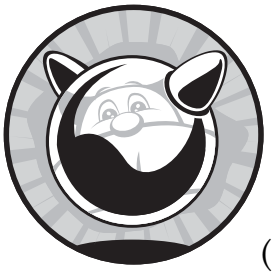
8

DISKS AND FILESYSTEMS



The importance of managing filesystems and disks cannot be overemphasized. (Go ahead, try to emphasize it too much. I'll wait.) Your disks contain your data, making reliability and flexibility paramount to the operating system. FreeBSD supports a variety of filesystems and has many different ways to handle them. In this chapter we'll consider the most common disk tasks every system administrator performs.

Disk Drives 101

Most people treat disk drives as fragile magic boxes. If you treat a drive badly, you can make the drive screech and grind, and with enough abuse, you can let the magic smoke escape so it will never work again. To really understand

filesystems, you must know a little bit about what's going on inside the drive. If you have a dusty old disk drive that you no longer have any respect for, feel free to crack the case and follow along.

Inside the hard drive case you'll find a stack of round aluminum or plastic disks, commonly called *platters*. When the drive is active, the platters spin at thousands of revolutions per minute. The RPM count on hard drives measures platter rotation speed.

A thin layer of magnetic media covers the platters. This magnetic material is arranged in thousands of circular rings, called *tracks*, that extend from the platter's inner core to its outer edge, much like the growth rings in a tree. These tracks hold data as strings of zeros and ones. Each track is subdivided into *sectors*. Each sector on the outer tracks holds more data than a corresponding sector on an inner track, and reading a constant amount of data takes less time on an outer track than on an inner track because any point on the outer track is moving faster.

Heads, mounted over each platter, write and read data as the platters pass by, much like a phonograph needle. These heads can read and write data quickly, but they must wait for the disk to move into the proper position under them. Drive performance basically boils down to how quickly those platters can move under the drive heads, which is what makes RPM important.

ATA, SATA, SCSI, AND SAS

I assume that you're familiar with the basics of the standard disk storage technologies. If you're not, please spend a few moments online with any of the excellent tutorials on these subjects, or even the brief articles available on Wikipedia. I'll make suggestions here and there about how these technologies can be used, but SCSI IDs and LUNs are a subject for another book.

Device Nodes

We touched briefly on device nodes in Chapter 3, but let's consider them in more detail. Device nodes are special files that represent hardware on the system. They're used as logical interfaces between user programs and either a device driver or a physical device. By using a command on a device node, sending information to a device node, or reading data from a device node, you're telling the kernel to perform an action upon a physical device. These actions can be very different for different devices—writing data to a disk is very different than writing data to a sound card. All device nodes exist in */dev*.

Before you can work with a disk or disk partition, you must know its device name. FreeBSD disk device nodes come from the names of the device drivers for that type of hardware. Device driver names, in turn, come from the chipset used in the device and not from what the device appears to be.

Table 8-1 shows the most common disk device nodes. See the man page for each if you want the full details.

Table 8-1: Storage Devices and Types

Device Node	Man Page	Description
*/dev/fd**	fdc(4)	Floppy disks
*/dev/acd**	acd(4)	IDE CD drives
*/dev/ad**	ad(4)	ATA and SATA hard disks and partitions
*/dev/cd**	cd(4)	SCSI CD drives
*/dev/da**	da(4)	SCSI and SAS hard disks, USB and flash storage, etc.

Disks attached to most hardware RAID controllers don't use device names for each disk. Instead, these RAID controllers present a virtual disk for each RAID container, using a device node named after the RAID driver. For example, the amr(4) driver presents its virtual disks as */dev/amrd**. A few RAID cards use the cam(4) abstraction layer, so their hard drives do show up as */dev/da** devices.

Hard Disks and Partitions

While we discussed partitioning in Chapter 2, let's consider partitions from a disk device perspective. The first possible ATA disk on our first ATA controller is called */dev/ad0*. Subsequent disks are */dev/ad1*, */dev/ad2*, and so on. Subdivisions of each disk start with this name and add something at the end, like */dev/ad0s1b*. While you might expect a disk to be a monolithic whole, you'll see lots of subdivisions if you look in */dev* for everything that begins with */dev/ad0*.

```
# ls /dev/ad*
/dev/ad0        /dev/ad0s1a     /dev/ad0s1c     /dev/ad0s1e
/dev/ad0s1      /dev/ad0s1b     /dev/ad0s1d     /dev/ad0s1f
```

So, what are all these subdivisions? Think back to when you allocated disk space. If you followed the recommendations in this book, you used the whole disk for FreeBSD. You could have created a second chunk of disk for a second operating system, or even cut the disk into two FreeBSD sections. These sections are called partitions in the Microsoft and Linux worlds, and *slices* in FreeBSD land. The `s1` in the preceding list represents these large partitions, or slices. The drive ad0 has one slice, ad0s1, with further subdivisions marked by letters.

In FreeBSD, a *partition* is a further subdivision within a slice. You created partitions inside the slice during the install. Each partition has a unique device node name created by adding a letter to the slice device node. For example, partitions inside the slice */dev/ad0s1* show up as */dev/ad0s1a*, */dev/ad0s1b*, */dev/ad0s1c*, and so on. Each partition you created—*/usr*, */var*, and so on—is assigned one of these device nodes.

You can assign partition device names almost arbitrarily, with some exceptions. Tradition says that the node ending in *a* (in our example, */dev/ad0s1a*) is the root partition, and the node ending in *b* (*/dev/ad0s1b*) is

the swap space. The *c* label indicates the entire slice, from beginning to end. You can assign *d* through *h* to any partition you like. You can have only eight partitions in one slice, and up to four slices per drive. For example, the device node */dev/ad0s1a* is disk number 0, slice 1, partition 1, and is probably the root filesystem. The device node */dev/ad1s2b* is on disk number 2, and is probably swap space.

If you have non-ATA disks, substitute the appropriate device for */dev/ad*.

ATA DISK NUMBERING

Just because the first possible ATA disk on the system would be */dev/ad0* doesn't mean that you have to have a hard drive */dev/ad0* installed. My laptop's hard drive is */dev/ad4* because it's on a RAID controller, not on the built-in ATA controller. SCSI and SAS hard drives are smarter about this and generally number the first disk with */dev/da0* no matter where they're attached. Removing the kernel option `ATA_STATIC_ID` makes ATA disks start numbering at 0, if you desire.

The Filesystem Table: /etc/fstab

So, how does your system map these device names to partitions? With the filesystem table, */etc/fstab*. In that file, each filesystem appears on a separate line, along with any special options used by mount(8). Here's a sample entry from a filesystem table:

```
/dev/ad4s2a     /     ufs     rw     1     1
```

The first field in each entry gives the device name.

The second field lists the mount point, or the directory where the filesystem is attached. Every partition you can write files to is attached to a mount point such as */usr*, */var*, and so on. A few special partitions, such as swap space, have a mount point of none. You can't write files to swap space—at least, not if you want to use either the file or the swap space!

Next, we have the type of filesystem on this partition. The standard FreeBSD partition is of type `ufs`, or Unix Fast File System. The example below includes `swap` (swap space), `cd9660` (CD), and `nfs` (Network File System). Before you can attach a partition to your directory tree, you must know what sort of filesystem it has. As you might guess, trying to mount a DOS floppy as an FFS filesystem will fail.

The fourth field shows the mount options used on this filesystem. The mount options tell FreeBSD to treat the filesystem in a certain matter. We'll discuss mount options in more detail later in this chapter, but here are a few special ones used only by */etc/fstab*:

`ro` The filesystem is mounted as read-only. Not even root can write to it.

`rw` The filesystem is mounted read-write.

`noauto` FreeBSD won't automatically mount the filesystem, neither at boot nor when using `mount -a`. This option is useful for removable media drives which might not have media in them at boot.

The fifth field tells the dump(8) whether or not this filesystem needs dumping. If set to `0`, dump won't back up the filesystem at all. Otherwise, the number gives the minimum dump level needed to trigger a backup of this filesystem. See Chapter 4 for details.

The last field, `Pass#`, tells the system when to check the filesystem's integrity during the boot process. All of the partitions in the same `Pass#` are checked in parallel with fsck(8). Only the root filesystem has a `Pass#` of `1`, and it is checked first. All other filesystems are set to `2`, which means that FreeBSD mounts them after the root filesystem. Swap and read-only media don't require integrity checking, so are set to `0`.

With this knowledge, let's look at a complete */etc/fstab*.

```
   # Device          Mountpoint   FStype   Options          Dump    Pass#
❶  /dev/ad4s1b       none         swap     sw               0       0
❷  /dev/ad4s2a       /            ufs      rw               1       1
❸  /dev/ad4s1a       /amd64       ufs      rw               2       2
   /dev/ad4s1f       /amd64/usr   ufs      rw               2       2
   /dev/ad4s1d       /amd64/var   ufs      rw               2       2
❹  /dev/ad4s1e       /tmp         ufs      rw               2       2
❺  /dev/ad4s2e       /usr         ufs      rw               2       2
❻  /dev/ad4s2d       /var         ufs      rw               2       2
❼  /dev/ad4s3d       /home        ufs      rw               2       2
❽  /dev/acd0         /cdrom       cd9660   ro,noauto        0       0
❾  data:/mp3         /mp3         nfs      rw,noauto,soft   0       0
```

Our first entry, */dev/ad4s1b* ❶, is swap space. It isn't mounted anywhere; FreeBSD uses swap as secondary memory.

The second entry ❷ is the root partition. Note the device name—while swap is partition `b` on slice 1, the root filesystem is partition `a` on slice 2. The root directory is on a different slice than the swap space!

The third partition is */dev/ad4s1a* ❸. We'd normally expect the root partition to be here, but instead it's mounted as */amd64*. The next two partitions are also on slice 1, but mounted under */amd64*.

Our */tmp* ❹ filesystem is a partition on slice 1, which contains the */amd64* filesystem.

The next entries, */usr* ❺ and */var* ❻, are normal-looking partitions on slice 2.

The next partition, */home* ❼, is on the same disk, slice 3, partition `d`. Where the heck did slice 3 come from?

Our CD drive is mounted on */cdrom* ❽ and is not automatically mounted at boot.

The final entry doesn't start with a device node. This is a Network File System (NFS) entry, and it tells us to mount the partition `mp3` on the machine `data` as */mp3* on the local machine when we specifically request it. We'll talk about NFS later in this chapter.

This filesystem table comes from a dual-boot machine, running FreeBSD/i386 and FreeBSD/amd64. That's why the table doesn't look quite normal. I can access the amd64 filesystem and use the amd64 swap space while running the i386 slice.

What's Mounted Now?

If not all filesystems are mounted automatically at boot, and if the sysadmin can request additional mounts, how can you determine what's mounted right now on the system? Run mount(8) without any options to see a list of all mounted filesystems.

```
# mount
/dev/ad4s2a on / (ufs, local)
devfs on /dev (devfs, local)
/dev/ad4s1e on /tmp (ufs, local, soft-updates)
/dev/ad4s2e on /usr (ufs, local, soft-updates)
/dev/ad4s2d on /var (ufs, local, soft-updates)
/dev/ad4s3d on /usr/home (ufs, local, soft-updates)
```

Here we see that our filesystems are almost all standard UFS partitions. The word `local` means that the partition is on a hard drive attached to this machine. We also see `soft-updates`, a feature of the FreeBSD filesystem we'll discuss later in this chapter. If you're using features such as NFS or SMB to mount partitions, they'll appear here.

mount(8) is a quick way to get the device names for each of your partitions, but it also provides other functions.

Mounting and Unmounting Disks

mount(8)'s main purpose is to mount partitions onto your filesystem. If you've never played with mounting before, boot your FreeBSD machine into the single-user mode (see Chapter 3) and follow along.

In single-user mode FreeBSD has mounted the root partition read-only. The root partition contains just enough of the system to perform basic setup, get core services running, and find the rest of the filesystems. Those other filesystems are not mounted, so their content is inaccessible. Go ahead and look in */usr* on a system in single-user mode; it's empty. FreeBSD hasn't lost the files, it just hasn't mounted the partition with those files on it yet. To do anything interesting in single-user mode, you must mount other filesystems.

Mounting Standard Filesystems

To manually mount a filesystem listed in */etc/fstab*, such as */var* or */usr*, give mount(8) the name of the filesystem you want to mount.

```
# mount /usr
```

This mounts the partition exactly as listed in */etc/fstab*, with all the options specified in that file. If you want to mount all the partitions listed in */etc/fstab*, use mount's `-a` flag.

```
# mount -a
```

Mounting at Nonstandard Locations

Perhaps you need to mount a filesystem at an unusual point. I do this most commonly when installing a new disk. Use the device name and the desired mount point. If my */usr* partition is */dev/ad0s1e*, and I want to mount it on */mnt*, I would run:

```
# mount /dev/ad0s1e /mnt
```

Unmounting a Partition

When you want to disconnect a filesystem from the system, use umount(8) to tell the system to unmount the partition. (Note that the command is `umount`, not `unmount`.)

```
# umount /usr
```

You cannot unmount filesystems that are in use by any program. If you cannot unmount a partition, you're probably accessing it somehow. Even a command prompt in the mounted directory prevents you from unmounting the underlying partition.

