4th Edition

Covers Version 5.0

The Complete FreeBSD

Documentation from the Source

Greg Lehey

O'REILLY®
COMMUNITY PRESS

8 Taking control

In Chapter 7 we saw the basics of working with FreeBSD. In this part of the book, we'll look at some more system-specific issues. This chapter discusses the following topics:

- UNIX is a multi-user operating system. We've already skimmed over creating user accounts, but on page 144 we'll look at it in more detail.
- Not all users are created equal. In particular, the system administration login `root` has power over all other users. We'll look at `root` on page 146.
- UNIX implements multi-tasking via a mechanism called *processes*. We'll look at them on page 148.
- Timekeeping is extremely important in a networking system. If your system has the wrong time, it can cause all sorts of strange effects. On page 155 we'll look at how to ensure that your system is running the correct time.
- A number of events are of interest in keeping a machine running smoothly. The system can help by keeping track of what happens. One mechanism for this is *log files*, files that contain information about what has happened on the machine. We'll look at them on page 157.
- On page 159, we'll look at how FreeBSD handles systems with more than one processor. This is also called *Symmetrical Multi-Processor* or *SMP* support.
- Nearly every modern laptop has as special bus for plugin cards. It used to be called *PCMCIA*, an acronym for the rather unlikely name *Personal Computer Memory Card International Association*. Nowadays it's called *PC Card*. It was later upgraded to a 32 bit bus called *CardBus*. We'll look at how FreeBSD supports PC Card and CardBus on page 159.

- Starting on page 162, we'll look at FreeBSD's support for emulating other operating systems.
- Other aspects of FreeBSD are so extensive that we'll dedicate separate chapters to them. We'll look at them in Chapters 9 to 15.
- Starting and stopping the system is straightforward, but there are a surprising number of options. Many of them are related to networking, so Chapter 29 is located after the networking section.

Users and groups

We've already looked at users in Chapter 7. In this chapter, we'll take a deeper look.

In traditional UNIX, information about users was kept in the file */etc/passwd*. As the name suggests, it included the passwords, which were stored in encrypted form. Any user could read this file, but the encryption was strong enough that it wasn't practical to decrypt the passwords. Nowadays processors are much faster, and it's too easy to crack a password. As a result, FreeBSD keeps the real information in a file called */etc/master.passwd*, and for performance reasons it also makes it available in database form in */etc/pwd.db* and */etc/spwd.db*. None of these file are user-readable. */etc/passwd* remains for compatibility reasons: some third-party programs access it directly to get information about the environment in which they are running.

Choosing a user name

So what user name do you choose? User names are usually related to your real name and can be up to eight characters long. Like file names, they're case-sensitive. By convention, they are in all lower case, even when they represent real names. Typical ways to form a user name are:

- First name. In my personal case, this would be `greg`.
- Last name (`lehey`).
- First name and initial of last name (`gregl`).
- Initial of first name, and last name (`glehey`).
- Initials (`gpl`).
- Nickname (for example, `grog`).

I choose the last possibility, as we will see in the following discussion.

Adding users

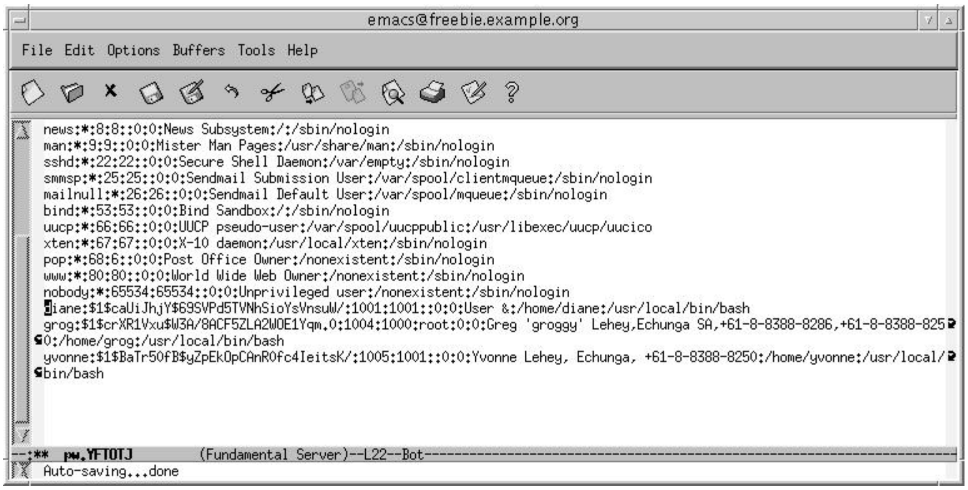
We’ve already seen how to use *sysinstall* to create a user. It’s not the only way. There are at least two other methods. One is the program *adduser*:

```
# **adduser**
Use option ``-verbose'' if you want see more warnings & questions
or try to repair bugs.

Enter username [a-z0-9]: **yana**
Enter full name []: **Yana Lehey**
Enter shell bash csh date no sh [bash]:          *accept the default*
Uid [1000]:                                      *accept the default*
Enter login class: default []:                   *accept the default*
Login group yana [yana]: **home**
Login group is ``home''. Invite yana into other groups: **no**
[no]: **wheel**                                      *to be able to use su*
Enter password []:                               *no echo*
Enter password again []:                         *no echo*

Name:     yana
Password: ****
Fullname: Yana Lehey
Uid:      1000
Gid:      1001 (home)
Class:
Groups:   home wheel
HOME:     /home/yana
Shell:    /bin/bash
OK? (y/n) [y]:                                   *accept the default*
Added user ``yana''
Add another user? (y/n) [y]: **n**
```

An alternative way of adding or removing users is with the *vipw* program. This is a more typical UNIX-hackish approach: *vipw* starts your favourite editor and allows you to edit the contents of the file */etc/master.passwd*. After you have finished, it checks the contents and rebuilds the password database. Figure 8-1 shows an example.


