6

STARTUP AND BOOTING

Single-user mode
unscheduled in the nighttime?
Something just went "boom!"

Now that you have performed some basic configuration of your OpenBSD system, we're going to look at the startup process. To properly manage any computer platform, you must understand its booting system.

In general, when a computer boots it fires up the built-in operating system, or BIOS. The BIOS figures out little things like what hard drives are attached, what sort of CPU is installed, how much memory is available, and so on, then loads a minimal boot handling program from one of the hard drives. On i386 systems, this is where the Master Boot Record comes in; other hardware platforms have their own method of bootstrapping the operating system. This boot loader finds and starts the kernel, and the kernel starts the operating system, attaches device drivers to hardware, and performs other operating system setup. Finally, the kernel starts init(8), which starts various processes and enables the user programs, network interfaces, daemons, and so on. Large chunks of this cannot be managed — nobody actually configures init(8)! However, many parts of the process can be managed.

First we'll discuss OpenBSD's cross-platform booting process. At the end of that section you'll understand how single-user mode works and why it's there, plus the most useful of the things you can do at the boot loader. Then we'll learn how to set up and use a serial console, a vital task for remote system administration. While some hardware integrates serial console support into the hardware, the i386 platform doesn't. Setting up a serial console is fairly straightforward, as is accessing the serial console from another computer.

Much of what people consider "system configuration" is actually handled by the shell script /etc/rc, which is started by init(8). All sorts of system features, file systems, and daemons are configured during this process. While we discussed the various options available in /etc/rc.conf in the previous chapter, we didn't touch on how those options are actually used. We'll explain how the configuration process works, and the OpenBSD configuration options that are available out-of-the-box.

Lastly, we'll discuss how to automatically start or stop programs when the system boots and shuts down.

Boot Configuration

When your hardware's BIOS finishes counting onboard memory and finding all the system hard drives, it will pass control of the system to the boot loader. The boot loader is a small program that handles initial system configuration and booting the kernel. OpenBSD provides the ability to interrupt the booting process, configure the system before it boots, and adjust your kernel settings, or even boot an alternate kernel. This program is documented in boot(8), but we'll cover some of the basic functions here.

Boot Prompt

When your hardware hands control of the boot process over to the OpenBSD partition, you'll see a prompt much like this.

```
boot>
```

The boot loader runs from the BIOS bootstrap loader, and provides very rudimentary configuration abilities. The boot program's main purpose is to load the kernel into memory and start it. The boot loader loads the kernel, waits for five seconds, and starts the kernel. Because this runs before the kernel starts, the boot program gives you the opportunity to issue pre-booting instructions to the kernel.

Delaying the Boot

Once this prompt has been idle for five seconds, the system will boot! If you're not exactly sure what you're doing, you might want to tell it to delay the boot for a little longer. You can do this by increasing the timeout value.

```
boot> set timeout 60
boot>
```

This tells OpenBSD to boot after being idle for 60 seconds, which is not an unreasonable delay when you're poking around the boot loader trying to figure out what you want to do! Now, let's look at some more useful functions than slowing your system down.

Booting Single-User

Single-user mode is the earliest point where your OpenBSD system can give you a command prompt. At this point the kernel has probed all the hardware, attached drivers to hardware it's going to acknowledge, and started init(8). No file systems are mounted, except for a read-only root partition. The network is not started, no daemons are running, security is not implemented, and file system permissions are ignored. You get a bare-naked command prompt on a minimally running system.

To boot into single-user mode, use the -s flag on the boot command.

```
boot> boot -s
```

Why would you want to use single-user mode? Suppose you've upgraded a program on your system, and it now crashes your computer on every boot, before you can log in. A computer can easily get caught in a panic loop where it crashes and restarts until someone manually intervenes and shuts down the offending program. You might have a disk go bad, and crash the system before the boot can finish. Perhaps you made some stupid mistake in configuring your system, and now it just won't finish booting at all, or perhaps you need to clear some file flags (see Chapter 10). Any of these require intervention before the boot finishes.