How Full Is a Partition?

To get an overview of how much space each partition has left, use df(1). This provides a list of partitions on your system, the amount of space used by each one, and where it's mounted. The annoying thing about df is that it defaults to providing information in 1KB blocks. This was fine when disks were much much smaller, but counting out blocks can make you go cross-eyed today. Fortunately, the `-h` and `-H` flags provide human-readable output. The small `-h` uses base 2 to create a 1,024-byte megabyte, while the large `-H` uses base 10 for a 1,000-byte megabyte. Typically, network administrators and disk manufacturers use base 10, while system administrators use base 2.[1] Either works so long as you know which you've chosen. I'm a network administrator, so you get to suffer through my prejudices in these examples, despite what my tech editor thinks.

```
# df -H
Filesystem     Size    Used   Avail Capacity  Mounted on
/dev/ad4s2a    520M    301M    177M    63%    /
devfs          1.0k    1.0k      0B   100%    /dev
/dev/ad4s1e    520M    2.4M    476M     0%    /tmp
/dev/ad4s2e     11G    4.1G    5.9G    41%    /usr
/dev/ad4s2d    1.0G    322M    632M    34%    /var
/dev/ad4s3d     49G     43G    2.0G    96%    /usr/home
```

[1] This discussion is even less productive than the "Emacs versus vi" argument. And I bet you thought no such argument could exist!

Here we see the partition name, the size of the partition, the amount of space used, the amount of space available, the percent of space used, and the mount point. For example, the home directory on this machine is 96 percent full, but it still has 2GB of disk free. The root partition is only 63 percent full, but it has only 177MB free.

FFS holds back 8 percent of the disk space for on-the-fly optimization. This is used for moving files and reducing fragmentation. You can overfill your disks and see negative disk space remaining. When this happens, disk performance drops dramatically. It is best to keep a little free space on your partitions, so that FFS can continually defragment itself. While you can adjust the filesystem to reduce the amount of reserved space, this negatively impacts performance and is basically unwise. See tunefs(8) if you really want to try it.

The obvious question is, "What is taking up all that space?" If your systems are like mine, disk usage somehow keeps growing for no apparent reason. You can identify individual large files with `ls -l`, but recursively doing this on every directory in the system is impractical.

$BLOCKSIZE

Many disk tools show sizes in blocks of 512 bytes, or one-half KB. If you set the environment variable `$BLOCKSIZE` to k, df(1) and many other programs display file sizes in blocks of 1KB, which is much more useful. Others like a setting of 1M, for sizes in megabytes.

du(1) displays disk usage in a single directory. Its initial output is intimidating and can scare off inexperienced users. Here, we use du(1) to find out what's taking up all the space in my home directory:

```
# cd $HOME
# du
1       ./bin/RCS
21459   ./bin/wp/shbin10
53202   ./bin/wp
53336   ./bin
5       ./.kde/share/applnk/staroffice_52
6       ./.kde/share/applnk
...
```

This goes on and on, displaying every subdirectory and giving its size in blocks. The total of each subdirectory is given—for example, the contents of *$HOME/bin* totals 53,336 blocks, or roughly 53MB. I could sit and let du(1) list every directory and subdirectory, but then I'd have to dig through much more information than I really want to. And blocks aren't that convenient a measurement, especially not when they're printed left-justified.

Let's clean this up. First, du(1) supports an `-h` flag much like df. Also, I don't need to see the recursive contents of each subdirectory. We can control the number of directories we display with du's `-d` flag. This flag takes one

argument, the number of directories you want to explicitly list. For example, `-dO` goes one directory deep and gives a simple subtotal of the files in a directory.

```
# du -h -dO $HOME
 37G    /home/mwlucas
```

I have 37 gigs of data in my home directory? Let's look a layer deeper and identify the biggest subdirectory.

```
# du -h -d1
 38K    ./bin
 56M    ./mibs
...
 34G    ./mp3
...
```

Apparently I must look elsewhere for storage space, as the data in my home directory is too important to delete.

If you're not too attached to the `-h` flag, you can use sort(1) to find the largest directory with a command like `du -kxd 1 | sort -n`.

The Fast File System

FreeBSD's filesystem, the Fast File System (FFS), is a direct descendant of the filesystem shipped with BSD 4.4. One of the original FFS authors still develops the FreeBSD filesystem and has added many nifty features in recent years. FFS is sometimes called UFS (for Unix File System), and many system utilities still call FFS partitions UFS. FreeBSD is not the only operating system to still use the 4.4 BSD filesystem or a descendant thereof. If a Unix vendor doesn't specifically tout its "improved and advanced" filesystem, it is almost certainly running a derivative of FFS.

FFS is designed to be fast and reliable, and to handle the most common situations as effectively as possible while still supporting unusual situations reliably. FreeBSD ships with FFS configured to be as widely useful as possible on relatively modern hardware, but you can choose to optimize a particular filesystem for trillions of small files or a half-dozen 30GB files if you must. You don't have to know much about FFS's internals, but you do need to understand blocks, fragments, and inodes, if nothing else.

Blocks are segments of disk that contain data. FreeBSD defaults to 16KB blocks. Not all files are even multiples of 16KB, so FFS uses *fragments* to store leftovers. The standard is one-eighth of the block size, or 2KB. For example, a 20KB file would fill one block and two fragments. *Inodes* are index nodes, special blocks that contain basic data such as a file's size, permissions, and the list of blocks that this file uses. Collectively, the data in an inode is known as *metadata*, or data about data. This arrangement isn't unique to FFS; other filesystems such as NTFS use data blocks and index nodes as well. The indexing system used by each filesystem is largely unique, however.

Each filesystem has a certain number of inodes, proportional to the filesystem size. A modern disk probably has hundreds of thousands of inodes on each partition, which is sufficient to support hundreds of thousands of files. If you have a truly large number of very tiny files, however, you might need to rebuild your filesystem to support additional inodes. Use `df -i` to see how many inodes remain free on your filesystem. If you must rebuild your filesystem to increase the number of inodes, see Chapter 18.

Vnodes

Inodes and blocks worked wonderfully in Unix's early days, when hard drives were permanently attached to machines. As time passed, however, swapping disks between different machines and even different operating systems became common. CDs, with their unique read-only filesystem, became popular, floppy disks slowly converged on the FAT32 filesystem as a standard, and other Unix-like systems developed their own variant filesystems. Since BSD needed to speak to all those different systems, another layer of abstraction was needed.

That abstraction was the virtual node, or vnode. Users never manipulate vnodes directly, but you'll see references to them throughout the system documentation. The *vnode* is a translator between the kernel and whatever specific filesystem type you've mounted. Every tool that reads and writes to disks actually does so through vnodes, which map the data to the appropriate filesystem for the underlying media. When you write a file to an FFS filesystem, the kernel addresses data to a vnode which, in turn, is mapped to an inode. When you write a file to a FAT32 filesystem, the kernel addresses data to a vnode mapped to a point in the FAT32 filesystem. You use inodes only when dealing with FFS filesystems, but you'll use vnodes when dealing with any filesystem.

FFS Mount Types

Unlike Windows filesystems, FreeBSD treats FFS partitions differently depending on how they're mounted. The manner in which a partition is mounted is called the *mount type*. If you're mounting a partition manually, you can specify mount options on the command line, but if you're using */etc/fstab* to mount it, you must specify any choice in the `Options` column.

Use `-o` *`mounttype`* to specify a mount type on the command line, or specify the option in */etc/fstab* in the `Options` column.

Read-Only Mounts

If you want to look at the contents of a disk but not write to it, mount the partition read-only. In most cases, this is the safest and the most useless way to mount a disk, because you cannot alter the data on the disk or write any new data.

Many system administrators mount the root partition, and perhaps even */usr*, as read-only to minimize potential system damage from a loss of power or software problems. Even if you lose the physical hard drive due to a power surge or other hardware failure, the data on the platters remains intact. That's the advantage of read-only mounts. The disadvantage is that system maintenance becomes much more difficult because you cannot write to read-only disks!

Read-only mounts are especially valuable when your computer is damaged. While FreeBSD won't let you perform a standard read-write mount on a damaged or dirty filesystem, it can perform a read-only mount most of the time. This gives you a chance to recover data from a dying system.

To mount a filesystem read-only, use one of the options `rdonly` or `ro`. Both work identically.

Synchronous Mounts

Synchronous (or *sync*) *mounts* are the old-fashioned way of mounting filesystems. When you write to a synchronously mounted disk, the kernel waits to see whether the write is actually completed before informing the program. If the write did not complete successfully, the program can choose to act accordingly.

Synchronous mounts provide the greatest data integrity in the case of a crash, but they are also slow. Admittedly, "slow" is relative today, when even a cheap disk outperforms what was the high end several years ago. Consider using synchronous mounting when you wish to be truly pedantic on data integrity, but in almost all cases it's an overkill.

To mount a partition synchronously, use the option `sync`.

Asynchronous Mounts

While *asynchronous mounts* are pretty much supplanted by soft updates, you'll still hear about them. For faster data access at higher risk, mount your partitions asynchronously. When a disk is asynchronously mounted, the kernel writes data to the disk and tells the writing program that the write succeeded without waiting for the disk to confirm that the data was actually written. Asynchronous mounting is fine on disposable machines, but don't use it with important data. The performance difference between asynchronous mounts and noasync with soft updates is very small.

To mount a partition asynchronously, use the option `async`.

Noasync Mounts

Finally, we have a method that combines sync and async mounts, called *noasync*. This is FreeBSD's default. When using noasync mounts, data that affects inodes is written to the disk synchronously, while actual data is handled asynchronously. Combined with soft updates (see later in this chapter), a noasync mount creates a very robust filesystem.

Noasync mounts are the default, and you don't need to specify anything.

FFS Mount Options

FreeBSD supports several mount options in addition to the mount types. These options change the behavior of mounted filesystems.

noatime

Every file in FFS includes an access-time stamp, called the *atime*, which records when the file was last accessed. If you have a large number of files and don't need this data, you can mount the disk noatime so that FFS does not update this timestamp. This is most useful for flash media or disks that suffer from heavy load, such as Usenet news spool drives.

noexec

The `noexec` mount option prevents any binaries from being executed on this partition. Mounting */home* noexec can help prevent users from running their own programs, but for it to be effective, be sure to also mount */tmp*, */var/tmp*, and anywhere else users can write their own files noexec as well. Also note that a noexec mount doesn't prevent a user from running a shell script, which is just a set of instructions for a program elsewhere in the system. Another common use for a noexec mount is when you have on your server binaries for a different operating system or a different hardware architecture and you don't want anyone to execute them.

nosuid

The `nosuid` option prevents setuid programs from running on your system. Setuid programs allow users to run programs as if they're another user. For example, programs such as login(1) must perform actions as root but must be run by regular users. Setuid programs obviously must be written carefully so that intruders cannot exploit them to get unauthorized access to your system. Many system administrators habitually disable all unneeded setuid programs. You can use nosuid to disable setuid programs on a partition, but script wrappers like suidperl get around this.

nosymfollow

The `nosymfollow` option disables symlinks, or aliases to files. *Symlinks* are mainly used to create aliases to files that reside on other partitions. To create an alias to another file on the same partition, use a regular link instead. See ln(1) for a discussion of links.

Aliases to directories are always symlinks; you cannot use a hard link for that.

Soft Updates and Journaling with FFS

Soft updates is a technology to organize and arrange disk writes so that file-system metadata remains consistent at all times, giving nearly the performance of an async mount with the reliability of a sync mount. While that doesn't mean that all data will be safely written to disk—a power failure at the

wrong moment can still lose data—soft updates prevent many problems. If the system has a failure, soft updates can run its filesystem checks in the background while the system runs. In my opinion, soft updates are suitable for partitions of less than 80GB or so.

As of FreeBSD 7.0, FFS also supports *journaling*. A journaling filesystem records all changes made to the filesystem on a separate part of the disk, so that if the system shuts down unexpectedly, data changes can be recovered from the journal automatically upon restart. This eliminates the need to run fsck(8). Each journaled filesystem uses about 1GB for the journal, which means that journaling is wasteful on small filesystems. However, FreeBSD's journaling differs from most journaled filesystems in that the journal is maintained in a disk layer beneath the filesystem rather than in the filesystem itself. This journaling is brand new in FreeBSD and is still considered somewhat experimental.

For example, as of this writing, I have a logging host. Most of the partitions are less than 10GB, but */var* is almost two terabytes. I journal */var*, but use soft updates on the other partitions. Between background fsck and journaling, system recovery after an unexpected power outage is fast and painless.

We'll talk more about journaling and gjournal in Chapter 18.

Write Caching

FFS works best with SCSI and SAS drives due to the robustness of the SCSI architecture. FFS also works as well as the ATA architecture allows, with one critical exception: Many modern IDE drives support *write caching*.

Write caching IDE drives have a small onboard chip. This chip records data that needs to be written to the drive. This can be tricky for soft updates and for journaling, because these technologies expect honest hard drives—when the hard drive reports the data is written to disk, the soft updates mechanism expects the data to actually be on that platter. But IDE write caching reports success as soon as the data is stored in the drive's cache, not when it has been written to disk. It might be a second or more until that data is actually safely stored.

