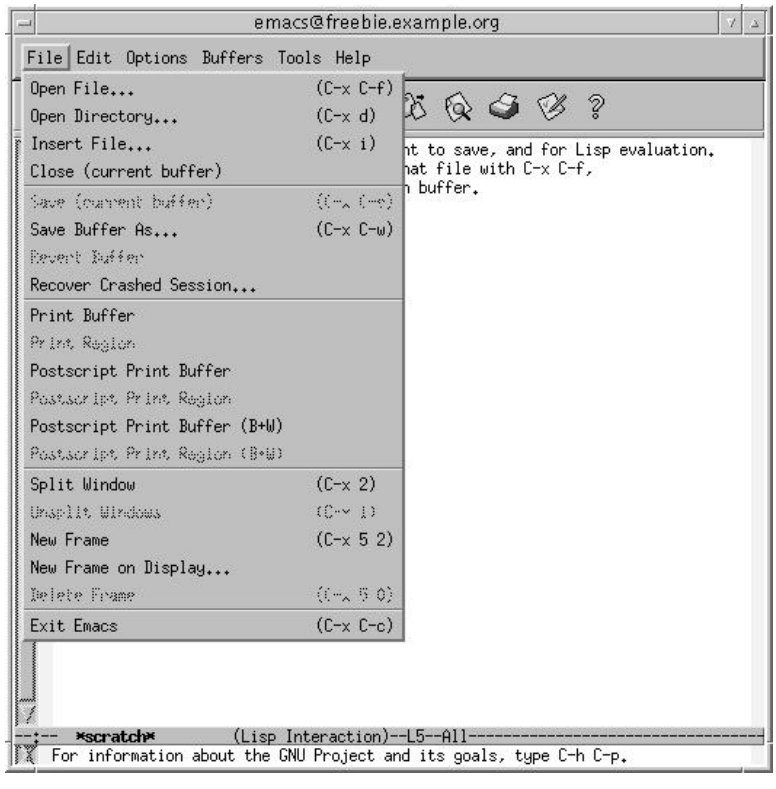


Figure 7-5: Emacs files menu

Stopping the system

To stop X, press the key combination **Ctrl-Alt-Backspace**, which is deliberately chosen to resemble the key combination **Ctrl-Alt-Delete** used to reboot the machine. **Ctrl-Alt-Backspace** stops X and returns you to the virtual terminal in which you started it. If you run from *xdm*, it redisplays a login screen.

To stop the system, use the *shutdown* program. To do so, you need to be a member of group `operator`.

By default, KDE uses the *halt* program. Only `root` can use this program, so you should reconfigure KDE to use *shutdown*. After this, you can shut down from KDE with the keystroke combination **Ctrl-Alt-PageDown**.