emacs@freebie.example.org
File Edit Options Buffers Tools Help

```
news:*:8:8::0:0:News Subsystem:/:/sbin/nologin
man:*:9:9::0:0:Mister Man Pages:/usr/share/man:/sbin/nologin
sshd:*:22:22::0:0:Secure Shell Daemon:/var/empty:/sbin/nologin
smmsp:*:25:25::0:0:Sendmail Submission User:/var/spool/clientmqueue:/sbin/nologin
mailnull:*:26:26::0:0:Sendmail Default User:/var/spool/mqueue:/sbin/nologin
bind:*:53:53::0:0:Bind Sandbox:/:/sbin/nologin
uucp:*:66:66::0:0:UUCP pseudo-user:/var/spool/uucppublic:/usr/libexec/uucp/uucico
xten:*:67:67::0:0:X-10 daemon:/usr/local/xten:/sbin/nologin
pop:*:68:6::0:0:Post Office Owner:/nonexistent:/sbin/nologin
www:*:80:80::0:0:World Wide Web Owner:/nonexistent:/sbin/nologin
nobody:*:65534:65534::0:0:Unprivileged user:/nonexistent:/sbin/nologin
diane:$1$caUiJhjY$69SVPd5TVNhSioYsVnsuW/:1001:1001::0:0:User &:/home/diane:/usr/local/bin/bash
grog:$1$crXR1Vxu$W3A/8ACF5ZLA2WOE1Yqm.0:1004:1000:root:0:0:Greg 'groggy' Lehey,Echunga SA,+61-8-8388-8286,+61-8-8388-825
0:/home/grog:/usr/local/bin/bash
yvonne:$1$BaTr50fB$yZpEkOpCAnR0fc4IeitsK/:1005:1001::0:0:Yvonne Lehey, Echunga, +61-8-8388-8250:/home/yvonne:/usr/local/
bin/bash

--:** pw.YFTOTJ      (Fundamental Server)--L22--Bot------------------------------
Auto-saving...done
```

Figure 8-1: vipw display

You might be wondering why would you ever want to do things this way, and you might find it funny that most experienced UNIX administrators prefer it. The reason is that you get more of an overview than with a peephole approach that graphical environments give you, but of course you need to understand the format better. It's less confusing once you know that each line represents a single user, that the lines are divided into *fields* (which may be empty), and that each field is separated from the next by a colon (`:`). Table 8-1 describes the fields you see on the line on which the cursor is positioned. You can read more about the format of */etc/master.passwd* in the man page *passwd(5)*.

Table 8-1: /etc/master.passwd format

Field	Meaning
`yvonne`	User name.
`(gibberish)`	Encrypted password. When adding a new user, leave this field empty and add it later with the *passwd* program.
`1005`	User number.
`1001`	Group number.
`(empty)`	Login class, which describes a number of parameters for the user. We'll look at it in Chapter 29, on page 570. This field is not included in */etc/passwd*.
`0`	Password change time. If non-0, it is the time in seconds after which the password must be changed. This field is not included in */etc/passwd*.
`0`	Account expiration time. If non-0, it is the time in seconds after which the user expires. This field is not included in */etc/passwd*.
`Yvonne Lehey`	The so-called *gecos* field, which describes the user. This field is used by a number of programs, in particular mail readers, to extract the real name of the user.
`/home/yvonne`	The name of the home directory.
`/bin/bash`	The shell to be started when the user logs in.

The super user

FreeBSD has a number of privileged users for various administration functions. Some are just present to be the owners of particular files, while others, such as `daemon` and `uucp`, exist to run particular programs. One user stands above all others, however: `root` may do just about anything. The kernel gives `root` special privileges, and you need to become `root` to perform a number of functions, including adding other users. Make sure `root` has a password if there is any chance that other people can access your system (this is a must if you have any kind of dialup access). Apart from that, `root` is a user like any other, but to quote the man page *su(1)*:

By default (unless the prompt is reset by a startup file) the super user prompt is set to # *to remind one of its awesome power.*

Becoming super user

Frequently when you're logged in normally, you want to do something that requires you to be `root`. You can log out and log in again as `root`, of course, but there's an easier way:

```
$ su                                            become super user
Password:                                       as usual, it doesn't echo
#                                               root prompt
```

To use *su*, you must be a member of the group `wheel`. Normally you do this when you add the user, but otherwise just put the name of the user at the end of the line in */etc/group*:

```
wheel:*:0:root,grog                             add the text in bold face
```

BSD treats *su* somewhat differently from System V. First, you need to be a member of the group `wheel`, and secondly BSD gives you more of the super user environment than System V. See the man page for further information.

Having a single `root` password is a security risk on a system where multiple people know the password. If one of them leaves the project, you need to change the password. An alternative is the *sudo* port (*/usr/ports/security/sudo*). It provides fine-grained access to `root` privileges, all based on the user's own password. Nobody needs to know the `root` password. If a user leaves, you just remove his account, and that cancels his access.

Adding or changing passwords

If your system has any connection with the outside world, it's a good idea to change your password from time to time. Do this with the *passwd* program. The input doesn't look very interesting:

```
$ passwd
Changing local password for yana.
Old password:                                   doesn't echo
New password:                                   doesn't echo
Retype new password:                            doesn't echo
passwd: rebuilding the database...
passwd: done
```

You have to enter the old password to make sure that some passer-by doesn't change it for you while you're away from your monitor, and you have to enter the new password twice to make sure that you don't mistype and lock yourself out of your account. If this does happen anyway, you can log in as `root` and change the password: `root` doesn't have to enter the old password, and it can change anybody's password. For example:

```
# passwd yana
Changing local password for yana.
New password:                            doesn't echo
Retype new password:                     doesn't echo
passwd: rebuilding the database...
passwd: done
```

In this case, you specify the name of the user for whom you change the password.

If you are changing the `root` password, be careful: it's easy enough to lock yourself out of the system if you mess things up, which could happen if, for example, you mistyped the password twice in the same way (don't laugh, it happens). If you're running X, open another window and use *su* to become `root`. If you're running in character mode, select another virtual terminal and log in as `root` there. Only when you're sure you can still access `root` should you log out.

If you *do* manage to lose the `root` password, all may not be lost. Reboot the machine to single-user mode (see page 540), and enter:

```
# mount -u /                             mount root file system read/write
# mount /usr                             mount /usr file system (if separate)
# passwd root                            change the password for root
Enter new password:
Enter password again:
# ^D                                     enter ctrl-D to continue with startup
```

If you have a separate */usr* file system (the normal case), you need to mount it as well, since the *passwd* program is in the directory */usr/bin*. Note that you should explicitly state the name `root`: in single-user mode, the system doesn't have the concept of user IDs.