While this doesn't pose a big risk if it happens occasionally, write caching occurs continuously on a server. Therefore, if you care about your data, disable write caching by adding the following to */boot/loader.conf*:

```
hw.ata.wc=0
```

Disabling write caching slows the IDE drive, but eliminates the risk. I use write caching on desktop and laptop systems, where data is not being continuously written to the disk, but on servers it's a bad idea to leave caching on. You can either tell your management that the ATA-based server is throttled and you need more hardware, or you can tell them that you're missing data because you overstressed the hardware. Personally, I prefer the former.

Snapshots

FFS with soft updates can take an image of a disk at a moment in time, or a *snapshot*. You'll find these snapshots in the *.snapshot* directory in the root of each partition.

While you don't perform system administration on snapshots, various tools take advantage of snapshots. For example, dump(8) backs up a snapshot of each filesystem rather than the live filesystem so that you have an internally consistent backup. Background fsck (explained later in this chapter) uses snapshots to do its work. You can mount a snapshot and get a picture of your filesystem at a particular instant. While this is interesting, most system administrators don't find it terribly useful. Still, you'll see references to snapshots throughout working with FreeBSD.

Dirty Disks

No, disks don't get muddy with use (although dust on a platter will quickly damage it, and adding water won't help). A dirty FFS partition is one that's in a kind of limbo; the operating system has asked for information to be written to the disk, but the data is not yet completely written out. Part of the data block might have been written, the inode might have been edited but the data not written out, or any combination of the two. If the power goes out while your disk is dirty, the system reboots with *unclean disks*.

FreeBSD refuses to mount unclean disks read-write; you can write them read-only, but that's not suitable for normal operation. You must clean your disk.

fsck(8)

FreeBSD includes a filesystem integrity checking tool, fsck(8). When a rebooting system finds a dirty disk, it automatically checks the filesystem and tries to clean everything. You have already lost any data not written to disk, but fsck(8) verifies that every file is attached to the proper inodes and in the correct directory. If successful, everything is right where you left it—except for that unwritten data, of course!

Failed Automatic fscks Runs

Occasionally this automated fsck-on-reboot fails to work. When you check the console, you'll be looking at a single-user mode prompt and a request to run fsck(8) manually. At this point, you have a simple choice: run fsck or not. If you enter `fsck` at the command prompt, fsck(8) verifies every block and inode on the disk. It finds any blocks that have become disassociated from their inodes and guesses how they fit together and how they should be attached. fsck(8) might not be able to identify which directory these files belong in, however.

fsck asks if you want to perform these reattachments. If you answer `n`, it deletes the damaged files. If you answer `y`, it adds the lost file to a *lost+found* directory in the root of the partition, with a number as a filename. For example, the *lost+found* directory on your */usr* partition is */usr/lost+found*. If there are

only a few files, you can identify them manually; if you have many files and are looking for particular ones, tools such as file(1) and grep(1) can help you identify them by content.

Turning Off the fsck Prompt

If your server was in the middle of a very busy operation when it became dirty, you could end up with many disassociated files. Instead of spending an hour at the console typing `y` over and over again to tell fsck to reassemble these files, type `fsck -y` at the single-user prompt. This makes fsck(8) believe that you're answering yes to every question.

> **DANGER!**
>
> Running `fsck -y` is not guaranteed safe. At times, when running experimental filesystems on -current or when doing other daft things, I've had the entire contents of a partition migrate to */usr/lost+found* and */var/lost+found* thanks to `fsck -y`. Recovery becomes difficult at that point. Having said that, in a production system running FreeBSD-stable with a standard UFS filesystem, I've never had a problem.

You can set your system to automatically try `fsck -y` on boot. I don't recommend this, however, because if there's the faintest chance my filesystem will wind up in digital nirvana I want to know about it. I want to type the offending command myself and feel the trepidation of watching fsck(8) churn my hard drives. Besides, it's always unpleasant to discover that your system is trashed without having the faintest clue of how it got that way. If you're braver than I, you can set `fsck_y_enable="YES"` in *rc.conf*.

Avoiding fsck -y

What options do you have if you don't want to use `fsck -y`? Well, fsdb(8) and clri(8) allow you to debug the filesystem and redirect files to their proper locations. You can restore files to their correct directories and names. This is difficult,[2] however, and is recommended only for Secret Ninja Filesystem Masters.

Background fsck

Background fsck gives FFS some of the benefits of a journaled filesystem without using all the disk required by journaling. When FreeBSD sees that a background fsck is in process after a reboot, it mounts the dirty disk read-write. fsck(8) runs in the background while the server is running, identifying loose bits of files and tidying them up behind the scenes.

A background fsck actually has two major stages. When FreeBSD finds dirty disks during the initial boot process, it runs a preliminary fsck(8) assessment of the disks. fsck(8) decides if the damage can be repaired while the

[2] In the first edition of this book, I said using fsdb(8) and clri(8) was like climbing Mount Everest in sandals and shorts. Since writing that, I've tried them more than once and discovered that I was wrong. You don't get the shorts.

system is running, or if a full single-user mode fsck run is required. Most frequently, fsck thinks it can proceed and lets the system boot. After the system reaches single-user mode the background fsck runs at a low priority, checking the partitions one by one. The results of the fsck process appear in */var/log/messages*.

You can expect performance of any applications requiring disk activity to be lousy during a background fsck. fsck(8) occupies a large portion of the disk's possible activity. While your system might be slow, it will at least be up.

You *must* check */var/log/messages* for errors after a background fsck. The preliminary fsck assessment can make an error, and perhaps a full single-user mode's fsck on a partition really is required. If you find such a message, schedule downtime within a few hours to correct the problem. While inconvenient, having the system down for a scheduled period is better than the unscheduled downtime caused by a power outage and the resulting single-user mode's `fsck -y`.

Forcing Read-Write Mounts on Dirty Disks

If you really want to force FreeBSD to mount a dirty disk read-write without using a background fsck, you can. You will not like the results. At all. But, as it's described in mount(8), some reader will think it's a good idea unless they know why. Use the `-w` (read-write) and `-f` (force) flags to mount(8).

Mounting a dirty partition read-write corrupts data. Note the absence of words like *might* and *could* from that sentence. Also note the absence of words like *recoverable*. Mounting a dirty filesystem may panic your computer. It might destroy all remaining data on the partition or even shred the underlying filesystem. Forcing a read-write mount of a dirty filesystem is bad juju. Don't do it.

FFS Syncer at Shutdown

When you shut down your FreeBSD system the kernel synchronizes all its data to the hard drive, marks the disks clean, and shuts down. This is done by a kernel process called the *syncer*. During a system shutdown, the syncer reports on its progress in synchronizing the hard drive.

You'll see odd things from the syncer during shutdown. The syncer doesn't actually go down the list of inodes and blocks that need updating on the hard drive; it walks the list of vnodes that need synchronizing. Thanks to soft updates, writing one vnode to disk can generate another dirty vnode that needs updating. You can see the number of buffers being written to disk rapidly drop from a high value to a low value, and perhaps bounce between zero and a low number once or twice as the system really synchronizes the hard drive.

THE REAL DETAILS ON FFS

If you really want to know more about FFS, you can download a large diagram of the kernel internals of FFS at *http://phk.freebsd.dk/misc/ufs.pdf*. You'll need a large engineering or architectural printer, or 18 sheets of regular paper and a lot of clear tape.

Background fsck, fsck -y, Foreground fsck, Oy Vey!

All these different fsck(8) problems and situations can occur, but when does FreeBSD use each command? FreeBSD uses the following conditions to decide when and how to fsck(8) on a filesystem:

- If the filesystem is clean, it is mounted without fsck(8).
- If a filesystem without soft updates is dirty at boot, FreeBSD runs fsck on it. If the filesystem damage is severe, FreeBSD stops the fsck and requests your intervention. You can either run `fsck -y` or manually check each reconnection.
- If a filesystem with soft updates is dirty at boot, FreeBSD performs a very basic fsck(8) check. If the damage is mild, FreeBSD uses a background fsck(8) in multi-user mode. If the damage is severe, FreeBSD interrupts the boot and requests your intervention with either `fsck -y` or approval or rejection of each filesystem problem.
- If a journaled filesystem is dirty at boot, FreeBSD recovers the data from the journal and continues the boot. A journaled filesystem rarely needs fsck(8).

Using Foreign Filesystems

For our purposes, any partition or disk that isn't FFS is a *foreign filesystem*. FreeBSD includes extensive support for foreign filesystems, with the caveat that only those functions supported by the filesystem work. Microsoft's FAT32 doesn't support filesystem permissions, for example, and Linux filesystems don't support BSD-style file flags.

Each foreign filesystem needs support in the FreeBSD kernel. To make your life a little easier, mount(8) automatically loads the proper kernel modules on demand.

To mount any foreign filesystem, you need the same information needed to mount an FFS filesystem: a device name and a mount point. You'll also need the type of filesystem, although you might figure that out by trial and error. For example, FreeBSD provides a */cdrom* mount point for CDs. The first IDE CD on your system is */dev/acd0*. CDs use the ISO 9660 filesystem, and FreeBSD mounts them with `mount -t cd9660(8)`. Here we mount our CD on */cdrom*:

```
# mount -t cd9660 /dev/acd0 /cdrom
```

You can now go to the */cdrom* directory and view the contents. Simple enough, eh?

If you try to mount a disk using the wrong mount command for its filesystem, you'll get an error. For example, any floppy disk in my house has either a FAT32 or an FFS filesystem on it; */dev/fd0* is the proper device node for floppy disks, and */media* is their standard mount point.

```
# mount /dev/fd0 /media
mount: /dev/fd0 on /media: incorrect super block
```

This floppy has a FAT32 filesystem. Had I tried `mount -t msdosfs` first, it would have worked.

You can unmount any filesystem with umount(8):

```
# umount /cdrom
```

umount(8) doesn't care about the filesystem type. It just disconnects the disk partition.

Supported Foreign Filesystems

Here are some of the most commonly used foreign filesystems, along with a brief description of each and the appropriate mount command.

FAT (MS-DOS)

FreeBSD includes extensive support for FAT, the DOS/Windows 9*x* File Allocation Table filesystem, commonly used on removable media and some dual-boot systems. This support covers the FAT12, FAT16, and FAT32 varieties. You *can* format a floppy disk with an FFS filesystem, however, so do not blindly assume that all floppy disks are FAT-formatted. As the most common use for a floppy these days is transferring files between machines, however, most are FAT32. The mount type is `msdosfs` (`mount -t msdosfs`).

If you handle a lot of FAT32 disks, investigate */usr/ports/tools/mtools*, a collection of programs for working with FAT filesystems that offer greater flexibility than the default FreeBSD tools.

ISO 9660

ISO 9660 is the standard CD filesystem. FreeBSD supports reading CDs and writing them as well if you have a CD burner. Just about every CD you encounter is formatted with ISO 9660. The mount command is `mount -t cd9660`.

cdrtools, in */usr/ports/sysutils/cdrtools*, contains many helpful tools for working with CD images, including tools that build an ISO image from files on disk.

UDF

UDF, or Universal Data Format, is a replacement for ISO 9660. You'll find UDF on some DVDs and on flash/USB devices larger than the 32GB supported under FAT32. As the capacity of removable media increases, you'll see more and more UDF filesystems. The mount command is `mount -t udf`.

NTFS

The Windows NT/2000/XP filesystem, NTFS, is tightly integrated with Microsoft's proprietary kernel. To write to an NTFS partition, you must have extensive knowledge of how the filesystem works. Since Microsoft does not

make that information available, FreeBSD can safely mount NTFS partitions read-only but has limited read-write functionality. The mount command is `mount -t ntfs`.

I find NTFS mounts most useful when migrating from Windows systems to FreeBSD systems; just remove the hard drive from the old Windows machine, mount it in the FreeBSD machine, and copy your data from one to the other. NTFS support is also useful for dual-boot laptops.

As the NTFS filesystem has a closed specification and contains data encoded in a proprietary manner, FreeBSD's NTFS read support is not guaranteed to work.

ext2fs and ext3fs

The standard Linux filesystems, ext2fs and ext3fs, support many of the same features as the FreeBSD filesystem and can be safely written to and read from without any problems. Like the NTFS mounts, mounting Linux filesystems is most useful for disaster recovery or dual-boot systems. Despite the name, `mount -t ext2fs` supports mounting both ext2fs and ext3fs.

Linux filesystem users might find the tools in */usr/ports/sysutils/e2fsprogs* useful. They let you fsck(8) and assess Linux filesystems, among other things.

ReiserFS

ReiserFS is a filesystem with a small percentage of devotees amidst Linux users. FreeBSD supports read-only ReiserFS partitions. Support for mounting ReiserFS is implemented directly in mount(8); just run `mount -t reiserfs` *`partition mountpoint`*.

XFS

FreeBSD supports reading from SGI's XFS partitions, but writing to XFS is available on an experimental basis as of this writing. XFS is the oldest journaling filesystem in existence and has a well-debugged codebase. XFS is licensed under the GPL, however, which makes it a poor candidate for inclusion in the FreeBSD base system. If you're interested in journaling, please check out gjournal in Chapter 18.

Programs for formatting, mounting, and managing XFS partitions can be found in */usr/ports/sysutils/xfsprogs*.