Generally speaking, you want a fully functional file system before doing much of anything in single-user mode. If your system crashed, you'll have to check the file system consistency before mounting any file systems. The following commands will clean and mount all of your file systems. (fsck(8) and mount(8) have many more options; check out Chapter 18 for the most common ones or the man page for the full gory details.)

```
# fsck -p
# mount -a
```

Once you're in single-user mode and have your file systems mounted, all of the usual command-line functions should be available. You can edit configuration files, start and stop programs, and generally do whatever you like. What exactly you want to do depends on exactly what your problem is.

Starting the Network in Single-User Mode

The shell script /etc/netstart can start the network while in single-user mode. You could go and run all the appropriate commands by hand, but /etc/netstart will read the appropriate /etc/ files and do all the grunt work for you. You need to explicitly run this script through sh(1).

```
# /bin/sh /etc/netstart
```

Of course, if network configuration problems are *why* you're running in single-user mode, this script will only re-create your problem!

Booting in Kernel Configuration Mode

The -c flag to boot makes the system come up in kernel configuration mode. This allows you to change some of a kernel's built-in constants. We'll discuss this at great length in Chapter 11. For now, you just need to know that the mode exists so that other examples here make some sort of sense.

Booting Alternate Kernels

You can choose to boot a kernel other than /bsd. You might need this if you're building your own kernel, as discussed in Chapter 11. Just give the boot command the full path to the kernel you want to boot. For example, if your kernel in /bsd is faulty and you need to boot off your known-good /bsd.GENERIC kernel, do the following:

```
boot> boot /bsd.GENERIC
```

This should get your system up and running and let you install a proper kernel in /bsd.

You can use any other boot flags with this. For example, boot -s /bsd.GENERIC will boot the GENERIC kernel in single-user mode.

Booting from an Alternate Hard Disk

You might have multiple OpenBSD installs on different hard disks on one computer, for either testing or redundancy purposes. By default, OpenBSD boots from the first disk it finds. If you have four IDE disks, for example, it boots from the first disk on the first IDE controller. You could have a separate root partition installed on another disk, with a separate /etc/fstab pointing to emergency /usr and /var partitions.

To tell the boot loader to use a root partition on another drive give the full path to the kernel you want to boot, including the device name of the drive the root partition is on. This is much like booting an alternate kernel, just adding the hard drive device name as part of the path. Here, we boot the kernel /bsd.old on the "a" partition (traditionally root) on the third IDE hard disk, also known as "wd2a." (If you have four IDE disks, this is the master drive on the second controller.)

```
boot> boot wd2a:/bsd.old
```

The OpenBSD installer will do its very best to put the root partition on the "a" partition, but if you managed to put it elsewhere you will have to enter the proper partition here.

Other Useful Boot Commands

If you forget which kernels you have on a system, the "ls" command lists all the files in the root directory. You can list other directories on the root partition by giving a full path, i.e., "ls /etc."

The "boot" command by itself will boot the system immediately, without waiting for the five-second timeout. Similarly, the "reboot" command tells the system to do a warm boot.

The "help" command lists all available boot loader commands, including the less frequently used ones that we don't discuss here. If you want truly detailed help with the boot loader, however, you should go read the boot(8) man page.

Finally, you can combine the boot flags to achieve exactly the effects you want. To boot an old kernel in single-user mode, you would do this:

```
boot> boot -s /bsd.old
```

/etc/boot.conf

The /etc/boot.conf file allows you to permanently reconfigure the system's booting process. Entries in this file are parsed before you get the boot prompt, so you have the opportunity to override anything you enter in this file. Commands here are parsed and processed automatically.