Processes

As we have seen, UNIX is a multi-user, multi-tasking operating system. In particular, you can run a specific program more than once. We use the term *process* to refer to a particular instance of a running program. Each process is given a *process ID*, more frequently referred to as *PID*, a number between 0 and 99999 that uniquely identifies it. There are many things that you might like to know about the processes that are currently running, such as:

- How many processes are running?
- Who is running the processes?
- Why is the system so slow?
- Which process is blocking my access to the modem?

Your primary tool for investigating process behaviour is the *ps* (*process status*) command. It has a large number of command options, and it can tell you a whole lot of things that you will only understand when you have investigated how the kernel works, but it can be very useful for a number of things. Here are some typical uses:

What processes do I have running?

After starting a large number of processes in a number of windows under X, you probably can't remember what is still running. Maybe processes that you thought had stopped are still running. To display a brief summary of the processes you have running, use the *ps* command with no options:

```
$ ps
  PID  TT  STAT      TIME COMMAND
  187  p0  Is+    0:01.02 -bash (bash)
  188  p1  Ss     0:00.62 -bash (bash)
  453  p1  R+     0:00.03 ps
```

This display shows the following information:

- The PID of the process.
- `TT` is short for *teletype*, and shows the last few letters of the name of the *controlling terminal*, the terminal on which the process is running. In this example, the terminals are */dev/ttyp0* and */dev/ttyp1*.
- `STAT` shows the current process status. It's involved and requires a certain amount of understanding of how the kernel runs to interpret it—see the man page for *ps* for more details.
- `TIME` is the CPU time that the process has used in minutes, seconds and hundredths of a second. Note that many other UNIX systems, particularly System V, only show this field to the nearest second.
- `COMMAND` is normally the command you entered, but don't rely on this. In the next section, you'll see that *sendmail* has changed its `COMMAND` field to tell you what it is doing. You'll notice that the command on the last line is the *ps* that performs the listing. Due to some complicated timing issue in the kernel, this process may or may not appear in the listing.

What processes are running?

There are many more processes in the system than the list above shows. To show them all, use the `a` option to *ps*. To show daemons as well (see the next section for a definition of *daemon*), use the `x` option. To show much more detail, use the `u` or `l` options. For example:

```
$ ps waux
USER      PID %CPU %MEM   VSZ  RSS  TT  STAT STARTED      TIME COMMAND
root       12 95.7  0.0     0   12  ??  RL    1Jan70 1406:43.85  (idle: cpu0)
root       11 95.1  0.0     0   12  ??  RL    1Jan70 1406:44.64  (idle: cpu1)
root        1  0.0  0.0   708   84  ??  ILs   1Jan70   0:09.10 /sbin/init --
root       12  0.0  0.0     0   12  ??  WL    1Jan70  15:04.95  (swi1: net)
root       13  0.0  0.0     0   12  ??  WL    1Jan70  21:30.29  (swi6: tty:sio clock)
root       15  0.0  0.0     0   12  ??  DL    1Jan70   2:17.27  (random)
root       18  0.0  0.0     0   12  ??  WL    1Jan70   0:00.00  (swi3: cambio)
root       20  0.0  0.0     0   12  ??  WL    1Jan70   0:00.00  (irq11: ahc0 uhci0++)
root       21  0.0  0.0     0   12  ??  WL    1Jan70  39:00.32  (irq5: rl0)
root       22  0.0  0.0     0   12  ??  WL    1Jan70   7:12.92  (irq14: ata0)
root       23  0.0  0.0     0   12  ??  WL    1Jan70   0:47.99  (irq15: ata1)
root       24  0.0  0.0     0   12  ??  DL    1Jan70   0:00.08  (usb0)
root       25  0.0  0.0     0   12  ??  DL    1Jan70   0:00.00  (usbtask)
root       26  0.0  0.0     0   12  ??  DL    1Jan70   0:00.07  (usb1)
root       27  0.0  0.0     0   12  ??  DL    1Jan70   0:00.08  (usb2)
root      340  0.0  0.1  1124  280  ??  S    18Dec02  16:41.11 nfsd: server (nfsd)
root      375  0.0  0.0  1192   12  ??  Ss   18Dec02   0:01.70 /usr/sbin/lpd
daemon    408  0.0  0.0  1136  152  ??  Ss   18Dec02   0:11.41 /usr/sbin/rwhod
root      420  0.0  0.1  2648  308  ??  Ss   18Dec02   0:04.20 /usr/sbin/sshd
root      491  0.0  0.1  2432  368  ??  Ss   18Dec02   0:38.61 /usr/local/sbin/httpd
root      551  0.0  0.0  1336   12  ??  Ss   18Dec02   0:02.71 /usr/sbin/inetd -wW
root      562  0.0  0.0  1252  216  ??  Is   18Dec02   0:15.50 /usr/sbin/cron
root      572  0.0  0.0  1180    8  v2  IWs+ -         0:00.00 /usr/libexec/getty Pc
www       582  0.0  0.0  2432    8  ??  IW   -         0:00.00 /usr/local/sbin/httpd
grog      608  0.0  0.1  1316  720  v0  I    18Dec02   0:00.04 -bash (bash)
root     2600  0.0  0.0  1180    8  v1  IWs+ -         0:00.00 /usr/libexec/getty Pc
root    33069  0.0  0.3  5352 1716  ??  Ss   29Dec02   0:01.30 xterm -name xterm
grog    33081  0.0  0.1  1328  752  p8  Is+  29Dec02   0:00.09 /usr/local/bin/bash
```

This list is just an excerpt. Even on a freshly booted system, the real list of processes will be much larger, about 50 processes.

We've seen a number of these fields already. The others are:

- `USER` is the *real user ID* of the process, the user ID of the person who started it.
- `%CPU` is an approximate count of the proportion of CPU time that the process has been using in the last few seconds. This is the column to examine if things suddenly get slow.
- `%MEM` is an approximate indication of the amount of physical memory that the process is using.
- `VSZ` (*virtual size*) is the amount of virtual memory that the process is using, measured in kilobytes.
- `RSS` (*resident segment size*) is the amount of physical memory currently in use, measured in kilobytes.
- `STARTED` is the time or date when the process was started.

In addition, a surprising number of processes don't have a controlling terminal. They are *daemons*, and we'll look at them in the next section.

Daemons

A significant part of the work in a FreeBSD system is performed by *daemons*. A daemon is not just the BSD mascot described on page 20—it's also a process that goes around in the background and does routine work such as sending mail (*sendmail*), handling incoming Internet connections (*inetd*), or starting jobs at particular times (*cron*).