ZFS

As of 7.0, FreeBSD includes experimental support for ZFS, ported from OpenSolaris. While the installer doesn't support ZFS, if you need the advanced ZFS features you can use them. ZFS's license is not suitable for making it the primary FreeBSD filesystem, but its high-end features can be very useful for certain applications on multiterabyte filesystems and on 64-bit systems. ZFS on 32-bit systems suffers from memory problems, but ZFS has a good reputation on 64-bit systems.

Permissions and Foreign Filesystems

The method used to mount a filesystem, and the person who mounts it, control the permissions of the mounted filesystem. For example, ext3fs and XFS store permissions in the filesystem and map them to UIDs. Since their permissions behave much like FFS, and all the permissions information needed is available within the filesystem and the kernel, FreeBSD respects the permissions on these filesystem.

NTFS has its own permission system. Since that system bears only a coincidental resemblance to that used by Unix-like systems, permissions are discarded when a NTFS partition is mounted on a FreeBSD system, and it's treated much like a CD or a floppy disk.

By default, only root can mount filesystems, and root owns all non-Unix filesystems. If that's not your preference, you can use the `-u` and `-g` flags to set the user ID and group ID of the owner when you're mounting a FAT32, NTFS, or ISO 9660 filesystem. For example, if you're mounting a FAT32 USB device for the user cstrzelc and want her to be able to edit the contents, use this command:

```
# mount -t msdosfs -u cstrzelc -g cstrzelc /dev/da0 /mnt
```

The user cstrzelc now owns the files on the device.

You might get sick of mounting media for your users, especially in a facility with dozens of machines. To let a user mount a filesystem, set the sysctl `vfs.usermount` to `1`. The user can then mount any filesystem she wants on any mount point she owns. While cstrzelc couldn't mount the removable device on */mnt*, she could mount it on */home/cstrzelc/mnt*.

Removable Media Filesystems

Removable media has boomed in the last few years. While once we only had to worry about floppy disks, we now have CDs and USB devices. You must be able to manage any removable media that might wander in through the door of your data center. We'll discuss dealing with filesystems on floppy disks, USB devices, and CDs.

I recommend not plugging removable media willy-nilly into your production servers, for security reasons if nothing else. Who knows what's actually on that vendor's USB device? I prefer to mount those devices on my workstation, examine the contents, and then copy the desired data over to the FreeBSD machine. Removable media is just too easy for certain applications, however, and of course the rules change when it's my personal USB device.

Formatting FAT32 Media

Both floppy disks and USB media use the FAT32 filesystem. USB media uses the FAT32 filesystem, but usually comes preformatted. As USB media has a limited number of reads and writes, do not reformat it capriciously. That

leaves floppies, which frequently need reformatting if used heavily. What most Windows users think of as "formatting a floppy" is actually a multistage process that includes formatting the disk, giving it a disk label, and creating a filesystem. You must perform all of these operations to use a floppy in FreeBSD.

> **NONSTANDARD FLOPPIES**
>
> We'll assume that you have a standard 1.44MB floppy disk, which has been the standard on *x*86 hardware for almost two decades. If you have a disk drive of 800KB or another uncommon size, you'll have to modify this process somewhat, but the general steps are the same.

Low-Level Formatting

First, perform a low-level format with fdformat(1). This program only requires two arguments: the floppy's size and the device name.

```
# fdformat -f 1440 /dev/fd0
Format 1440K floppy '/dev/fd0.1440'? (y/n): y
```

When you type y, fdformat(1) runs a low-level format to prepare the disk to receive a filesystem. Low-level formatting is the slowest part of making a floppy usable. Windows performs this sort of formatting when you request a complete format.

Creating an FFS Filesystem

After the low-level format, if you want to use FFS on your floppy, label the disk with disklabel(8). This writes basic identification information to the floppy, sets partition information, and can even mark a disk as bootable. Marking a disk as bootable doesn't actually put any programs on the disk; it simply puts a marker on the disk so the hardware BIOS identifies this disk as bootable. If you need an actual FreeBSD boot floppy, just copy the provided disk image from the installation media. Here, we'll install a plain disklabel without any special characteristics:

```
# disklabel -r -w /dev/fd0 fd1440
```

The `-r` option in this example tells disklabel to access the raw disk, because there is no filesystem on it yet. The `-w` option says to write to the disk. We're writing to */dev/fd0* and installing a standard 1.44MB floppy disk label. You can find a full list of floppy disk labels in */etc/disktab*, as well as labels for many other types of drives, as discussed in Chapter 10.

Finally, newfs(8) the disk to create a filesystem.

```
# newfs /dev/fd0
```

You'll see a few lines of newfs output and then get a command prompt back.

Note that using FFS on a floppy wastes space and does not provide the protection many people expect it would. While permission information is stored on the floppy, the system owner can override those privileges easily. Also, remember that FFS reserves 8 percent of disk space for its own overhead and bookkeeping. Do you really need a 1.32MB floppy? FFS is nifty, but FAT32 is a better choice for floppy disks.

Creating a FAT32 Filesystem

To swap data between a wide range of hardware, use the FAT32 filesystem on your floppy. While you still need to low-level format the floppy as discussed earlier, you don't need to disklabel it. Just use newfs_msdos(8):

```
# newfs_msdos /dev/fd0
```

You'll get a couple lines of output, and your floppy is done.

Using Removable Media

Handling removable media is just like working with fixed media such as hard drives. You must know the filesystem on the device, the device node where the device appears, and assign a mount point.

CDs use the ISO 9660 filesystem, while DVDs use either a UDF or a combination of ISO 9660 and UDF. USB devices and floppy disks are usually FAT32. Very new USB devices might be UDF. I expect UDF to become more common, especially as flash devices become larger.

The device node varies with the device type. IDE CDs are */dev/acd0*, while SCSI CDs appear at */dev/cd0*. Floppy disks are at */dev/fd0*. USB devices appear as the next available unit of */dev/da*. When you insert a USB device, a message giving the device node and type appears on the console and in */var/log/messages*.

Finally, you need a mount point. By default, FreeBSD includes a */cdrom* mount point for CD and DVD media. You'll also find a */media* mount point for general *removable media* mounts. You can create additional mount points as you like—they're just directories. For miscellaneous short-term mounts, FreeBSD offers */mnt*.

So, to mount your FAT32 USB device */dev/da0* on */media*, run:

```
# mount -t msdosfs /dev/da0 /media
```

Occasionally you'll find a USB device that has an actual slice table on it, and those devices will insist you mount */dev/da0s1* rather than */dev/da0*. This depends on how the device is formatted, not on anything in FreeBSD.

One annoyance with CD drives is that SCSI CD and ATA CD drives offer different interfaces to software. Much software is written to only use the SCSI interface, as SCSI is generally considered more reliable. If you encounter such a piece of software, check out atapicam(4) kernel module which provides a SCSI emulation layer to your ATA CD devices.

Ejecting Removable Media

To disconnect removable media from your FreeBSD system, you must unmount the filesystem. Your CD tray won't open until you unmount the CD, and the floppy drive won't eject your disk. You can probably forcibly remove a USB dongle, but doing so while the filesystem is mounted panics the server and might damage data on the device. Use umount(8) just as you would for any other filesystem:

```
# umount /cdrom
```

Removable Media and /etc/fstab

You can update */etc/fstab* with entries for removable media to make system maintenance a little easier. If a removable filesystem has an entry in */etc/fstab*, you can drop both the filesystem and the device name when mounting it. This means that you don't have to remember the exact device name or filesystem to mount the device. You probably already have an */etc/fstab*'s entry for your CD device.

```
/dev/acd0               /cdrom          cd9660  ro,noauto       0       0
```

While I'm sure you've already memorized the meaning of every column in */etc/fstab*, we'll remind you that entry means "mount */dev/acd0* on */cdrom*, using the ISO 9660 filesystem, read-only, and do not mount it automatically at boot." Using this as a template, make similar entries for USB devices and floppy disks.

```
/dev/fd0                /mnt            msdosfs  rw,noauto      0       0
❶/dev/da0               /medi           msdosfs  rw,❷noauto     0       0
```

FreeBSD doesn't provide these by default, but I find having them to be much easier on systems where I use removable media regularly. Confirm that your next available da device is */dev/da0* ❶, as trying to mount a hard drive that's already mounted won't work.

When listing removable media in */etc/fstab*, be sure to include the `noauto` ❷ flag. Otherwise, whenever you don't have the removable media in place, your boot will stop in single-user mode because a filesystem is missing.

Other FreeBSD Filesystems

In addition to FFS and non-FreeBSD filesystems, FreeBSD supports several special-purpose filesystems. Some of these are interesting but not generally useful, such as mount_umapfs(8), while others are very useful or even required in certain circumstances. We'll talk about memory filesystems, a popular optimization for certain environments; building filesystems on files; as well as the procfs and fdescfs filesystems occasionally required by software packages.

Memory Filesystems

In addition to using disks for partitions, FreeBSD lets you create partitions from files, from pure RAM, and from a combination of the two. One of the most popular uses of this is for *memory filesystems*, or *memory disks*. Reading and writing files to and from memory is much faster than accessing files on disk, which makes a memory-backed filesystem a huge optimization for certain applications. As with everything else in memory, however, you lose the contents of your memory disk at system shutdown.

Memory Disk /tmp

The most common place a memory filesystem is used is for */tmp*. This is so common that FreeBSD includes specific *rc.conf* support for it. Look for the following *rc.conf* variables:

```
❶ tmpmfs="YES"
❷ tmpsize="20m"
❸ tmpmfs_flags="-S"
```

By setting `tmpmfs` ❶ to `YES`, you're telling FreeBSD to automatically create a memory-based */tmp* filesystem at boot. Specify the size of your new */tmp* with the `tmpsize` ❷ variable. We'll discuss the flags ❸ a little later in this section.

If this is all you want to use a memory filesystem for, you're done. To create and use special memory devices with mdmfs(8), read on.

Memory Disk Types

Memory disks come in three types: malloc-backed, swap-backed, and vnode-backed.

Malloc-backed disks are pure memory. Even if your system runs short on memory, FreeBSD won't swap out the malloc-backed disk. Using a large malloc-backed disk is a great way to make your system run out of free memory and panic. This is most useful for embedded devices with no swap, as we'll see in Chapter 20.

Swap-backed disks are mostly memory, but they also access the system swap partition. If the system runs out of memory, it moves the least recently used parts of memory to swap as discussed in Chapter 19. Swap-backed disks are the safest way to use a memory filesystem.

Vnode-backed disks are files on disk. While you can use a file as backing for your memory disk, this is really only useful for filesystem developers who want to perform tests or for mounting a disk image.

Your application dictates the type of memory disk you need. If your system has no swap space and a read-only filesystem, a malloc-backed disk is your only choice. If you're running a workstation or a server and want a piece of fast filesystem for scratch space, you want a swap-backed disk. You only need a vnode-backed memory disk to mount a disk image.

Once you know what you want to do, use mdmfs(8) to perform the action.

Creating and Mounting Memory Disks

The mdmfs(8) utility is a handy front end for several programs such as mdconfig(8) and newfs(8). It handles the drudgery of configuring devices and creating filesystems on those devices, and makes creating memory disks as easy as possible. You only need to know the size of the disk you want to use, the type of the memory disk, and the mount point.

Swap-backed memory disks are the default. Just tell mdmfs(8) the size of the disk and the mount point. Here, we create an 8MB swap-backed memory disk on */home/mwlucas/test*:

```
# mdmfs -s 8m md /home/mwlucas/test
```

The `-s` flag gives the size of the disk. If you run mount(8) without any arguments, you'll see that you now have the memory disk device */dev/md0* mounted on that directory.

To create and mount a malloc-backed disk, add the `-M` flag.

To mount a vnode-backed memory disk, use the `-F` flag and the path to the image file.

```
# mdmfs -F diskimage.file md /mnt
```

The `md` entry we've been using all along here means, "I don't care what device name I get, just give me the next free one." You can also specify a particular device name if you like. Here, I used */dev/md9* as the memory disk device:

```
# mdmfs -F diskimage.file md9 /mnt
```

Memory Disk Headaches

Memory disks sound too good to be true for high-performance environments. They do have constraints that you must understand before you scurry around and implement them everywhere.

The big issue is that system shutdown erases memory disks. While this might not seem like a problem, I've been surprised more than once by losing a file on a filesystem I had forgotten was a memory disk.

Also, malloc-backed memory disks *never* release memory. When you write a file to a memory disk and then erase the file, the file still takes up memory. With a swap-backed memory disk FreeBSD eventually writes that file to swap as memory pressure rises. A memory disk on a long-running system will eventually exhaust your system's RAM.

The one way you can free this used memory is to unmount the memory disk, destroy the disk device, and re-create the memory disk. I find that performing this task on a monthly basis is more than sufficient, even on a busy server.

A couple months before release, FreeBSD 7.0 added a tmpfs(5) memory filesystem that is supposed to release memory back to the system. It's brand new to FreeBSD, however, and considered an experimental feature. If you're

reading this a few releases into the future I'd recommend you check out tmpfs(5), but if you're installing 7.0 or 7.1, I'd let someone else discover the bugs for you.