You can tell your OpenBSD system to boot a different kernel every time giving the command here. If /etc/boot.conf contains the following, your system will automatically boot using the kernel file /bsd.CUSTOM:

```
boot /bsd.CUSTOM
```

You can change the boot prompt timeout by setting it here as well. For example, if a five-second delay is just too long and you want barely enough time to hit the spacebar and start typing before the system boots, you might set your timeout to two seconds.

```
set timeout 2
```

By far, however, the most popular use of /etc/boot.conf is to configure a serial console.

Serial Consoles

All these nifty boot functions let you do some pretty useful things in trouble situations, but how are you supposed to use them if your server isn't right in front of you? If your computer is on the other side of the country or wedged uncomfortably behind the last ten years of payroll records in the basement storeroom, and you want to perform some low-level hardware maintenance, a serial console will make your life far more pleasant.

A true serial console allows you to run a serial cable between two computers and have complete access to the hardware BIOS, the early operating system boot messages, and startup processes. One computer (the client) will be able to see all the messages that appear on the console of the booting machine (the server). This makes remote system management much easier. Serial consoles are invaluable when you're trying to debug a system crash — the debugging messages come over the serial port where they can be captured easily, rather than displayed on a glass screen to be copied by hand.

Real UNIX hardware (such as HP and Sparc) has a serial console capability. Most i386 hardware does not support this functionality. A very few Intel motherboards, such as the L440GX, do support serial consoles, but it's a feature you must specifically shop around for.

Because i386 hardware is the most common these days, that lack is something of a problem. Fortunately, it's possible to work around this and build a highly functional serial console anyway. While OpenBSD's i386 serial console doesn't give you access to the hardware BIOS, it does let you interface with the OpenBSD boot process. You could also choose to install an actual hardware serial console.

Hardware Serial Console

Nothing any operating system can do will give you access to the i386 BIOS messages across a serial port. This stuff happens before the operating system starts and before the hard drive even starts to spin up.

Some hardware solutions can work around this by pretending to be a video board and directing the console out to a serial port. The best I've seen is the PC Weasel (http://www.realweasel.com/). By putting the Weasel in your computer and running a null modem cable between the Weasel and another computer's serial port you can manipulate the BIOS remotely, interrupt the boot to come up in single-user mode, and generally muck around with the hardware just as if you were at the actual keyboard and monitor attached to the system. Other companies do manufacture similar devices, but they either require proprietary client software or are far more expensive.

Software Serial Console

OpenBSD includes a software serial console. As OpenBSD boots, it decides where to put its console. This defaults to the monitor and keyboard, but with a few tweaks you can have the console come up on a serial port. The only hardware requirement is that your system has a serial port. Some systems are increasingly arriving "legacy-free," meaning that they lack an ISA bus, serial ports, and even PS/2 ports. My latest laptop had a nasty surprise in lacking an actual serial port. You might need to buy a PCI serial card for your server if this is the case.

This serial console does not kick in until the OpenBSD boot loader starts, so you will not see the BIOS messages. You do get a chance to interact with the OpenBSD boot process, which is good enough for most cases — after all, you presumably made sure that the BIOS was correct before shipping the computer across the country!

Non-i386 Serial Consoles

Every different hardware platform has its own standards for serial consoles. If you're running on one of these platforms, check your hardware documentation. In general, if your hardware supports serial consoles, you need to set it up at the hardware level. Your Sparc hardware will support OpenBSD's console just as well as it supports Solaris's console.

Serial Console Physical Setup

You must have a null modem cable to use a serial console. A regular modem cable will not work! Get the best cable you can find; if you have an emergency and need the serial console, you're probably not in the mood to deal with line noise.