> To quote the *Oxford English Dictionary*: **Demon** Also **dæmon**. ME [In form, and in sense I, a. L. *dæmon* (med. L. *demon*)...] 1a. In ancient Greek mythology (= *δαίμων*): A supernatural being of a nature intermediate between that of gods and men, an inferior divinity, spirit, genius (including the souls of deceased persons, *esp* deified heros). Often written *dæmon* for distinction.

You can recognize daemons in a `ps waux` listing by the fact that they don't have a controlling terminal—instead you see the characters `??`. Each daemon has a man page that describes what it does.

Normally, daemons are started when the system is booted and run until the system is stopped. If you stop one by accident, you can usually restart them. One exception is *init*, which is responsible for starting other processes. If you kill it, you effectively kill the system. Unlike traditional UNIX systems, FreeBSD does not allow *init* to be killed.

cron

One of the more useful daemons is *cron*, named after Father Time. *cron* performs functions at specific times. For example, the system runs the script */etc/periodic/daily* every day at 2:00 am, the script */etc/periodic/weekly* every Saturday at 3:30 am, and the script */etc/periodic/monthly* on the first day of every month at 5:30 am.

To tell *cron* to perform a function at a particular time, you need a file called a *crontab*. The system keeps the real *crontab* where you can't get at it, but you can keep a copy. It's a good idea to call it *crontab* as well.

Let's look at the format of the default system *crontab*, located in */etc/crontab*:

```
# /etc/crontab - root's crontab for FreeBSD
#
# $Id: crontab,v 1.10 1995/05/27 01:55:21 ache Exp $
# From: Id: crontab,v 1.6 1993/05/31 02:03:57 cgd Exp
#
SHELL=/bin/sh
PATH=/etc:/bin:/sbin:/usr/bin:/usr/sbin
HOME=/var/log
#
#minute hour    mday    month   wday    who     command
#
*/5     *       *       *       *       root    /usr/libexec/atrun
#
# rotate log files every hour, if necessary
#0      *       *       *       *       root    /usr/bin/newsyslog
#
# do daily/weekly/monthly maintenance
0       2       *       *       *       root    /etc/daily 2>&1
30      3       *       *       6       root    /etc/weekly 2>&1
30      5       1       *       *       root    /etc/monthly 2>&1
#
# time zone change adjustment for wall cmos clock,
# See adjkerntz(8) for details.
1,31    0-4     *       *       *       root    /sbin/adjkerntz -a
```

As usual, lines starting with # are comments. The others have seven fields. The first five fields specify the minute, the hour, the day of the month, the month, and the day of the week on which an action should be performed. The character ***** means "every." Thus, `0 2 * * *` (for */etc/daily*) means "0 minutes, 2 o'clock (on the 24 hour clock), every day of the month, every month, every weekday."

Field number six is special: it only exists in */etc/crontab*, not in private *crontab*s. It specifies the user for whom the operation should be performed. When you write your own *crontab* file, don't use this field.

The remaining fields define the operation to be performed. *cron* doesn't read your shell initialization files. In particular, this can mean that it won't find programs you expect it to find. It's a good idea to put in explicit `PATH` definitions, or specify an absolute pathname for the program, as is done in this example. *cron* mails the output to you, so you should check `root`'s mail from time to time.

To install or list a *crontab*, use the *crontab* program:

```
$ crontab crontab                          install a crontab
$ crontab -l                               list the contents of an installed crontab
# DO NOT EDIT THIS FILE - edit the master and reinstall.
# (crontab installed on Wed Jan  1 15:15:10 1997)
# (Cron version -- $Id: crontab.c,v 1.7 1996/12/17 00:55:12 pst Exp $)
0 0 * * * /home/grog/Scripts/rotate-log
```

Processes in FreeBSD Release 5

Some of the processes in the example above are specific to FreeBSD Release 5:

- FreeBSD Release 5 has an *idle process* to use up the excess processor time and perform certain activities needed when no process is active. This example machine has two processors, so there are two of them:

```
root   12 95.7  0.0  0   12  ??  RL    1Jan70 1406:43.85  (idle: cpu0)
root   11 95.1  0.0  0   12  ??  RL    1Jan70 1406:44.64  (idle: cpu1)
```

- A number of the processes have names starting with `irq` or `swi`:

```
root  12  0.0  0.0    0   12  ??  WL    1Jan70  15:04.95  (swi1: net)
root  13  0.0  0.0    0   12  ??  WL    1Jan70  21:30.29  (swi6: tty:s
root  18  0.0  0.0    0   12  ??  WL    1Jan70   0:00.00  (swi3: cambi
root  20  0.0  0.0    0   12  ??  WL    1Jan70   0:00.00  (irq11: ahc0
root  21  0.0  0.0    0   12  ??  WL    1Jan70  39:00.32  (irq5: rl0)
root  22  0.0  0.0    0   12  ??  WL    1Jan70   7:12.92  (irq14: ata0)
root  23  0.0  0.0    0   12  ??  WL    1Jan70   0:47.99  (irq15: ata1)
```

  These processes handle hardware interrupts (`irq`) or software interrupts (`swi`). The text which follows gives an idea of which devices or software services they support.

top

Another tool for investigating system performance is *top*, which shows a number of performance criteria, including the status of the processes are using the most resources. Start it with the number of processes you want displayed. Figure 8-2 gives an example.

```
$ top -S 10
last pid:  3992; load averages:  0.59, 0.17, 0.06 up 0+23:54:49  17:25:13
87 processes:  3 running, 73 sleeping, 8 waiting, 3 lock
CPU states: 10.2% user, 0.0% nice, 18.8% system, 1.7% interrupt, 69.4% idle
Mem: 43M Active, 36M Inact, 31M Wired, 7460K Cache, 22M Buf, 2996K Free
Swap: 512M Total, 512M Free

  PID USER   PRI NICE   SIZE    RES STATE  C   TIME   WCPU    CPU COMMAND
   12 root   -16    0     0K    12K RUN    0  23.7H 55.32% 55.32% idle: cpu0
   11 root   -16    0     0K    12K CPU1   1  23.7H 54.49% 54.49% idle: cpu1
 2854 grog    97    0  4940K  3932K *Giant 1   0:04  3.88%  3.86% xterm
   20 root   -64 -183     0K    12K WAIT   1   0:08  0.83%  0.83% irq14: ata0
 2925 root    96    0   712K   608K select 1   0:01  0.15%  0.15% make
 3193 grog    96    0  2220K  1304K CPU0   0   0:01  0.15%  0.15% top
 3783 root    96    0   520K   416K select 1   0:00  0.10%  0.05% make
  167 root    96    0 13876K  2112K select 0   1:02  0.00%  0.00% xcpustate
   25 root   -68 -187     0K    12K WAIT   0   0:28  0.00%  0.00% irq9: xl0
  110 root    96    0  1528K   956K select 1   0:26  0.00%  0.00% ntpd
```