Memory Disk Shutdown

To remove a memory disk you must unmount the partition and destroy the disk device. Destroying the disk device frees the memory used by the device, which is useful when your system is heavily loaded. To find the disk device, run mount(8) and find your memory disk partition. Somewhere in the output, you'll find a line like this:

```
/dev/md41 on /mnt (ufs, local, soft-updates)
```

Here, we see memory disk */dev/md41* mounted on */mnt*. Let's unmount it and destroy it.

```
# ❶umount /mnt
# mdconfig ❷-d ❸-u 41
```

Unmounting with umount ❶ is done exactly as with other filesystems. The mdconfig(8) call is a new one, however. Use mdconfig(8) to directly manage memory devices. The `-d` ❷ flag means *destroy*, and the `-u` ❸ gives a device number. The above destroys the device */dev/md41*, or the md device number 41. The memory used by this device is now freed for other uses.

Memory Disks and /etc/fstab

If you list memory disks in */etc/fstab,* FreeBSD automatically creates them at boot time. These entries look more complicated than the other entries, but aren't too bad if you understand the mdmfs(8) commands we've been using so far. To refresh your memory, here's the top of */etc/fstab* and a single standard filesystem:

```
# Device        Mountpoint   FStype   Options      Dump    Pass#
/dev/ad4s2a     /            ufs      rw           1       1
```

We're allowed to use the word *md* as a device name to indicate a memory disk. We can choose the mountpoint just as we would for any other device, and the filesystem type is `mfs`. Under `Options`, list *rw* (for read-write) and the command-line options used to create this device. To create our 8MB file-system mounted at */home/mwlucas/test*, use the following */etc/fstab* entry:

```
md      /home/mwlucas/test      mfs      rw,-s 8m      0   0
```

Looks easy, doesn't it? The only problem is that the long line messes up your nice and even */etc/fstab*'s appearance. Well, they're not the only things that make this file ugly, as we'll soon see.

Mounting Disk Images

Memory disks can also be used to mount and access disk images. This is most useful for viewing the contents of CD images without burning them to disk. Just attach a memory disk to a file with the mdconfig(8) command's `-a` flag. Here, I attach a FreeBSD ISO to a memory device:

```
# mdconfig ❶-a -t ❷vnode -f ❸/home/mwlucas/7.0-CURRENT-snap.iso
❹md0
```

We tell mdconfig(8) to attach ❶ a vnode-backed ❷ memory device to the file specified ❸. mdconfig(8) responds by telling us the device ❹ it's attached to. Now we just mount the device with the proper mount command for the filesystem:

```
# mount -t cd9660 /dev/md0 /mnt
```

One common mistake people make at this point is mounting the image without specifying the filesystem type. You might get an error, or you might get a successful mount that contains no data—by default, mount(8) assumes that the filesystem is FFS!

When you're done accessing the data, be sure to unmount the image and destroy the memory disk device just as you would for any other memory device. While vnode-backed memory disks do not consume system memory, leaving unused memory devices around will confuse you months later when you wonder why they appear in */dev*. If you're not sure what memory devices a system has, use `mdconfig -l` to view all configured md(5) devices.

```
# mdconfig -l
md0 md1
```

I have two memory devices? Add the `-u` flag and the device number to see what type of memory device it is. Let's see what memory device 1 (*/dev/md1*) is:

```
# mdconfig -l -u 1
md1     vnode     456M  /slice1/usr/home/mwlucas/iso/omsa-51-live.iso
```

I have an ISO image mounted on this system? Wow. I should probably reboot some month. Nah, thats too much work, I'll just unmount the filesystem.

Filesystems in Files

One trick used for jails (see Chapter 9) and homebrew embedded systems (see Chapter 20) is building complete filesystem images on a local filesystem. In the previous section, we saw how we could use memory disks to mount and access CD disk images. You can use the same techniques to create,

update, and access FFS disk images. The downside of this is that each image takes up an amount of disk space equal to the size of the disk image. If you create a 500MB disk image, it takes up a full 500MB of space.

To use a filesystem in a file, you must create a file of the proper size, attach the file to a memory device, place a filesystem on the device, and mount the device.

Creating an Empty Filesystem File

The filesystem file doesn't initially contain any data; rather, it's just a file of the correct size for the desired filesystem. You could sit down and type a whole bunch of zeroes to create the file, but FreeBSD provides a unlimited source of nothing to save you the trouble. The */dev/zero* device is chock full of emptiness you can use to fill the file.

Use the same dd(1) command you used to copy the installation floppy images to your filesystem file. Here we copy data from */dev/zero* and to the file *filesystem.file*:

```
# dd ❶if=/dev/zero ❷of=filesystem.file ❸bs=1m ❹count=1k
1048576+0 records in
1048576+0 records out
1073741824 bytes transferred in ❺24.013405 secs (❻44714268 bytes/sec)
```

We take our data from the input file */dev/zero* ❶ and dump it to the output file *filesystem.file* ❷. Each transfer occurs in blocks of 1K ❹, and we do this one million times ❸. This takes a few seconds ❺ to complete, but as */dev/zero* doesn't have to generate the next character, the file fills quickly ❻. If you look in your current directory, you now have a 1GB file called *filesystem.file*.

One common source of confusion with dd(1) is the block size and count. Calculating the final size of a file with dd(1) is like moving a pile of sand; you break the job into loads and move a number of those loads. You can use several loads with a wheelbarrow, many loads with a bucket, or lots and lots of loads with a spoon. The size of each load is the block size, and the number of trips is the count. Perhaps you're using your bare hands, which would correspond to a small block size and a high count. Maybe you have a wheelbarrow, for a medium block size and a moderate count. Perhaps you have a steam shovel and can do the whole job in one scoop. The larger the block size, the more load you put on the system with each block. dd(1) recognizes a variety of abbreviations for increasing block size and count, as shown in Table 8-2.

Suppose you want a 1GB file. Remember that `1k` is actually one kilobyte. One megabyte is a thousand kilobytes, and one gigabyte is one thousand megabytes. If you use a block size of 1 byte, and a count of one gig, you're asking your system to make 1,073,741,824 trips to the sand pile. Each trip is very easy, but there's an awful lot of them! On the other hand, if you select a block size of one gig and a count of 1, you're asking the system to pick up the

whole pile at once. It will not be pleased. Neither will you. Generally speaking, a block size of `1m` and a lower count produces reasonably quick results without overburdening your system. When you use a `1m` block, the count equals the number of megabytes in the file. A count of `1k` creates a 1GB filesystem, a count of `2k` creates a 2GB filesystem, a count of `32` creates a 32MB filesystem, and so on.

Table 8-2: dd Abbreviations

Letter	Meaning	Multiplier
`b`	Disk blocks	512
`k`	Kilo	1024
`m`	Mega	1048576
`g`	Giga	1073741824
`w`	Integer	However many bytes per integer on your platform

Creating the Filesystem on the File

To get a filesystem on the file, you must first associate the file with a device with a vnode-backed memory disks. We did exactly this in the last section:

```
# mdconfig -a -t vnode -f filesystem.file
md0
```

Now, let's make a filesystem on this device. This is much like creating an FFS filesystem on a floppy disk with the newfs(8) command. We specify the `-U` flag to enable soft updates on this filesystem, as soft updates are useful on file-backed filesystems.

```
# newfs -U /dev/md0
/dev/md0: 1024.0MB (2097152 sectors) block size 16384, fragment size 2048
        using 6 cylinder groups of 183.77MB, 11761 blks, 23552 inodes.
        with soft updates
super-block backups (for fsck -b #) at:
 160, 376512, 752864, 1129216, 1505568, 1881920
```

newfs(8) prints out basic information about the disk such as its size, block and fragment sizes, and the inode count.

Now that you have a filesystem, mount it:

```
# mount /dev/md0 /mnt
```

Congratulations! You now have a 1GB file-backed filesystem. Copy files to it, dump it to tape, or use it in any way you would use any other filesystem. But in addition to that, you can move it just like any other file. We'll make use of this when we talk about jails in the next chapter, and it's a vital technique for building your own embedded systems.

File-Backed Filesystems and /etc/fstab

You can mount a file-backed filesystem automatically at boot with the proper entry in */etc/fstab*, much like you can automatically mount any other memory disk. You simply have to specify the name of the file with `-F`, and use `-P` to tell the system to not create a new filesystem on this file but just to use the one already there. Here we mount the file-backed filesystem we created on */mnt* automatically at boot time. (I told you we'd see */etc/fstab* entries uglier than the one we created for generic memory disks, didn't I?)

```
md    /mnt     mfs     rw,-P,-F/home/mwlucas/filesystem.file    0    0
```

Miscellaneous Filesystems

FreeBSD supports several lesser-known filesystems. Most of them are useful only in bizarre circumstances, but bizarre circumstances arise daily in system administration.

devfs(5) is the device filesystem used for */dev*. You cannot store normal files on a devfs filesystem; it only supports device nodes. The kernel and the device filesystem daemon devd(8) directly manage the contents of a device filesystem.

procfs(5) is the process filesystem; it contains lots of information about processes. It's considered a security risk and is officially deprecated on modern FreeBSD releases. You can learn a lot about processes from a mounted process filesystem, however. A few older applications still require a process filesystem mounted on */proc*; if a server application requires procfs, try to find a similar application that does the job without requiring it.

If you're using Linux mode (see Chapter 12), you might need the Linux process filesystem linprocfs(5). Much Linux software requires a process filesystem, and FreeBSD suggests installing linprocfs at */compat/linux/proc* when you install Linux mode. I'd recommend only installing linprocfs if a piece of software complains it's not there.

fdescfs(5), the file descriptor filesystem, offers a filesystem view of file descriptors for each process. Some software is written to require fdescfs(5). It's less of a security risk than procfs, but still undesirable.

Wiring Down Devices

SCSI disks don't always power up in the same order, but FreeBSD numbers SCSI drives in the order that they appear on the SCSI bus. As such, if you change the devices on your SCSI bus, you change the order in which they're probed. What was disk 0 when you installed FreeBSD could become disk 1 after you add a new disk drive. This change would cause partitions to be mounted on the wrong mount points, possibly resulting in data damage. You can have similar problems with SCSI buses—if you add another SCSI

interface, your buses can be renumbered! What was */dev/da0* when you installed FreeBSD could become */dev/da1* or even */dev/da17* after you add a new tape drive. This causes FreeBSD to mount partitions on the wrong mount points.

To prevent this problem, you can hard-code disk numbering into the kernel, a process called *wiring down* the SCSI devices. To wire down the device you need the SCSI ID, SCSI bus number, and LUN (if used) of each device on your SCSI chain, available in */var/run/dmesg.boot*. For example, on a test system I have the following dmesg entries for my SCSI adapter:

```
ahc0: <Adaptec 2940B Ultra2 SCSI adapter> port 0xe000-0xe0ff mem 0xe8042000-
0xe8042fff irq 11 at device 20.0 on pci0
aic7890/91: Ultra2 Wide Channel A, SCSI Id=7, 32/253 SCBs
```

The first line shows that the main SCSI card is an Adaptec 2940B Ultra2 adapter. The second line gives us more information about the chipset on this card. This is really only one physical card. The host adapter is using SCSI ID 7, and no LUN.

A little later in *dmesg.boot* I have entries for all SCSI disks. While these entries include things such as disk capacity, model, speed, and features, the first line for each disk looks much like these:

```
❶da0 at ❷ahc0 ❸bus 0 ❹target 8 ❺lun 0
da1 at ahc0 bus 0 ❻target 9 lun 0
da2 at ahc0 bus 0 target 10 lun 0
...
```

This tells us that the disk da0 ❶ is on SCSI card ahc0 ❷, on SCSI bus 0 ❸, at SCSI ID 8 ❹, at LUN 0 ❺. Disk da1 is on the same card and bus, at SCSI ID 9 ❻.

To wire down a drive, tell the kernel exactly what SCSI bus number to attach to which SCSI card, and then the SCSI ID and LUN of each disk. Do this with kernel hints in */boot/device.hints*:

```
hint.❶scbus.❷0.at="❸ahc0"
hint.❹da.0.at="❺scbus0"
hint.da.0.target="❻8"
hint.da.1.at="scbus0"
hint.da.1.target="❼9"
```

Here, we've told FreeBSD to attach SCSI bus ❶ number 0 ❷ to card ahc0 ❸. Disk da0 ❹ is attached to SCSI bus 0 ❺ at SCSI ID 8 ❻, and disk da1 is attached to the same bus at SCSI ID 9 ❼. On your next reboot the drives will be numbered as you configured. If you add another SCSI card, or more SCSI hard drives, FreeBSD configures the new drives and buses with unit numbers other than those you've reserved for these devices. You can also use `lun` hints if you have targets with multiple LUNs.

Adding New Hard Disks

Before you can use a new hard drive, you must slice it, create filesystems, mount those filesystems, and move data to them. While FreeBSD has command-line tools that can handle all this for you, the simplest and fastest way is with sysinstall(8). We'll assume that you are adding disks to an existing system, and that your eventual goal is to move a part of your data to this disk.

> **BACK UP, BACK UP, BACK UP!**
>
> Before doing anything with disks, be sure that you have a complete backup. A single dumb fat-finger mistake in this process can destroy your system! While you rarely plan to reformat your root filesystem, if it happens you want to recover really, really quickly.