Plug one end of the null modem cable into the first serial port on your OpenBSD server. Traditionally, this is the first COM port. You can use any serial port that is convenient, so long as you remember which port it is.[1] You can choose to use any serial port as your serial console. Plug the other end of your null modem cable into an open serial port on another system. I recommend that you use either another OpenBSD or UNIX system, or a terminal server if you have a lot of servers that include serial consoles. You can use a Windows system as your serial console terminal, but that won't give you any remote-control functionality. (Yes, you could use VNC or Windows Terminal Services on the Windows system, but you're starting to look at a complicated and error-prone setup when a simple 486 running OpenBSD would do.) In a pinch, on a local system that didn't have a monitor or keyboard, I've used a vt100 emulator running on my PalmPilot — the screen was cramped, but it worked.

If you have two OpenBSD machines at a remote location and want to use serial consoles on both of them, simply attach the console cable to the second serial port on the other server. If you have three machines, you can daisy-chain them into a loop. By combining twos and threes, you should be able to get a serial console on every one of your systems. I've worked in areas with dozens of UNIX servers tightly packed together, and serial consoles saved a huge amount of space that monitors and keyboards would have taken up.

[1] A surprising number of people go to a lot of trouble to set up a serial port, then either forget which port it is on or forget which physical port is actually COM1.

Serial Console Client

Before you can test your serial console, you need to configure your client to access the serial console. The key to setting up your client is to remember the following:

- 9600 baud
- 8 bits
- no parity
- 1 stop bit

If you can configure your client program to use these settings, the serial console will "just work." Conveniently enough, these are the default settings on Microsoft's HyperTerm program.[2] If you don't like HyperTerm, you can find any number of vt100 terminal programs for Microsoft platforms. Even Macintosh and Palm platforms have any number of free vt100 terminal programs kicking around. If your second computer also runs OpenBSD or, for that matter, almost any version of open-source UNIX, you can use the OpenBSD terminal program. Because this is an OpenBSD book, we'll discuss exactly how to do this.

OpenBSD accesses serial lines with tip(1), a program that allows you to connect to a remote system in a manner similar to telnet. To run tip and have it connect to a remote machine's serial port over a serial cable connected to the local machine's first serial port, do this:

```
# tip tty00
```

A port name is shorthand for specifying the settings and speed to be used when accessing a serial port. The file /etc/remote contains a list of port names for a variety of platforms.

Configuring the Serial Console

You can tell OpenBSD to boot either off the serial console or off the physical console, by an entry in /etc/boot.conf or a command at the boot prompt. The "set tty" command tells OpenBSD where to put the console. The common choices are com0 (for classic i386 COM1), com1 (for classic i386 COM2), or pc0 (for the physical hardware).

Plug in your serial console and access it from a client machine. Now reboot your test OpenBSD system. At the initial boot loader prompt, type:

```
boot> set tty com0
```

All of a sudden, your physical keyboard won't seem to be doing anything, and nothing else comes across your screen. On the other hand, your serial console client will abruptly show the boot loader prompt.

[2] I'm refraining from making any comments about how this one of those rare times that Microsoft has done anything conveniently. That would be too cheap a shot even for me.

```
boot>
```

Anything you type in your serial console client is passed to the OpenBSD boot loader, just as we discussed in "Boot Configuration" earlier. It's just as if you were at the console. You can load alternate kernels, perform preboot configuration (as discussed in Chapter 11), boot in single-user mode, and do any of the other booting tricks we discuss in this chapter.

To switch back to the PC's physical console, use the pc0 device.

```
boot> set tty pc0
```

The keyboard and monitor will work again.

If you want to use the serial console permanently, you can place a "set tty" entry in /etc/boot.conf.

```
set tty com0
```

Multiuser Startup

We examined /etc/rc.conf in some detail in Chapter 5. Now let's see how those variables are processed by the system.

Whenever your system boots to the point where it can execute userland commands, it runs the shell script /etc/rc. This script mounts all file systems, brings up the network interfaces, configures device nodes, sets up shared libraries, and does all the other tasks required to bring a system up to multiuser mode. These are an awful lot of tasks, and some of them aren't necessary on all systems. The purpose of /etc/rc.conf is to tell /etc/rc what to run, what values to run with, and what to not bother with. Everything you set in /etc/rc.conf is used in /etc/rc in one way or another. The /etc/rc system actually has six associated files: /etc/rc, /etc/rc.conf, /etc/rc.local, /etc/rc.securelevel, /etc/netstart, and /etc/rc.shutdown.