Figure 8-2: top display

By default, the display is updated every two seconds and contains a lot of information about the system state:

- The first line gives information about the last PID allocated (you can use this to follow the number of processes being created) and the *load average*, which gives information about how many processes are waiting to be scheduled.
- The next line gives an overview of process statistics, and in what state they are. A process waits for external events to complete; it waits on a lock if some other process has a kernel resource which it wants.
- The third line shows the percentage of time used in user mode, in system (kernel) mode and by interrupts.
- The fourth line shows memory usage.
- The fifth line shows swap statistics. When swapping activity occurs, it also appears on this line.
- The remaining lines show the ten most active processes (because the parameter 10 was specified on the command line). The `-S` option tells *top* to include system processes, such as the idle and the interrupt processes. The state can be:
    - `RUN`, when the process is waiting for a processor to run on.
    - `CPU0` or `CPU1`, when the process is actively executing.
    - `*`*`lock`*, where *`lock`* is the name of a kernel lock. In this example, the *xterm* is waiting on the lock `Giant`.
    - A *wait string*, which indicates an event on which the process is waiting.

See the man page *top(1)* for more details.

Stopping processes

Sometimes you may find that you want to stop a currently running process. There are a number of ways to do this, but the easiest are:

- If the process is running on a terminal, and it's accepting input, hitting the EOF key (usually **Ctrl-D**) will often do it.
- If EOF doesn't do it, try the INTR key (usually **Ctrl-C**).
- If the process is ignoring INTR, or if it is not associated with a terminal, use the *kill* command. For example, to find who is using all the CPU time, use *ps* and look at the `%CPU` field:

  ```
  # ps waux | grep cron
  root   105  97.3  1.1  236  340  ??  Is    9:11AM 137:14.29 cron
  ```

  Here, *cron* is using 97% of the CPU time, and has accumulated over 2 hours of CPU time since this morning. It's obviously sick, and we should put it out of its misery. To stop it, enter:

  ```
  # kill 105
  ```

  This command sends a signal called `SIGTERM` (terminate) to the process. This signal gives the process time to tidy up before exiting, so you should always try to use it first. The 105 is *cron*'s PID, which we got from the *ps* command.

  If the process doesn't go away within a few seconds, it's probably ignoring `SIGTERM`. In this case, you can use the ultimate weapon:

  ```
  # kill -9 105
  ```

  The `-9` is the number of `SIGKILL`, a signal that cannot be caught or ignored. You can find a list of the signals and their numeric values in */usr/include/sys/signal.h*, which is part of the software development package.

FreeBSD also has a script called *killall*. As the name implies, it kills a group of processes, by name. If you find that you have, say, a whole lot of runaway *sendmail* processes, you might save the day by writing:

```
# killall sendmail
```

As we'll see elsewhere, you can also use *killall* to send a signal to a single process when you know that only one is present. For example, to cause *inetd* to re-read its configuration file, you could write:

```
# killall -1 inetd
```

Timekeeping

FreeBSD is a networking system, so keeping the correct time is more important than on a standalone system. Apart from the obvious problem of keeping the same time as other local systems, it's also important to keep time with systems in other time zones.

Internally, FreeBSD keeps the time as the number of seconds since the *epoch*, the beginning of recorded history: 00:00:00 UTC, 1 January 1970. *UTC* is the international base time zone, and means *Universal Coordinated Time*, despite the initials. It corresponds very closely, but not exactly, to Greenwich Mean Time (GMT), the local time in England in the winter. It would be inconvenient to keep all dates in UTC, so the system understands the concept of time zones. For example, in Walnut Creek, CA, the time zone in the winter is called *PST* (*Pacific Standard Time*), and in the summer it is *PDT* (*Pacific Daylight Time*). FreeBSD comes with a set of time zone description files in the directory hierarchy */usr/share/zoneinfo*. We've already seen on page 95 that when you install the system, it stores information about the local time zone in the file */etc/localtime*. If you move time zones, you should change the time zone, not the time, either by running the *tzsetup* program, or simply by copying the file. For example, if you travel with a laptop from Adelaide, South Australia, to San Francisco CA, you would do:

```
# cp /usr/share/zoneinfo/America/Los_Angeles /etc/localtime
```

When you get home again, you would do:

```
# cp /usr/share/zoneinfo/Australia/Adelaide /etc/localtime
```

At no time do you need to change the date or time directly.

Why `Los_Angeles` and not `San_Francisco`? The developers of the time zone package chose the largest city in the time zone. You need to have a certain understanding of the time zones to choose the correct one.

The TZ environment variable

An alternate means of describing the time zone is to set the environment variable `TZ`, which we looked at on page 128. You might use this form if you're connected to a remote system in a different time zone, or maybe just to find the time at some other place. For example, in Adelaide, SA I might find:

```
$ date
Sun Apr 14 13:31:15 CST 2002
$ TZ=America/Los_Angeles date
Sat Apr 13 21:01:15 PDT 2002
```

Set the `TZ` variable to the name of the time zone info file in the */usr/share/zoneinfo* hierarchy. For example, the value of `TZ` for Berlin, Germany is `Europe/Berlin` in FreeBSD.

This is not the same as the usage of the TZ variable in UNIX System V. System V doesn't have the time zone definition files in */usr/share/zoneinfo*, so the TZ variable tells it information about the time zone. If you were using System V in Berlin, you would set your TZ variable to MEZ1MSZ2, indicating time zone names and offsets from UTC.

Keeping the correct time

If you're connected to the Internet on a reasonably regular basis, there are a number of programs which can help you synchronize your time via the *ntp* (*Network Time Protocol*) service.

A number of systems around the world supply time information via the *ntp* service. Look at *http://www.eecis.udel.edu/˜mills/ntp/servers.html* to find one near you.