Creating Slices

Your first task in preparing your new hard disk is to create a slice and partition it. Follow these steps:

1. Become root and start sysinstall(8). Choose **Configure**, and then **Disk**.
2. This menu should look somewhat familiar; you used it when you installed FreeBSD. (You can see screenshots in Figure 2-4 in Chapter 2.) You'll see your existing FreeBSD disks and your new disk. Choose the new disk.
3. If this disk is recycled from another server, you might find that it has a filesystem on it. It's usually simplest to remove the existing partitions and start over. Use the arrow keys to move to any existing partitions, and press **D** to delete them.
4. Either create a slice by pressing **C**, or just use the whole disk by pressing **A**. In a server, you almost certainly want to use the entire disk. When you've chosen your slices, make the changes effective immediately by pressing **W**. You'll see a warning like this:

```
WARNING: This should only be used when modifying an EXISTING installation.
If you are installing FreeBSD for the first time then you should simply
type Q when you're finished here and your changes will be committed in one
batch automatically at the end of these questions. If you're adding a
disk, you should NOT write from this screen, you should do it from the
label editor.
```

5. Are you absolutely sure you want to do this now?
6. Yes, you're absolutely sure. Tab over to **Yes** and hit ENTER.
7. You'll then be asked if you want to install a boot manager on this disk. Additional disks don't need boot managers, so arrow down to **Standard** and press the spacebar. The sysinstall program tells you that it has written out the fdisk information successfully. You now have a FreeBSD slice on the disk. Hit **Q** to leave the fdisk part of sysinstall.

Creating Partitions

To create partitions on your disk, follow these steps:

1. Choose the **Label** option of sysinstall(8), on the same submenu as FDISK. Select your new disk to reach the disklabel editor. Here you can create a new partition with the **C** command, specifying its size in either megabytes, gigabytes, disk blocks, or disk cylinders. You can also decide if each new partition will be a filesystem or swap space. When you're asked for a mount point, use */mnt* for the moment. Sysinstall temporarily mounts the new partition at that location.
2. When the disk is partitioned as you need, press **W** to commit the changes. You'll get the same warning you saw in the fdisk menu, and then messages from newfs(8).

   Once newfs(8) finishes, exit sysinstall.

Configuring /etc/fstab

Now tell */etc/fstab* about your new disks. The configuration differs depending on whether the new partition is a filesystem or swap space. Every swap space entry in */etc/fstab* looks like this:

```
devicename          none          swap    sw          0       0
```

If your new swap partition is */dev/da10s1b*, you would add this line to */etc/fstab*:

```
/dev/da10s1b        none          swap    sw          0       0
```

Upon your next reboot, FreeBSD will recognize this swap space. You can also use `swapon -a` *`devicename`* to activate new swap without rebooting.

If you created a data partition, add a new entry as described earlier in this chapter, much like this:

```
/dev/da10s1d        /usr/obj      ufs     rw          2       2
```

Now you can just unmount the new partition from its temporary location, run `mount /usr/obj`, and your new disk is ready for files.

Installing Existing Files onto New Disks

Chances are that you intend your new disk to replace or subdivide an existing partition. You'll need to mount your new partition on a temporary mount point, move files to the new disk, then remount the partition at the desired location.

In our example above, we've mounted our new partition on */mnt*. Now you just need to move the files from their current location to the new disk without changing their permissions. This is fairly simple with tar(1). You can simply tar up your existing data to a tape or a file and untar it in the new location, but that's kind of clumsy. You can concatenate `tar` commands to avoid that middle step, however.

```
# tar cfC - /old/directory . | tar xpfC - /tempmount
```

If you don't speak Unix at parties, this looks fairly stunning. Let's dismantle it. First, we go to the old directory and tar up everything. Then pipe the output to a second command, which extracts the backup in the new directory. When this command finishes, your files are installed on their new disk. For example, to move */usr/src* onto a new partition temporarily mounted at */mnt*, you would do this:

```
# tar cfC - /usr/src . | tar xpfC - /mnt
```

Check the temporary mount point to be sure that your files are actually there. Once you're confident that the files are properly moved, remove the files from the old directory and mount the disk in the new location. For example, after duplicating your files from */usr/src*, you would run:

```
# rm -rf /usr/src/*
# umount /mnt
# mount /usr/src
```

MOVING LIVE FILES

You cannot safely move files that are changing as you copy. For example, if you're moving your email spool to a new partition, you must shut down your mail server first. Otherwise, files change as you're trying to copy them. Tools such as rsync (*/usr/ports/net/rsync*) can help reduce outage duration, but an interruption is still necessary.

Stackable Mounts

Suppose you don't care about your old data; you simply want to split an existing disk to get more space and you plan to recover your data from backup. Fair enough. All FreeBSD filesystems are *stackable*. This is an advanced idea that's not terribly useful in day-to-day system administration, but it can bite you when you try to split one partition into two.

Suppose, for example, that you have data in */usr/src*. See how much space is used on your disk, and then mount a new empty partition on */usr/src*. If you look in the directory afterwards, you'll see that it's empty.

Here's the problem: The new partition is mounted "above" the old disk, and the old disk still has all that data on it. The old partition has no more free space than before you moved the data. If you unmount the new partition

and check the directory again, you'll see the data miraculously restored! The new mount obscured the lower partition.

Although you cannot see the data, data on the old disk still takes up space. If you're splitting a disk to gain space, and you just mount a new disk over part of the old, you won't free any space on your original disk. The moral is: Even if you are restoring your data from backup, make sure that you remove that data from your original disk to recover disk space.

Network Filesystems

A network filesystem allows accessing files on another machine over the network. The two most commonly used network filesystems are the original Network File System (NFS) implemented in Unix, and the CIFS (aka SMB) filesystem popularized by Microsoft Windows. We'll touch on both of these, but start with the old Unix standard of NFS.

Sharing directories and partitions between Unix-like systems is perhaps the simplest Network File System you'll find. FreeBSD supports the Unix standard Network File System out of the box. Configuring NFS intimidates many junior system administrators, but after setting up a file share or two you'll find it not so terribly difficult.

Each NFS connection uses a client-server model. One computer is the server; it offers filesystems to other computers. This is called *NFS exporting*, and the filesystems offered are called *exports*. The clients can mount server exports in a manner almost identical to that used to mount local filesystems.

One interesting thing about NFS is its statelessness. NFS does not keep track of the condition of a connection. You can reboot an NFS server and the client won't crash. It won't be able to access files on the server's export while the server is down, but once it returns, you'll pick up right where things left off. Other network file sharing systems are not always so resilient. Of course, statelessness also causes problems as well; for example, clients cannot know when a file they currently have open is modified by another client.

NFS INTEROPERABILITY

Every NFS implementation is slightly different. You'll find minor NFS variations between Solaris, Linux, BSD, and other Unix-like systems. NFS should work between them all, but might require the occasional tweak. If you're having problems with another Unix-like operating system, check the *FreeBSD-net* mailing list archive; the issue has almost certainly been discussed there.

Both NFS servers and clients require kernel options, but the various NFS commands dynamically load the appropriate kernel modules. FreeBSD's GENERIC kernel supports NFS, so this isn't a concern for anyone who doesn't customize their kernel.

NFS is one of those topics that have entire books written about them. We're not going to go into the intimate details about NFS, but rather focus

on getting basic NFS operations working. If you're deploying complicated NFS setups, you'll want to do further research. Even this basic setup lets you accomplish many complicated tasks.

Enabling the NFS Server

Turn on NFS server support with the following *rc.conf* options. While not all of these options are strictly necessary for all environments, turning them all on provides the broadest range of NFS compatibility and decent out-of-the-box performance.

```
❶ nfs_server_enable="YES"
❷ rpcbind_enable="YES"
❸ mountd_enable="YES"
❹ rpc_lockd_enable="YES"
❺ rpc_statd_enable="YES"
```

First, tell FreeBSD to load the *nfsserver.ko* ❶ kernel module, if it's not already in the kernel. rpcbind(8) ❷ maps remote procedure calls (RPC) into local network addresses. Each NFS client asks the server's rpcbind(8) daemon where it can find a mountd(8) daemon to connect to. mountd(8) ❸ listens to high-numbered ports for mount requests from clients. Enabling the NFS server also starts nfsd(8), which handles the actual file request. rpc.lockd(8) ❹ ensures smooth file locking operations over NFS, and rpc.statd(8) ❺ monitors NFS clients so that the NFS server can free up resources when the host disappears.

While you can start all of these services at the command line, if you're just learning NFS it's best to reboot your system after enabling NFS server. Afterwards, you'll see rpc.lockd, rpc.statd, nfsd, mountd, and rpcbind listed in the output of sockstat(1). If you don't see all of these daemons listening to the network, check */var/log/messages* for errors.

Configuring NFS Exports

Now tell your server what it can share, or *export*. We could just export all directories and filesystems on the entire server, but any competent security administrator would have a (justified) fit. As with all server configurations, permit as little access as possible while still letting the server fulfill its role. For example, in most environments clients have no need to remotely mount the NFS server's root filesystem.

Define which clients may mount which filesystems and directories in */etc/exports*. This file takes a separate line for each disk device on the server and each client or group of clients that access that device. Each line has up to three parts:

- Directories or partitions to be exported (mandatory)
- Options on that export
- Clients that can connect

Each combination of clients and a disk device can only have one line in the exports file. This means that if */usr/ports* and */usr/home* are on the same partition, and you want to export both of them to a particular client, they

must both appear in the same line. You cannot export */usr/ports* and */usr/home* to one client with different permissions. You don't have to export the entire disk device, mind you; you can export a single directory within a partition. This directory cannot contain either symlinks or double or single dots.

Of the three parts of the */etc/exports* entry, only the directory is mandatory. If I wanted to export my home directory to every host on the Internet, I could use an */etc/exports* line consisting entirely of this:

```
/home/mwlucas
```

We show no options and no host restrictions. This would be foolish, of course, but I could do it.[3]

After editing the *exports* file, tell mountd to reread it:

```
# /etc/rc.d/mountd restart
```

mountd(8) logs any problems in */var/log/messages*. The log messages are generally enigmatic: While mountd(8) informs you that a line is bad, it generally doesn't say why. The most common errors I experience involve symlinks.

Enabling the NFS Client

Configuring the client is much simpler. In */etc/rc.conf*, put:

```
nfsclient="YES"
```

Then, reboot or run `/etc/rc.d/nfsclient start`. Either will enable NFS client functions.

Now we can mount directories or filesystems exported by NFS servers. Instead of using a device name, use the NFS server's host name and the directory you want to mount. For example, to mount the directory */home/mwlucas* from my server *sardines* onto the directory */mnt*, I would run:

```
# mount sardines:/home/mwlucas /mnt
```

Afterwards, test your mount with df(1).

```
# df
Filesystem             1K-blocks      Used    Avail Capacity  Mounted on
/dev/ad4s1a              1012974    387450   544488    42%    /
devfs                          1         1        0   100%    /dev
/dev/ad4s1f            109009798  12959014 87330002    13%    /usr
/dev/ad4s1e              1012974     42072   889866     5%    /var
sardines:/home/mwlucas 235492762 150537138 66116204    69%    /mnt
```

[3] Why is there no safeguard against shooting yourself in the foot like this? Well, Unix feels that anyone dumb enough to do this doesn't deserve to be its friend. Various people keep trying to put Unix in therapy for this type of antisocial behavior, but it just isn't interested.

The NFS-mounted directory shows up as a normal partition, and I can read and write files on it as I please. Well, maybe not *entirely* as I please . . .

NFS and Users

File ownership and permissions are tied to UID numbers. NFS uses UID to identify the owner. For example, on my laptop the user mwlucas has the UID of 1001. On the NFS server, mwlucas also has the UID 1001. This makes my life easy, as I don't have to worry too much about file ownership; I have the same privileges on the server as on my laptop. This can be a problem on a large network, where users have root on their own machines. The best way around this is to create a central repository of authorized users via Kerberos. On a small network or on a network with a limited number of NFS users, this usually isn't a problem; you can synchronize */etc/master.passwd* on your systems or just assign the same UID to each user on each system.

The root user is handled slightly differently, however. An NFS server doesn't trust root on other machines to execute commands as root on the server. After all, if an intruder breaks into an NFS client you don't want the server to automatically go down with it. You can map requests from root to any other username. For example, you might say that all requests from root on a client will run as the user nfsroot on the server. With careful use of groups, you could allow this nfsroot user to have limited file access. Use the `-maproot` option to map root to another user. Here, we map UID 0 (root) on the client to UID 5000 on the server:

```
/usr/home/mwlucas -maproot=5000
```

If you really want the root user on the client to have root privileges on the server, you can use `-maproot` to map root to UID 0. This might be suitable on your home network or on a test system.

If you do not use a `maproot` statement, the NFS maps a remote root account to `nobody:nobody` by default.

Don't forget to restart mountd(8) after editing the exports file.

Exporting Multiple Directories

Many directories under */usr* would make sensible exports. Good candidates include */usr/src*, */usr/obj*, and */usr/ports/distfiles*. List all directories of the same partition on the same line in */etc/exports*, right after the first exported directory, separated by spaces. My */etc/exports* now looks like this:

```
/usr/home/mwlucas /usr/src /usr/obj /usr/ports/distfiles -maproot=5000
```

There are no identifiers, separators, or delimiters between the parts of the line. Yes, it would be easier to read if we could put each shared directory on its own line, but we can't; they're all on the same partition. The FreeBSD team could rewrite this so that it had more structure, but then FreeBSD's */etc/exports* would be incompatible with that from any other Unix.

