ABSOLUTE OPENBSD

2ND EDITION

UNIX FOR THE PRACTICAL PARANOID

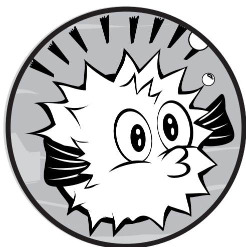
MICHAEL W. LUCAS

"The definitive book on OpenBSD gets a long-overdue refresh."
—Theo de Raadt, OpenBSD Founder

8

DISKS AND FILESYSTEMS

Oh, my head hurts bad.
Rings of ones and zeros, ouch!
Filesystems hide them.



Proper data management is perhaps a systems administrator's most vital duty. You can replace almost every computer component, but the data on your disk is irreplaceable. Perhaps that data isn't important or it's backed up, but losing files will ruin your day. As a sysadmin, you must protect important data by carefully managing your disks and filesystems.

We covered the basics of disklabels and MBR partitions in Chapter 2, but OpenBSD lets you use and abuse disks and filesystems in any number of ways. You'll learn how in this chapter.

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

Device Nodes

A *device node* is a file that provides a logical interface to a piece of hardware. By reading from a device node, sending data to it, or using a command on it, you're telling the operating system to perform an action on a piece of hardware or, in some cases, a logical device.

Different devices behave differently when data is sent to them. For example, writing to the console makes text appear on the screen or terminal, while writing to a disk device puts data on that disk. (OpenBSD puts device nodes in */dev* and disallows device nodes on other filesystems.)

Many disk management programs expect to be given a device name as an argument. Unfortunately, device node names are frequently cryptic and vary widely among operating systems—even on closely related operating systems that run on the same hardware. To simplify your life just a bit, Table 8-1 lists the device node names for common OpenBSD disk devices.

Table 8-1: Common Disk Device Node Names

Device Node	Description
*/dev/fd**	Floppy disk (block)
*/dev/rfd**	Floppy disk (raw)
*/dev/wd**	IDE and some SATA disks (block)
*/dev/rwd**	IDE and some SATA disks (raw)
*/dev/sd**	SCSI/SAS/SATA/USB/RAID/non-IDE disk (block)
*/dev/rsd**	SCSI/SAS/SATA/USB/RAID/non-IDE disk (raw)
*/dev/cd**	CD/DVD drive (block)

Device names also have a number that tells you which instance of that device it refers to. The numbering starts at 0. The first IDE hard drive is */dev/wd0*, */dev/wd1* is the second, and */dev/cd1* is the second CD drive.

Every partition is assigned a letter. For example, the root partition is *a*, the swap area is *b*, the whole disk is *c*, and so on. Each partition also has a separate device node, the result of appending the partition letter to your disk device name. For example, if you install to a single IDE drive, your root partition is */dev/wd0a*.

Raw and Block Devices

Notice in Table 8-1 that devices are listed in either block or raw (character) mode. This refers to how the devices are accessed.

Block Devices

Hard disks are usually accessed using a block device node (sometimes called a *cooked* device node). When accessing a device as a block, data transmitted to or from the device is *buffered*, or collected until there is sufficient data to

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

make accessing the device worth the trouble. Block devices are generally considered more efficient than raw devices.

The device nodes for block devices are named after the device driver; for example, */dev/wd3*.

Raw Devices

Raw devices are sometimes called *character* devices, because they access a device one character at a time. If you need to control exactly how data appears on a disk (for example, when creating a filesystem) use a raw device. Raw device nodes have an *r* in front of their name, as in */dev/rwd3*.

Raw devices do no buffering. When you tell your system to write to a raw device, the data is transmitted immediately. Raw mode works best with software that provides its own buffering or that wants to arrange data in a specific way.

Here's an easy way to remember the difference between block and raw throughput: Say you spill a bottle of aspirin. If you pick up each aspirin individually and deposit it directly in the bottle, you're doing an unbuffered, or raw, transfer. If you pick up the aspirin with your right hand and collect them in your left, then dump a bunch into the bottle at once (along with all of the dirt from your floor), you're doing a buffered transfer.[1]

Choosing Your Mode

Address disks (and many other devices) as raw or block by choosing the corresponding device node. Some programs expect to access raw devices, while others expect block devices. If a program opens */dev/sd1a*, it's accessing partition *a* on disk *sd1* as a block device. If it opens */dev/rsd1a*, it's accessing the exact same partition as a character device.

Regardless of the mode, the underlying hardware remains the same; the only thing that changes is how you exchange information with the device.

Device Attachment vs. Device Name

Not long ago, most disks were permanently affixed to a single physical location on the system. If your computer had two IDE buses, each with two hard drives, the operating system knew exactly where to find them, usually at */wd1* and */wd2*. A SCSI disk had a SCSI ID and a logical unit number (LUN), and changing them required rebooting the computer. Traditionally, you could use the disk's location in the system to identify the disk. For example, a booting i386 computer would find the root partition by looking for the hard drive attached to the first port on the first IDE controller, finding the *a* partition on that disk, and reading the filesystem table from that disk. You could go into the BIOS to tell the computer to look for the root partition on a different disk, but the computer still identified the disk by where it was physically attached to the computer.