Your choice of program depends on the nature of your connection to the Internet. If you're connected full time, you'll probably prefer *ntpd*, which keeps the system synchronized. Otherwise you can use *ntpdate*, which you can run as you feel like it.

ntpd

ntpd performs periodic queries to keep the system synchronized with a time server. There are many ways to run it—see the man page *ntpd(8)*. In most cases, you can set up one system on the network to connect to an external time reference, and the other systems on the same Ethernet can get the time information from the first system.

To get the time from an external source and broadcast it to the other systems on the network, create a file */etc/ntp.conf* with a content like this:

```
server     227.21.37.18                    this address is invalid; check what's near you
driftfile  /etc/ntp.drift
broadcast  223.147.37.255
```

The first line defines the server. The value in this example is invalid , so don't try to use it. It's important to get one near you: network delays can significantly impair the accuracy of the results. *ntpd* uses the file */etc/ntp.drift* to record information about the (in)accuracy of the local system's clock. You only need the final line if you have other systems on the network which wait for a broadcast message. It specifies the broadcast address for the network and also tells *ntpd* to broadcast on this address.

After setting up this file, you just need to start *ntpd*:

```
# ntpd
```

To ensure that *ntpd* gets started every time you reboot, make sure that you have the following lines in */etc/rc.conf*:

```
ntpd_enable="YES"               # Run ntpd Network Time Protocol (or NO).
```

The comment on the first line is misleading: the value of ntpd_enable must be YES. You don't need any flags. You put exactly the same text in the */etc/rc.conf* on the other machines, and simply omit the file */etc/ntp.conf*. This causes *ntpd* on these machines to monitor broadcast messages.

In previous versions of FreeBSD, *ntpd* was called *xntpd*, so you may find things like `xntpd_enable` in your */etc/rc.conf*. If you do, you'll have to change the name.

ntpdate

If you connect to the Internet infrequently, *ntpd* may become discouraged and not keep good time. In this case, it's better to use *ntpdate*. Simply run it when you want to set the time:

```
 # ntpdate server
```

You can't use both *ntpdate* and *ntpd* at the same time: they both use the same port. *ntpd* takes quite some time to synchronize, and if the time is wildly out, it won't even try, so it's often a good idea to run *ntpdate* on startup and then start *ntpd* manually.

Log files

Various components of FreeBSD report problems or items of interest as they happen. For example, there can always be problems with mail delivery, so a mail server should keep some kind of record of what it has been doing. If hardware problems occur, the kernel should report them. If somebody tries to break into the machine, the components affected should report the fact.

FreeBSD has a generalized system for *logging* such events. The *syslogd* daemon takes messages from multiple sources and writes them to multiple destinations, usually log files in the directory */var/log*. You can change this behaviour by modifying the file */etc/syslog.conf*. See *syslog.conf(5)* for further details. In addition to *syslogd*, other programs write directly to files in this directory. The following files are of interest:

- *XFree86.0.log* contains the log file for the last (or current) X session started on display 0. This is a prime source of information if you run into problems with X.
- *auth.log* contains information about user authentication. For example, you might see:

  ```
  Dec 10 10:55:11 bumble su: grog to root on /dev/ttyp0
  Dec 10 12:00:19 bumble sshd[126]: Server listening on :: port 22.
  Dec 10 12:00:19 bumble sshd[126]: Server listening on 0.0.0.0 port 22.
  Dec 10 12:06:52 bumble sshd[167]: Accepted publickey for grog from 223.147.37.80
   port 49564 ssh2
  Dec 10 12:06:58 bumble su: BAD SU grog to root on /dev/ttyp0
  ```

  The first line is a successful *su* invocation; the last line is an unsuccessful one (because the password was mistyped). The messages at 12:00:19 are from *sshd* startup, and the message at 12:06:52 is a successful remote login with *ssh*.

- *cron* is a log file for *cron*. It’s relatively uninteresting:

  ```
  Jan  5 16:00:00 bumble newsyslog[2668]: logfile turned over
  Jan  5 16:05:00 bumble /usr/sbin/cron[2677]: (root) CMD (/usr/libexec/atrun)
  Jan  5 16:05:00 bumble /usr/sbin/cron[2678]: (root) CMD (/usr/libexec/atrun)
  Jan  5 16:10:00 bumble /usr/sbin/cron[2683]: (root) CMD (/usr/libexec/atrun)
  ```

  If you have problems with *cron*, that could change rapidly.

- *dmesg.today* and *dmesg.yesterday* are created by a *cron* job at 2 am every day. The *dmesg* message buffer wraps around, overwriting older entries, so they can be of use.
- *lastlog* is a binary file recording last login information. You don’t normally access it directly.
- *maillog* contains information about mail delivery.
- *messages* is the main log file.
- The files *mount.today* and *mount.yesterday* show the currently mounted file systems in the format needed for */etc/fstab*.
- The file *ppp.log* contains information on PPP connections. We look at it on page 353.
- The files *setuid.today* and *setuid.yesterday* contain a list of *setuid* files. The daily security check compares them and sends a mail message if there are any differences.
- The file *vinum_history* contains information about *vinum* activity.
- The file *wtmp* contains information about logins to the system. Like *lastlog*, it’s in binary form. See *utmp(5)* for the format of both *lastlog* and *wtmp*.

A number of the more important log files are kept through several cycles. As the example above shows, *cron* runs the *newsyslog* command every hour. *newsyslog* checks the size of the files, and if they are larger than a certain size, it renames the old ones by giving them a numerical extension one higher than the current one, then renames the base file with an extension *.0* and compresses it. The result looks like this:

```
-rw-r--r--  1 root  wheel  31773 Jan  5 13:01 messages
-rw-r--r--  1 root  wheel   8014 Jan  2 01:00 messages.0.bz2
-rw-r--r--  1 root  wheel  10087 Dec 15 14:00 messages.1.bz2
-rw-r--r--  1 root  wheel   9940 Dec  3 17:00 messages.2.bz2
-rw-r--r--  1 root  wheel   9886 Nov 16 11:00 messages.3.bz2
-rw-r--r--  1 root  wheel   9106 Nov  5 18:00 messages.4.bz2
-rw-r--r--  1 root  wheel   9545 Oct 15 17:00 messages.5.bz2
```

newsyslog has a configuration file */etc/newsyslog.conf*, which we discuss on page 572.