/etc/rc

Every configuration step on an OpenBSD box, from setting the host name to starting server programs, can be performed by a simple shell command. As such, /etc/rc is a basic shell script. This script reads in variable assignments from /etc/rc.conf as well as files such as /etc/myname and /etc/hostname.*, and acts as those variables tell it to. The /etc/rc script also starts every other /etc/rc script at the appropriate time. When /etc/rc exits, the system fires up getty(8) and presents login prompts on all the appropriate terminals.

Generally speaking, you should not need to edit /etc/rc unless you are a very experienced systems administrator with truly unique needs. Editing the other /etc/rc.* files, especially /etc/rc.conf, should do everything you need.

/etc/rc.conf

This file contains nothing but variables used by other /etc/rc scripts. We covered it in extreme detail in Chapter 5. Various other /etc/rc.* scripts use /etc/rc.conf to get their configuration information.

/etc/netstart

While the name doesn't look like the others, /etc/netstart is definitely a system startup script. This script reads /etc/hostname.if*, /etc/mygate, and /etc/myname, and uses that information to configure all network functionality: interfaces, bridges, routing, and so forth. You can run this script in single-user mode to bring up the network without starting any of the other software that normally starts in multi-user mode.

/etc/rc.securelevel

This shell script runs just before the system raises its securelevel (see more about this in Chapter 10), but after the network is started. Many programs, particularly those that affect the kernel or file systems in some way, will not run once the securelevel is raised. The examples in the file relate to ntpd(8) and related programs, but you can edit /etc/rc.securelevel to include any programs that must be run before securelevel is raised. If at all possible, however, you're better off starting local programs from /etc/rc.local. We'll look at adding proper shell commands to these files in "Editing /etc/rc Scripts," later in this chapter.

One important detail in /etc/rc.securelevel is the securelevel setting itself. We discuss securelevel in Chapter 10. For now, just don't touch the line that sets the securelevel unless you're already familiar with BSD and know exactly what you're getting with securelevels!

/etc/rc.local

The /etc/rc.local shell script runs at the very end of system initialization. Once every other system process has been started, /etc/rc.local runs. This is the usual place to put startup commands for systems such as databases, small servers, and any other programs you want to run at boot time. You can place your add-on shell commands here, as discussed in "Editing /etc/rc Scripts," later in this chapter.

rc.conf.local

In various circumstances, you might not want to edit /etc/rc.conf for each machine. Perhaps you share one rc.conf amongst several machines, but have a few machines that require particular tweaks. If you're a developer and upgrade frequently, handling /etc/rc.conf can be tedious. That's where /etc/rc.conf.local comes in.

/etc/rc.conf.local starts off as an empty file. You can put any rc.conf variable assignments you like into this file. Entries in /etc/rc.conf.local override any values in /etc/rc.conf. For example, /etc/rc.conf contains this line.

```
identd_flags=NO
```

Let's suppose you want to change this value without editing the /etc/rc.conf line. You could create a line like the following in /etc/rc.conf.local.

```
identd_flags="-b -u nobody -elo"
```

When /etc/rc runs, it will use the values from /etc/rc.conf.local instead of /etc/rc.conf. This minimizes the number of changes necessary to /etc/rc.conf and makes upgrading easier.

/etc/rc.shutdown

The /etc/rc.shutdown script runs whenever you use reboot(8), halt(8), or a keyboard shutdown (i.e., CTRL-ALT-DELETE on i386). The commands here are shut down commands that require specialized shutdown sequences. Database programs use this feature frequently, which you need to shut down correctly to prevent data loss.