---

1. If it's buffered aspirin, then you're doing buffered buffered aspirin transfers. But let's not go there.

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

Today, disks can appear and disappear from multiple locations on the system. For example, you might attach and remove several flash drives as needed, or hot swap Serial Attached SCSI (SAS) or Serial ATA (SATA) drives from bus to bus. Physical location is no longer a safe way to identify a disk. While */dev/sd0* is the device node for the first SCSI disk, you cannot assume that the disk currently attached to the first SCSI port is the same disk that was plugged in there the last time the system booted. OpenBSD labels actual disks with unique IDs, as discussed in the next section.

DUIDs and /etc/fstab

All OpenBSD platforms use the disklabel to identify partitions and other information about a disk. When you label a disk (as we did in Chapter 3 and will do by hand later this chapter), `disklabel` adds a *disklabel unique identifier*, or DUID, to the disk label. The DUID is a unique hexadecimal number that lets OpenBSD identify a specific disk.

To find a disk's DUID, pass the device name to `disklabel` and look for the `duid` entry:

```
# disklabel sd0
...
duid: 55128c3700af5491
...
```

The disk currently attached as `sd0` has a DUID of `55128c3700af5491`. Even if you physically move the disk so that it becomes `sd9` or `sd18`, OpenBSD can use the DUID to uniquely identify this disk.

OpenBSD uses the filesystem table */etc/fstab* to map filesystems on a disk to mount points using either the disk location or the DUID. Each filesystem appears on its own line in */etc/fstab*, as shown here:

```
❶ 55128c3700af5491.b ❷none ❸swap ❹sw
55128c3700af5491.a / ffs rw 1 1
55128c3700af5491.k /home ffs rw,nodev,nosuid 1 2
55128c3700af5491.d /tmp ffs rw,nodev,nosuid 1 2
...
```

We'll focus on the first entry to explore what's going on here. The first field, `55128c3700af5491.b` ❶, is the location of the partition. Whereas older systems used the disk device name and the partition letter (such as */dev/sd0a*), newer systems can use the DUID, a period, and the partition letter (as in `55128c3700af5491.a`). By using DUIDs in the filesystem table, OpenBSD can always mount the same disk at the same location, no matter how it's attached.

The second field, `none` ❷, lists the *mount point*, which is the directory where the filesystem is attached to the directory tree. Every partition you can write files to is attached to a mount point (such as */usr*, */var*, and so on), with one partition being the root partition (/). Swap space uses a mount point of `none`.

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

Next, `swap` ❸, is the filesystem type. The standard OpenBSD partition uses type `ffs`, the UNIX Fast File System. Other options include, but are not limited to, `msdos` (Microsoft-style FAT partitions), `mfs` (Memory File System), and `cd9660` (CD).

The fourth field, `sw` ❹, shows the mount options used for this filesystem. I'll cover mount options in more detail in "FFS Mount Options" on page 135, but here are a few that frequently appear in */etc/fstab*:

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

`rw` The filesystem is mounted read-write.

`nodev` Device nodes are not interpreted.

`nosuid` `setuid` files are forbidden.

`noauto` OpenBSD won't automatically mount the filesystem at boot or when running `mount -a`. This option is useful for removable media drives that might not have media in them, such as CD and USB flash drives.

The fifth field indicates whether `dump(8)` should back up this filesystem. If this field is 0 (or absent), `dump` doesn't routinely back up the filesystem. Otherwise, the number given is the minimum dump level needed to back up the filesystem.

The last field is the *pass number*. It tells `fsck` when to check the filesystem during boot. Filesystems with a pass number of 1 are checked first, filesystems with a pass number of 2 are checked second, and so on. A pass number of 0 tells `fsck` to not check the filesystem during boot. If a filesystem doesn't have a pass number, it's equivalent to 0.

I strongly recommend using DUIDs in */etc/fstab* and anywhere else, rather than using device node names. While a device node name might change, a DUID will not.

MBR Partitions and fdisk(8)

Some hardware platforms have specific ideas about disk partitioning that differ from what OpenBSD expects. For example, the i386 and amd64 platforms expect to find MBR partitions on hard drives, and OpenBSD accommodates this quirk by putting its own disklabel partitions inside MBR partitions. We briefly touched on creating partitions during the installation process, but if you add hard drives to an existing system, you'll need to edit the MBR partition table by hand using `fdisk(8)`.

My particular test system has two hard drives: *wd0* and *wd1*. I think that *wd1* is completely blank but before I can use this drive, I need to verify that it is empty, and then create MBR partitions. While `fdisk` has all sorts of commands to edit disks, I find the simplest way is to use the interactive disk editor. Run `fdisk -e` and give it the device node for the new disk.

```
# fdisk -e wd1
Enter 'help' for information
fdisk: 1>
```


©2013, Michael W. Lucas

The editor is minimal, but lets you view, add, remove, and edit MBR partitions. If you forget the commands at any time, entering `help` will print out all the commands `fdisk` supports.