Multiple processor support

FreeBSD Release 5 can support most current Intel and AMD multiprocessor motherboards with the ia32 architecture. It also supports some Alpha, SPARC64 and Intel ia64 motherboards. Documentation on SMP support is currently rather scanty, but you can find some information at *http://www.freebsd.org/˜fsmp/SMP/SMP.html*.

The `GENERIC` kernel does not support SMP, so you must build a new kernel before you can use more than one processor. The configuration file */usr/src/sys/i386/conf/GENERIC* contains the following commented-out entries:

```
# To make an SMP kernel, the next two are needed
#options        SMP                     # Symmetric MultiProcessor Kernel
#options        APIC_IO                 # Symmetric (APIC) I/O
```

For other platforms, you don't need `APIC_IO`. See Chapter 33 for information on how to build a new kernel.

PC Card devices

As we have already seen, PC Card devices are special because they can be hot-plugged. They are also intended to be recognized automatically. Starting with Release 5, FreeBSD recognizes card insertion and removal in the kernel and invokes the appropriate driver to handle the event. When you insert a card you will see something like this on the system console:

```
ata2 at port 0x140-0x14f irq 11 function 0 config 1 on pccard0
ad4: 7MB <LEXAR ATA FLASH> [251/2/32] at ata2-master BIOSPIO
```

This is a compact flash memory card, which the system sees as an ATA disk. The kernel has created the necessary structures, but it can't know how to mount the device, for example. We'll look at what we can do about this in the next section.

devd: The device daemon

The device daemon, *devd*, provides a way to run userland programs when certain kernel events happen. It is intended to handle userland configuration of PC Card devices such as Ethernet cards, which it can do automatically. We'll look at this automatic usage on page 304.

devd reads the kernel event information from the device */dev/devctl* and processes it according to rules specified in the configuration file */etc/devd.conf*, which is installed with the system. If you want to use it for other devices, you must modify */etc/devd.conf*. This file contains a number of sections, referred to as *statements* in the man page:

- The *options* statement describes file paths and a number of regular expressions (patterns) to look for in the messages it reads from */dev/devctl*.

- *attach* statements specify what action to perform when a device is attached. For example:

  ```
  attach 0 {
          device-name "$scsi-controller-regex";
          action "camcontrol rescan all";
  };
  ```

  The `device-name` entry uses the regular expression `$scsi-controller-regex` to recognize the name of a SCSI controller in the attach message. The `action` entry then specifies what action to take when such a device is attached to the system. In this case, it runs the *camcontrol* program to rescan the SCSI buses and recognize any new devices that have been added.

  Multiple *attach* statements can match a specific event, but only one will be executed. The order in which they are checked is specified by a *priority*, a numerical value after the keyword `action`. The statements are checked in order of highest to lowest numerical priority.

- *detach* statements have the same syntax as *attach* statements. As the name suggests, they are executed when a device is detached.

  It's not always possible or necessary to perform any actions when a device is removed. In the case of SCSI cards, there is no *detach* statement. We'll look at this issue in more detail below.

- Finally, if the kernel was unable to locate a driver for the card, it generates a *no match* event, which is handled by the *nomatch* statement.

So what does *devd* do when we insert the compact flash card? By default, nothing. The ATA driver recognizes and configures the card. It would be nice to get *devd* to mount it as well. That's relatively simple:

- Ensure that you have an entry for the device in */etc/fstab*. Digital cameras create a single MS-DOS file system on flash cards. An appropriate entry in */etc/fstab* for this device might be:

  ```
  /dev/ad4s1              /camera         msdos   rw,noauto       0       0
  ```

  This is a removable device, so you should use the `noauto` keyword to stop the system trying to mount it on system startup.

- In the *options* section of */etc/devd.conf*, add an expression to recognize the names of ATA controllers:

  ```
          set ata-controller-regex
                  "ata[0-9]+";
  ```

- Add an *attach* section for the device:

```
    attach 0 {
            device-name "$ata-controller-regex";
            action "mount /camera";
    };
```

- Restart *devd*:

```
    # killall devd
    # devd
```

After this, the file system will be automatically mounted when you insert the card.

Removing PC Card devices

The next thing we'd like to do is to unmount the file system when you remove the flash card. Unfortunately, that isn't possible. Unmounting can involve data transfer, so you have to do it before you remove the card. If you forget, and remove the card without unmounting, the system may panic next time you try to access the card.

After unmounting, you can remove the card. On the console you'll see something like:

```
ad4: removed from configuration
ad4: no status, reselecting device
ad4: timeout sending command=e7 s=ff e=04
ad4: flushing cache on detach failed
ata2: detached
```

Alternate PC Card code

The PC Card implementation described here, called *NEWCARD*, is new in FreeBSD Release 5. At the time of writing, the older implementation, called *OLDCARD*, is still included in the system. It's possible that you might have an older card that is supported by OLDCARD but not by NEWCARD. In that case, you will need to build a kernel with OLDCARD support. Check the *NOTES* files in */usr/src/sys/conf* and */usr/src/sys/*`arch`*/conf*, where *arch* is the architecture of your system, and the man pages *pccardd* and *pccard.conf*.

Configuring PC Card devices at startup

A number of entries in */etc/rc.conf* relate to the use of PC Card devices, but nearly all of them are for OLDCARD. You only need one for NEWCARD:

```
devd_enable="YES"
```

This starts *devd* at system startup.

Emulating other systems

A large number of operating systems run on Intel hardware, and there is a lot of software that is available for these other operating systems, but not for FreeBSD.

Emulators and simulators

There are a number of ways to execute software written for a different platform. The most popular are:

- *Simulation* is a process where a program executes the functions that are normally performed by the native instruction set of another machine. They simulate the low-level instructions of the target machine, so simulators don't have to run on the same kind of machine as the code that they execute. A good example is the port *emulators/p11*, which simulates a PDP-11 minicomputer, the machine for which most early versions of UNIX were written.

  Simulators run much more slowly than the native instruction set: for each simulated instruction, the simulator may execute hundreds of machine instructions. Amusingly, on most modern machines, the *p11* emulator still runs faster than the original PDP-11: modern machines are over 1,000 times faster than the PDP-11.

- In general, *emulators* execute the program instructions directly and only simulate the operating system environment. As a result, they have to run on the same kind of hardware, but they're not noticeably slower than the original. If there is any difference in performance, it's because of differences between the host operating system and the emulated operating system.