Editing /etc/rc Scripts

Well, now that you know how the files fit together, what are you supposed to do with them? While OpenBSD's integrated software is started by /etc/rc, add-on software needs to be started separately. The ports and packages system tells you how to create these script commands and where to put them. If you install your own software, however, you need to create a script that handles its startup and shutdown process. Plus, to change an existing add-on package's startup process, you must understand how the script works.

Port-Based Software Startup

A port (or package) is a piece of add-on software that has been configured for OpenBSD. We discuss ports and packages at great length in Chapter 13. If your port needs to have a startup sequence added to an /etc/rc script to work, the installation process will tell you exactly what to add to which /etc/rc file. It should tell you to add some lines of shell script to either /etc/rc.local or /etc/rc.securelevel. For example, if you install the SNMP port, ucd-snmp, you'll see the following message at the end of the install process:

```
...
| To have snmpd start at boot time, you must have an entry similiar to the
| following in /etc/rc.local.
|
|       ❶if [ -x /usr/local/sbin/snmpd ]; then
|               ❷/usr/local/sbin/snmpd ❸-c /etc/snmpd.conf && ❹echo -n ' snmpd'
|       ❺fi
|
| This will start snmpd and use /etc/snmpd.conf for the configuration.
| (see snmpd(1) and snmpd.conf(5) for more options)
```

You can literally just copy the text you're given and add it to /etc/rc.local, and it will work. But understanding what you're looking at here, and why it works, will make you a better sysadmin. If you want to start your program in a slightly different manner, you'll have to edit this.

The first line ❶ checks for the existence of the /usr/local/bin/snmpd file. If that file exists, the script executes the next lines, up until the ❺fi (or "finish") entry. If there is no such program, then the rest of this little script is skipped entirely. The next line has the real meat of the script. The startup system will run ❷/usr/local/bin/snmpd, with the arguments ❸-c /etc/snmpd.conf, and it will print to the console ❹"snmpd" so you'll know it started.

It would be simple enough to have a port automatically add its startup information to /etc/rc.local or /etc/rc.securelevel and save you a step. This could potentially be a security hole, however! For example, I frequently install the net-snmp package just to get the cool SNMP client tools it includes. I don't want the SNMP server daemon to be running. More than once, on other UNIX-like operating systems, I've installed this package and completely forgotten about its daemon portion. My system is running a daemon I don't want it to be running, until I either remember or notice and manually shut it off. OpenBSD absolutely requires you to enable every daemon that runs on the system, even once you've installed the binaries for it.

Uninstalls

When you uninstall this piece of software, remove the corresponding startup entry from the /etc/rc script. The script will not cause even minor problems by being there, but it is rather sloppy to not clean up after yourself.

Custom Software Startup

Suppose you install a piece of software by hand, not using a port or package, and need to have it start automatically? That's simple enough to deal with. Just write a bit of shell code much like the entry a port gives. Your startup command doesn't have to bother checking to see if the piece of software is installed, mind you. You could just add the line to start the program to /etc/rc.local.

```
/usr/local/sbin/snmpd -c /etc/snmpd.conf
```

It's not that much harder to add a notification that the program started to your console messages.

```
/usr/local/sbin/snmpd -c /etc/snmpd.conf && echo -n ' snmpd'
```

If you stop here, your program will run just fine.

Uninstalls

When you uninstall the program, be sure to remove the matching /etc/rc.local entry.

If you uninstall the program without removing the /etc/rc.local entry, you'll start to see errors on boot complaining that "/usr/local/sbin/snmpd" does not exist. In my opinion, this is actually desirable behavior — all that the fancy check to see if a program exists really does is silence warnings when the program is gone, but the /etc/rc.local entry remains. I'm not sure how anyone could actually exploit such a script check without already having fairly deep access to the system, but it's sloppy in any event. And sloppiness is the biggest cause of system break-ins.