As with many other configuration files, you can use a backslash to break a single line of configuration into multiple lines. You might find the above configuration more readable as:

```
/usr/home/mwlucas \
    /usr/src \
    /usr/obj \
    /usr/ports/distfiles \
    -maproot = 5000
```

Restricting Clients

To allow only particular clients to access an NFS export, list them at the end of the */etc/exports* entry. Here, we restrict our share above to one IP address:

```
/usr/home/mwlucas /usr/src /usr/obj /usr/ports/distfiles \
    -maproot=5000 192.168.1.200
```

You can also restrict file shares to clients on a particular network by using the `-network` and `-mask` qualifiers:

```
/usr/home/mwlucas /usr/src /usr/obj /usr/ports/distfiles \
    -maproot=5000 -network 192.168.0 -mask 255.255.255.0
```

This lets any client with an IP address beginning in 192.168.0 access your NFS server. I use a setup much like this to upgrade clients quickly. I build a new world and kernel on the NFS server, then let the clients mount those partitions and install the binaries over NFS.

Combinations of Clients and Exports

Each line in */etc/exports* specifies exports from one partition to one host or set of hosts. Different hosts can have entirely different export statements.

```
/usr/home/mwlucas /usr/src /usr/obj /usr/ports/distfiles \
    -maproot=5000 192.168.1.200
/usr -maproot=0 192.168.1.201
```

Here, I've exported several subdirectories of */usr* to the NFS client at 192.168.1.200. The NFS client at 192.168.1.201 gets to mount the whole of */usr*, and may even do so as root.

NFS Performance and Options

FreeBSD uses conservative NFS defaults, so that it can work well with any other Unix-like operating system. If you're working in a pure FreeBSD environment or if your environment only contains higher-end Unix systems, you can improve NFS performance with mount options.

First of all, NFS defaults to running over UDP. The `tcp` or `-T` option tells the client to request a mount over TCP.

Programs expect the filesystem to not disappear, but when you're using NFS it's possible that the server will vanish from the network. This makes programs on the client trying to access the NFS filesystem hang forever. By making your NFS mount *interruptible*, you will be able to interrupt processes hung on unavailable NFS mount with CTRL-C. Set interruptibility with `intr`.

FreeBSD can tell clients if and when a filesystem is no longer accessible. Programs will fail when trying to access the filesystem, instead of hanging forever.

Finally, you can set the size of read and write requests. The defaults are well-suited to networks of the early 1990s, but you can set the read and write size to more modern values with the `-r` and `-w` options. I find that 32768 is a good value for both. So, putting this all together, I could have my client mount an NFS file system like this:

```
# mount -o tcp,intr,soft,-w=32768,-r=32768 server:/usr/home/mwlucas /mnt
```

The same entry in */etc/fstab* looks like this:

```
server:/usr/home/mwlucas /mnt nfs rw,-w=32768,-r=32768,tcp,soft,intr 0 0
```

This general NFS configuration gives me good throughput speed on a local network, limited only by hardware quality.

While NFS is pretty straightforward for simple uses, you can spend many hours adjusting, tuning, and enhancing it. If you wish to build a complicated NFS environment, don't rely entirely on this brief introduction but spend time with a good book on the subject.

FreeBSD and CIFS

If you're on a typical office network, the standard network file sharing protocol is Microsoft's Common Internet File Sharing, or CIFS. (CIFS was once known as Server Message Block, or SMB.) This is the "Network Neighborhood" that Windows users can access. While originally provided only by Microsoft Windows systems, this protocol has become something of a standard. Thankfully, today there's an open source CIFS file sharing server called Samba. Many commercial products provide services via this protocol. FreeBSD includes kernel modules to support the filesystem and programs to find, mount, and use CIFS shares.

Prerequisites

Before you begin working with Microsoft file shares, gather the following information about your Windows network:

- Workgroup or Windows domain name
- Valid Windows username and password
- IP address of the Windows DNS server

Kernel Support

FreeBSD uses several kernel modules to support CIFS. The *smbfs.ko* module supports basic CIFS operations. The *libmchain.ko* and *libiconv.ko* modules provide supporting functions and load automatically when you load *smbfs.ko*. You can compile these statically in your kernel as:

```
options    NETSMB
options    LIBMCHAIN
options    LIBICONV
options    SMBFS
```

Of course, you can load these at boot time with the appropriate entries in */boot/loader.conf*.

Configuring CIFS

CIFS relies on a configuration file, either *$HOME/.nsmbrc* or */etc/nsmb.conf*. All settings in */etc/nsmb.conf* override the settings in user home directories. The configuration file is divided into sections by labels in square brackets. For example, settings that apply to every CIFS connection are in the `[default]` section. Create your own sections to specify servers, users, and shares, in one of the following formats:

```
[servername]
[servername:username]
[servername:username:sharename]
```

Information that applies to an entire server goes into a section named after the server. Information that applies to a specific user is kept in a username section, and information that applies to only a single share is kept in a section that includes the share name. You can lump the information for all the shares under a plain [*servername*] entry if you don't have more specific per-user or per-share information.

Configuration entries use the values from the CIFS system—for example, my Windows username is lucas_m, but my FreeBSD username is mwlucas, so I use lucas_m in *nsmb.conf*.

nsmb.conf Keywords

Specify a *nsmb.conf* configuration with keywords and values under the appropriate section. For example, servers have IP addresses and users don't, so you would only use an IP address assignment in the server section. To use a keyword, assign a value with an equal sign, as in *keyword=value*. Here are the common keywords; for a full list, see nsmb.conf(5).

workgroup=string

The `workgroup` keyword specifies the name of the Windows domain or workgroup you want to access. This is commonly a default setting used for all servers.

addr=a.b.c.d

The `addr` keyword sets the IP address of a CIFS server. This keyword can only appear under a plain `[servername]` label.

nbns=a.b.c.d

The `nbns` keyword sets the IP address of a NetBIOS (WINS) nameserver. You can put this line in the default section or under a particular server. If you have Active Directory (which is based on DNS), you can use DNS host names. Adding a WINS server won't hurt your configuration, however, and helps in testing basic CIFS setup.

password=string

The `password` keyword sets a clear-text password for a user or a share. If you must store passwords in */etc/nsmb.conf*, be absolutely certain that only root can read the file. Storing a password in *$HOME/.nsmbrc* is a bad idea on a multi-user system.

You can scramble your Windows password with `smbutil crypt`, generating a string that you can use for this keyword. The scrambled string has double dollar signs (`$$`) in front of it. While this helps prevent someone accidentally discovering the password, a malicious user can unscramble it easily.

CIFS Name Resolution

Let's build a basic *nsmb.conf* file. As an absolute minimum, we first need to find hosts, which means we need a workgroup name. We'll set the domain controller as the WINS server. I also have a user set up on the Windows-based servers to share files, so I'm going to use that as a default in *nsmb.conf*.

```
[default]
workgroup=BIGLOSER
nbns=192.168.1.66
username=unix
```

Armed with this information, we can perform basic SMB name queries.

```
# smbutil lookup ntserv1
Got response from 192.168.1.66
IP address of ntserv1: 192.168.1.4
```

If this works, you have basic CIFS functionality.

Other smbutil(1) Functions

Before you can mount a filesystem from a Windows host, you must log into the host. Only root can perform this operation.

```
# smbutil login //unix@ntserv1
Password:
```

So, our configuration is correct. Let's see what resources this server offers with smbutil's `view` command.

```
# smbutil view //unix@ntserv1
Password:
Share        Type       Comment
-------------------------------
IPC$         pipe       Remote IPC
ADMIN$       disk       Remote Admin
C$           disk       Default share
unix      disk
4 shares listed from 4 available
```

You'll get a list of every shared resource on the CIFS server. Now, assuming you're finished, log out of the server.

```
# smbutil logout //unix@ntserv1
```

Mounting a Share

Now that you've finished investigating, let's actually mount a share with mount_smbfs(8). The syntax is as follows:

```
# mount_smbfs //username@servername/share /mount/point
```

I have a share on this Windows box called *MP3* that I want to access from my FreeBSD system. To mount this as */home/mwlucas/smbmount*, I would do this:

```
# mount_smbfs //unix@ntserv1/MP3 /home/mwlucas/smbmount
```

mount(8) and df(1) show this share attached to your system, and you can access documents on this server just as you could any other filesystem.

Other mount_smbfs Options

mount_smbfs includes several options to tweak the behavior of mounted CIFS filesystems. Use the `-f` option to choose a different file permission mode and the `-d` option to choose a different directory permission mode. For example, to set a mount so that only I could access the contents of the directory, I would use `mount_smbfs -d 700`. This would make the FreeBSD permissions far more stringent than the Windows privileges, but that's perfectly all right with me. I can change the owner of the files with the `-u` option, and the group with the `-g` option.

Microsoft filesystems are case-insensitive, but Unix-like operating systems are case sensitive. CIFS defaults to leaving the case as it finds it, but that may not be desirable. The `-c` flag makes mount_smbfs(8) change the case on the filesystem: `-c l` changes everything to lowercase and `-c u` changes everything to uppercase.

Sample nsmb.conf Entries

Here are samples of *nsmb.conf* entries for different situations. They all assume they're part of a configuration where you've already defined a workgroup, NetBIOS nameserver, and a username with privileges to access the CIFS shares.

Unique Password on a Standalone System

You would use something like the following if you have a machine named desktop with a password-protected share. Many Windows 9*x* systems have this sort of password-protection feature.

```
[desktop:shareusername]
password=$$1789324874ea87
```

Accessing a Second Domain

In this example, we're accessing a second domain, named development. This domain has a username and password different from those at our default domain.

```
[development]
workgroup=development
username=support
```

CIFS File Ownership

Ownership of files between Unix-like and Windows systems can be problematic. For one thing, your FreeBSD usernames probably won't map to Windows usernames, and Unix has a very different permissions scheme compared to Windows.

Since you're using a single Windows username to access the share, you have whatever access that account has to the Windows resources, but you must assign the proper FreeBSD permissions for that mounted share. By default, mount_smbfs(8) assigns the new share the same permissions as the mount point. In our earlier example, the directory */home/mwlucas/smbmount* is owned by the user mwlucas and has permissions of 755. These permissions say that mwlucas can edit what's in this directory, but nobody else can. Even though FreeBSD says that this user can edit those files, Windows might still not let that particular user edit the files it's sharing out.

Serving CIFS Shares

Just as FreeBSD can access CIFS shares, it can also serve them to CIFS clients with Samba. You can find Samba in */usr/ports/net/samba3*. You'll find the Samba website at *http://www.samba.org*, along with many useful tutorials. Serving CIFS shares from FreeBSD is much more complicated than accessing them, so we'll end our discussion here before this book grows even thicker.

devfs

devfs(5) is a dynamic filesystem for managing device nodes. Remember, in a Unix-like operating system *everything* is a file. This includes physical hardware. Every device on the system has a device node under */dev*.

Once upon a time, the system administrator was responsible for making these device node files. Lucky sysadmins managed an operating system that came with a shell script to handle device node creation and permissions. If the OS authors had not provided such a shell script, or if the server had unusual hardware not included in that shell script, the sysadmin had to create the node with black magic and mknod(8). If any little thing went wrong, the device would not work. The other option was to ship the operating system with device nodes for every piece of hardware imaginable. System administrators could be confident—well, *mostly* confident—that the desired device nodes were available, somewhere, buried within the thousands of files under */dev*.

Of course, the kernel knows exactly what characteristics each device node should have. With devfs(5), FreeBSD simply asks the kernel what device nodes the kernel thinks the system should have and provides exactly those—and no more. This works well for most people. You and I are not "most people," however. We expect odd things from our computers. Perhaps we need to make device nodes available under different names, or change device node ownership, or configure our hardware uniquely. FreeBSD breaks the problem of device node management into three pieces: configuring devices present at boot, global availability and permissions, and configuring devices that appear dynamically after boot with devd(8).

> **DEVICE MANAGEMENT AND SERVERS**
>
> For the most part, device node management on servers works without any adjustment or intervention. The place I most often need to muck with device nodes is on laptops and the occasional workstation. FreeBSD's device node management tools are very powerful and flexible, and include support for things I wouldn't expect to use in a century. We'll only touch upon the basics. Don't think that you must master devfs(5) to get your server running well!

devfs at Boot: devfs.conf

The big problem sysadmins have with a dynamic devfs is that any changes made to it disappear on reboot. When device nodes were permanent files on disk, the sysadmin could symlink to those nodes or change their permissions without worrying that his changes would vanish. With an automated, dynamic device filesystem, this assurance disappears. (Of course, you no longer have to worry about occult mknod(8) commands either, so you're better off in the long run.) The device node changes could include, for example:

- Making device nodes available under different names
- Changing ownership of device nodes
- Concealing device nodes from users

At boot time, devfs(8) creates device nodes in accordance with the rules in */etc/devfs.conf*.

devfs.conf

The */etc/devfs.conf* file lets you create links, change ownership, and set permissions for devices available at boot. Each rule has the following format:

```
action      realdevice      desiredvalue
```

The valid actions are `link` (create a link), `perm` (set permissions), and `own` (set owner). The `realdevice` entry is a preexisting device node, while the last setting is your desired value. For example, here we create a new name for a device node:

```
❶link       ❷acd0           ❸cdrom
```

We want a symbolic link ❶ to the device node */dev/acd0* ❷ (an ATAPI CD drive), and we want this link to be named */dev/cdrom* ❸. If we reboot with this entry in */etc/devfs.conf*, our CD device at */dev/acd0* also appears as */dev/cdrom*, as many desktop multimedia programs expect.