- Another use for the term *emulator* is where the hardware understands a different instruction set than the native one. Obviously this is not the kind of emulator we're talking about here.

FreeBSD can emulate many other systems to a point where applications written for these systems will run under FreeBSD. Most of the emulators are in the Ports Collection in the directory */usr/ports/emulators*.

In a number of cases, the emulation support is in an experimental stage. Here's an overview:

- FreeBSD will run most BSD/OS programs with no problems. You don't need an emulator.
- FreeBSD will also run most NetBSD and OpenBSD executables, though not many people do this: it's safer to recompile them under FreeBSD.
- FreeBSD runs Linux executables with the aid of the *linux kld* (*loadable kernel module*). We'll look at how to use it in the next section.
- FreeBSD can run SCO COFF executables with the aid of the *ibcs2 kld*. This support is a little patchy: although the executables will run, you may run into problems caused by differences in the directory structure between SCO and FreeBSD. We'll

look at it on page 164.

- A *Microsoft Windows* emulator is available. We'll look at it on page 165.

Emulating Linux

Linux is a UNIX-like operating system that in many ways is very similar to FreeBSD. We discussed it on page 10. Although it looks very UNIX-like, many of the internal kernel interfaces are different from those of FreeBSD or other UNIX-based systems. The Linux compatibility package handles these differences, and most Linux software will run on FreeBSD. Most of the exceptions use specific drivers that don't run on FreeBSD, though there is a considerable effort to minimize even this category.

To install the Linux emulator, you must:

- Install the compatibility libraries. These are in the port */usr/ports/emulators/linux_base*.
- Run the Linux emulator kld, *linux*.

Running the Linux emulator

Normally you load the Linux emulator when you boot the system. Put the following line in your */etc/rc.conf*:

```
linux_enable="YES"
```

If you don't want to do this for some reason, you can start it from the command line:

```
# kldload linux
```

You don't interact directly with the emulator module: it's just there to supply kernel functionality, so you get a new prompt immediately when you start it.

linux is a kld, so it doesn't show up in a *ps* listing. To check whether it is loaded, use *kldstat*:

```
$ kldstat
Id Refs Address    Size     Name
 1    5 0xc0100000 1d08b0   kernel
 2    2 0xc120d000 a000     ibcs2.ko
 3    1 0xc121b000 3000     ibcs2_coff.ko
 5    1 0xc1771000 e000     linux.ko
```

This listing shows that the SCO UNIX emulation (*ibcs2*) has also been loaded.

The Linux emulator and many Linux programs are located in the directory hierarchy */usr/compat/linux*. You won't normally need to access them directly, but if you get a Linux program that includes libraries destined for */lib*, you will need to manually place them in */usr/compat/linux/lib*. Be ***very*** careful not to replace any files in the */usr/lib* hierarchy with Linux libraries; this would make it impossible to run FreeBSD programs that depend on them, and it's frequently very difficult to recover from such problems. Note that FreeBSD does not have a directory */lib*, so the danger is relatively minor.

Linux procfs

Linux systems have a file system called *procfs*, or *Process File System*, which contains information used by many programs. FreeBSD also has a *procfs*, but it is completely different. To be able to run Linux programs which refer to *procfs*, place the following entry in your */etc/fstab* file:

```
linproc        /compat/linux/proc  linprocfs   rw      0       0
```

Problems executing Linux binaries

One of the problems with the ELF format used by older Linux binaries is that they may contain no information to identify them as Linux binaries. They might equally well be BSD/OS or UnixWare binaries. That's normally not a problem, unless there are library conflicts: the system can't decide which shared library to use. If you have this kind of binary, you must *brand* the executable using the program *brandelf*. For example, to brand the StarOffice program *swriter3*, you would enter:

```
# brandelf -t Linux /usr/local/StarOffice-3.1/linux-x86/bin/swriter3
```

This example deliberately shows a very old version of StarOffice: it's not clear that there are any modern binaries that cause such problems.

Emulating SCO UNIX

SCO UNIX, also known as *SCO OpenDesktop* and *SCO Open Server*, is based on UNIX System V.3.2. This particular version of UNIX was current in the late 1980s. It uses an obsolete binary format called *COFF* (*Common Object File Format*).

Like Linux support, SCO support for FreeBSD is supplied as a loadable kernel module. It's not called *sco*, though: a number of older System V.3.2 systems, including Interactive UNIX, also support the *ibcs2*[1] standard. As a result, the kld is called *ibcs2*.

Run ibcs2 support like Linux support: start it manually, or modify */etc/rc.conf* to start it automatically at bootup:

```
ibcs2_enable="YES"      # Ibcs2 (SCO) emulation loaded at startup (or NO).
```

Alternatively, load the kld:

1. *ibcs2* stands for *Intel Binary Compatibility System 2*.

```
# kldload ibcs2
```

One problem with SCO emulation is the SCO shared libraries. These are required to execute many SCO executables, and they're not supplied with the emulator. They *are* supplied with SCO's operating systems. Check the SCO license to determine whether you are allowed to use them on FreeBSD. You may also be eligible for a free SCO license—see the SCO web site for further details.

Emulating Microsoft Windows

The *wine* project has been working for some time to provide an emulation of Microsoft's *Windows* range of execution environments. It's changing continually, so there's little point describing it here. You can find up-to-date information at *http://www.winehq.com/about/*, and you can install it from the port *emulators/wine*. Be prepared for a fair amount of work.

Accessing Microsoft files

Often you're not as interested in running Microsoft applications as decoding their proprietary formats. For example, you might get a mail message with an attachment described only as

```
[-- Attachment #2: FreeBSD.doc --]
[-- Type: application/octet-stream, Encoding: x-unknown, Size: 15K --]

[-- application/octet-stream is unsupported (use 'v' to view this part) --]
```

This attachment has an unspecific MIME type,[1] but you might guess that it is Microsoft Word format because the file name ends in *.doc*. That doesn't make it any more legible. To read it, you need something that understands the format. A good choice is *OpenOffice.org*, a clone of Microsoft's ''Office'' product. Install from the Ports Collection (*/usr/ports/editors/openoffice*).

OpenOffice.org is not a good example of the UNIX way. It breaks a number of conventions, and in general it's a lot more difficult to use than normal FreeBSD tools. Its only real advantage is that you can process Microsoft document formats.

1. See Chapter 26, *Electronic mail: clients*, page 489, for more information about MIME.