To change the permissions of a device node, give the desired permissions in octal form as the desired value:

```
perm        acd0            666
```

Here, we set the permissions on */dev/acd0* (our CD device, again) so that any system user can read or write to the device. Remember, changing the permissions on the */dev/cdrom* link won't change the permissions on the device node, just the symlink.

Finally, we can also change the ownership of a device. Changing a device node's owner usually indicates that you're solving a problem the wrong way and that you may need to stop and think. FreeBSD happily lets you mess up your system if you insist, however. Here, we let a particular user have absolute control of the disk device */dev/da20*:

```
own         da20            mwlucas:mwlucas
```

This might not have the desired effect, however, as some programs still think that you must be root to carry out operations on devices. I've seen more than one piece of software shut itself down if it's not run by root, without even trying to access its device nodes. Changing the device node permissions won't stop those programs' complaints when they are run by a regular user.

Configuration with devfs.conf(5) solves many problems, but not all. If you want a device node to simply be invisible and inaccessible, you must use devfs rules.

Global devfs Rules

Every devfs(5) instance behaves according to the rules defined in *devfs.rules*. The devfs rules apply to both devices present at boot and devices that appear and disappear dynamically. Rules allow you to set ownership and permissions on device nodes and make device nodes visible or invisible. You cannot create symlinks to device nodes with devfs rules.

Similar to */etc/rc.conf* and */etc/defaults/rc.conf*, FreeBSD uses */etc/devfs.rules* and */etc/defaults/devfs.rules*. Create an */etc/defvs.rules* for your custom rules and leave the entries in the defaults file alone.

devfs Ruleset Format

Each set of devfs rules starts with a name and a ruleset number between square brackets. For example, here are a few basic devfs rules from the default configuration:

```
[❶devfsrules_hide_all=❷1]
❸add hide
```

The first set of rules is called `devf s_hide_all` ❶ and is ruleset number 1 ❷. This ruleset contains only one rule ❸.

Ruleset Content

All devfs rules (in files) begin with the word `add`, to add a rule to the ruleset. You then have either a `path` keyword and a regex of device names, or a `type` keyword and a device type. At the end of the rule, you have an *action*, or a command to perform. Here's an example of a devfs rule:

```
add path da* user mwlucas
```

This rule assigns all device nodes with a node name beginning with `da` to the ownership of mwlucas. On a multi-user system, or a system with SCSI hard drives, this would be a bad idea. On a laptop, where the only device nodes beginning with `da` are USB storage devices, it's not such a bad idea.

Devices specified by path use standard shell regular expressions. If you want to match a variety of devices, use an asterisk as a wildcard. For example, `path ad1s1` matches exactly the device */dev/ad1s1*, but `path ad*s*` matches every device node with a name beginning with `ad`, a character, the letter `s`, and possibly more characters. You could tell exactly what devices are matched by a wildcard by using it at the command line.

```
# ls /dev/ad*s*
```

This lists all slices and partitions on your ATA hard drives, but not the devices for the entire drive.

The `type` keyword indicates that you want the rule to apply to all devices of a given type. Valid keywords are `disk` (disk devices), `mem` (memory devices), `tape` (tape devices), and `tty` (terminal devices, including pseudoterminals). The `type` keyword is rarely used exactly because it's so sweeping.

If you include neither a `path` nor a `type`, devfs applies the action at the end of the rule to all device nodes. In almost all cases, this is undesirable.

The ruleset action can be any one of `group`, `user`, `mode`, `hide`, and `unhide`. The `group` action lets you set the group owner of the device, given as an additional argument. Similarly, the `user` action assigns the device owner. Here, we set the ownership of `da` disks to the username desktop and the group usb:

```
add path da* user desktop
add path da* group usb
```

The `mode` action lets you assign permissions to the device in standard octal form:

```
add path da* mode 664
```

The `hide` keyword lets you make device nodes disappear, and `unhide` makes them reappear. Since no program can use a device node if the device is invisible, this is of limited utility except when the system uses jail(8). We'll look at that specific use of `hide` and `unhide` in the next chapter.

Once you have a set of devfs rules you like, enable them at boot in */etc/rc.conf*. Here, we activate the devfs ruleset named `laptoprules`:

```
devfs_system_rulesets="laptoprules"
```

Remember, devfs rules apply to the devices in the system at boot and the devices configured dynamically after startup. To finish up, let's look at dynamic devices.

Dynamic Device Management with devd(8)

Hot-swappable hardware is much more common now than it was even ten years ago. In the last century, laptop network cards were considered cutting-edge. While you might drop a new CD into the workstation's cup holder, you wouldn't hot-plug a whole CD drive into your desktop. Today, that's entirely different. Flash drives are essentially USB-based hard drives that you can attach and detach at will, and other USB hardware lets you add keyboards, mice, and even network cards on a whim.

FreeBSD's devfs dynamically creates new device nodes when this hardware is plugged in and erases the nodes when the hardware is removed, making using these dynamic devices much simpler. devd(8) takes this a step further by letting you run userland programs when hardware appears and disappears.

devd Configuration

devd(8) reads its configuration from */etc/devd.conf* and any file under */usr/local/etc/devd/*. I recommend placing your local rules in */usr/local/etc/devd/devd.conf* to simplify upgrades. You could also add different rules files for different types of devices, if you find your devd(8) configuration becoming very complicated. You'll find four types of devd(8) rules: `attach`, `detach`, `nomatch`, and `notify`.

`attach` rules are triggered when matching hardware is attached to the system. When you plug in a network card, an `attach` rule configures the card with an IP address and brings up the network.

`detach` rules trigger when matching hardware is removed from the system. `detach` rules are uncommon, as the kernel automatically marks resources unavailable when the underlying hardware disappears, but you might find uses for them.

The `nomatch` rules triggers when new hardware is installed but not attached to a device driver. These devices do not have device drivers in the current kernel.

devd(8) applies `notify` rules when the kernel sends a matching event notice to userland. For example, the console message that a network interface has come up is a `notify` event. Notifications generally appear on the console or in */var/log/messages*.

devd(8) rules also have priority, with 0 being the lowest. Only the highest matching rule is processed, while lower-priority matching rules are skipped. Here's a sample devd(8) rule:

```
❶notify ❷0 {
          match "system"          ❸"IFNET";
          match "type"            ❹"ATTACH";
          action ❺"/etc/pccard_ether $subsystem start";
};
```

This is a `notify` ❶ rule, which means it activates when the kernel sends a message to userland. As a priority 0 ❷ rule, this rule can only be triggered if no rule of higher-priority matches the criteria we specify. This rule only triggers if the notification is on the network system `IFNET` ❸ (network), and only if the notification type is `ATTACH` ❹—in other words, when a network interface comes up. Under those circumstances, devd(8) takes action and runs a command to configure the network interface ❺.

devd(8) supports many options to handle all sorts of different situations. If you want to automatically mount a particular USB flash disk on a certain mount point, you can do that by checking the serial number of every USB device you put in. If you want to configure Intel network cards differently than 3Com network cards, you can do that too. We'll do enough with devd(8) to introduce you to its abilities, mainly through examples, but we won't delve deeply into it.

devd(8) Example: Laptops

I use my laptop both at work and at home. At work I use a wired connection, while at home I use wireless. Both require very particular setups, different DNS servers, and different configurations all around. I could cheat by renumbering my home network to match my work network, but experience tells me that my home network will last much longer than my attachment to any one job. (Maybe I'm wrong.[4]) I want the laptop to configure itself for home when I plug in my wireless card. I want the laptop to configure itself for work when I plug in the CAT5 cable to the integrated network interface.

If a wired interface is configured to use DHCP, plugging in the cable tells FreeBSD to start dhclient(8) and configure the network. When I plug in a wireless card, FreeBSD checks for a configuration and tries to start the network. Here's my wireless configuration from */etc/rc.conf* (this is all one line in the configuration, but it is broken to fit in this book):

```
ifconfig_wi0="inet dhcp ssid WriteFaster wepkey
0xdeadbeefbadc0decafe1234567 weptxkey 1 wepmode on channel 1"
```

When I insert my wireless card, the kernel attaches the card to the driver wi0. This is an `attach` event, and devd(8) searches its configuration for a corresponding rule. The following entry in */etc/devd.conf* matches:

```
❶attach  ❷0 {
         ❸media-type "802.11";
         ❹action "/etc/pccard_ether $device-name start";
};
```

We inserted a card into the laptop, and FreeBSD attached the wi(4) driver to it. This is an `attach` ❶ event. This rule has a priority of 0 ❷, so any other matching rule will run before this default rule. Anything with a media type of 802.11 ❸ (wireless) matches this rule. When an `attach` event matches this rule, FreeBSD runs a command ❹ to configure the network. I could run a custom script instead of using the built-in FreeBSD support, but why bother when the default configuration does everything I need?

Unlike the wireless I use at home, the Ethernet interface I use at work is always attached to my system. The interface might not be plugged into a network, but the card itself is always attached. I don't want to configure this interface every time the system boots, because if the interface isn't plugged in I'll have a long delay while my laptop's SSH daemon and other services time out on DNS queries. At work I have a variety of NFS mounts, as well as custom client/server settings not available from the DHCP server.

[4] It would be nice, some day, to have employment that didn't end with four corporate security gorillas dragging me to the door in chains, with my new ex-boss screeching in the background about federal regulators, crème brûlée, and caffeinated lemurs. I'm certain this is just coincidence, mind you, as nobody's ever pressed charges.

This means that I need to run a custom shell script to configure my network for work, but only when the network cable is plugged in. I want devd to watch for a notification that the interface has come up and then run my custom shell script. I've written the custom entry below and placed it in */usr/local/etc/devd/devd.conf*:

```
❶notify  ❷10 {
           match "system"          ❸"IFNET";
           match "type"            ❹"LINK_UP";
           media-type              ❺"ethernet";
❻#         action "/etc/rc.d/dhclient start $subsystem";
         ❼action "/home/mwlucas/bin/jobnetwork.sh";
};
```

This is a notification ❶ rule, which means it triggers when the kernel sends a matching notification message to the system. It also has a priority of 10 ❷, so it runs before any of FreeBSD's default rules. This rule triggers on the network ❸ system, when a link comes up ❹ (i.e., when we plug in a network cable), if the link is Ethernet ❺. When all of these conditions are met, FreeBSD runs a shell script ❼ in my home directory.

Note that this rule has a commented-out line ❻. When I said I wrote this rule, I lied. I actually copied the default entry from */etc/devd.conf* and modified it slightly. FreeBSD already has rules for handling network interfaces coming up, but I wanted to do something a little different. Instead of coming up with all these conditions myself, I found a entry that was almost right and modified it. I suggest you do the same. Just be sure to give your customized rule a higher priority than the default rule, so FreeBSD uses your rule instead of its own.

Another devd Example: Flash Drives

Copying an entry is cheating, I hear you cry. Well, let's try an example where FreeBSD doesn't have any infrastructure in place. On my laptop, I want any USB storage mounted on */media* automatically. The first USB storage device plugged into a laptop shows up in FreeBSD as */dev/da0*, but the device name is umass0. (If you already have other USB or SCSI devices, these numbers will differ for you.) Generally speaking, all flash media is formatted with the FAT32 fileystem, so you would use `mount -t msdosfs` to mount it.

When the device is plugged in, FreeBSD attaches it to the kernel, so we use an `attach` rule:

```
attach 10 {
       match "device-name"    ❶"umass0";
       action "/home/mwlucas/bin/mountusb.sh";
};
```

I assign this rule a priority of 10, so it overrides any new priority 0 functions FreeBSD might install during an upgrade. (If FreeBSD grows the ability to do this automatically, I'll probably use that instead, but I don't want an upgrade to change my system behavior unless I know about it.) Here, we match on the device name ❶. When the device appears, a shell script in my directory runs.

Why not just run the mount(8) command? USB devices need a moment or two to warm up and stabilize before they can be accessed. Also, some USB flash drives need to be poked before they really start to work. The shell script I use here is terribly simpleminded, but it suffices:

```
#!/bin/sh
sleep 2
/usr/bin/true > /dev/da0
/sbin/mount -t msdosfs /dev/da0s1 /media
```

While this handles mounting the device for me, devd(8) won't unmount the device. FreeBSD expects the sysadmin to unmount filesystems before removing the underlying devices. It's no different from unmounting a CD or floppy disk before ejecting it.

While devd(8) has many more advanced features, these two examples cover all the features I've needed or wanted to use in several years. Please read the manual page if you seek further enlightenment. Now that you have a little bit too much understanding of FreeBSD filesystems, let's look at its more advanced security features.