Viewing MBR Partitions

To see the MBR partitions on the current disk, enter **`print`** or **`p`**. Here's an example:

```
fdisk: 1> print
Disk: wd1       geometry: 2088/255/63 [33554304 Sectors]
Offset: 0       Signature: 0xO
            Starting         Ending         LBA Info:
 #: id      C   H   S -      C   H   S [       start:        size ]
-------------------------------------------------------------------------------
 0: 00      0   0   0 -      0   0   0 [           0:           0 ] unused
 1: 00      0   0   0 -      0   0   0 [           0:           0 ] unused
 2: 00      0   0   0 -      0   0   0 [           0:           0 ] unused
 3: 00      0   0   0 -      0   0   0 [           0:           0 ] unused
```

The first line shows the disk geometry (as discussed in Chapter 2). Every value in this disk's MBR table is set to 0, meaning that it has no configured partitions.

Adding and Removing Partitions

Say we want to create an MBR partition on this disk. I habitually use partition 0, but the OpenBSD installer usually uses partition 3. The specific number you pick doesn't matter unless you want multiple MBR partitions on the disk.

To edit a partition, enter **`edit`** or **`e`** followed by the partition number. For example, to edit partition 0, enter the following:

```
fdisk: 1> e 0
            Starting         Ending         LBA Info:
 #: id      C   H   S -      C   H   S [       start:        size ]
-------------------------------------------------------------------------------
 0: 00      0   0   0 -      0   0   0 [           0:           0 ] unused
❶ Partition id ('0' to disable)  [0 - FF]: [0] (? for help) a6
Do you wish to edit in CHS mode? [n]
❷ offset: [0]
❸ size: [0] *
```

WARNING *Conveniently, `fdisk` prints the current information on this MBR partition. Make sure it's the partition you think it is before you muck it up.*

First, at ❶, set a partition ID. This is a label indicating what kind of filesystem will be on the disk. OpenBSD uses partition ID `a6`, so enter that.

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

The offset at ❷ is the number of sectors from the beginning of the disk to the start of the partition. We want to use this entire disk for OpenBSD, so set it to `0`.

Finally, the size at ❸ is the number of sectors the MBR partition fills. There is no need to copy the number of sectors in the disk here; OpenBSD `fdisk` uses * to mean "all free space."

Now print the MBR table again to check your work.

```
fdisk:*1> p
Disk: wd1       geometry: 2088/255/63 [33554304 Sectors]
Offset: 0       Signature: 0x0
            Starting         Ending         LBA Info:
 #: id      C   H   S -      C   H   S [       start:        size ]
-------------------------------------------------------------------------------
 0: A6      0   0   1 -   2088 167  63 [           0:    33554304 ] OpenBSD
 1: 00      0   0   0 -      0   0   0 [           0:           0 ] unused
 2: 00      0   0   0 -      0   0   0 [           0:           0 ] unused
 3: 00      0   0   0 -      0   0   0 [           0:           0 ] unused
```

Notice that the entry for partition 0 is type A6 and extends from cylinder 0, head 0, sector 1, to cylinder 2088, head 167, sector 63. It fills 33,554,304 sectors—the same as the number of sectors in the disk. This MBR partition fills the entire disk.

If you had recycled this disk from another operating system, it would probably have a partition already on it. To remove a partition, edit the partition and set its partition ID to 0.

Making a Partition Bootable

In order to boot from a hard drive, you'll need to mark a partition as active. Use the `flag` command and a partition number to do this.

```
fdisk: 1> flag 0
Partition 0 marked active.
```

Include this hard drive in your BIOS boot order, and the computer should try to boot from it. Simply marking a partition as active doesn't mean that the computer *can* boot from it; however, you will still need a kernel, boot loader, and all the other things that go into bootstrapping a computer.

To mark a partition as no longer active, delete and re-create it. (There is no `unflag` command.)

Exiting fdisk

Once you're satisfied with your work, enter **`quit`** or **`q`**, and `fdisk` should write the new MBR table to disk and exit. If you changed your mind, and don't want to make any changes, enter **`abort`** or **`exit`**, and `fdisk` should exit without saving changes to the MBR partition table.

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

Labeling Disks

OpenBSD uses `disklabel` to set up partitions on all hardware platforms. We used `disklabel(8)` as part of the installation process, but you need to partition new disks before you can use them. (You can also use `disklabel` to back up, restore, and duplicate partition tables.)

Viewing Labels

To view the current disklabel, just give the disk name as an argument. Here's how to see the disklabel of the empty disk from the previous section:

```
  # disklabel wd1
❶ # /dev/rwd1c:
  ...
❷ duid: 0000000000000000
  ...
  16 partitions:
  #                size           offset  fstype [fsize bsize  cpg]
❸  c:         33554304                0  unused
```

This looks much like the disklabel we saw in Chapter 2, with a few critical differences.

First, note the device at ❶. The `disklabel` command accesses the raw device, but you should use the block device at the command line.

This label at ❷ has no DUID. This is the default empty disklabel. We will generate a DUID later.

At ❸, we see that this disk has only one partition, *c*, which represents the entire disk. You could create and use a filesystem on partition *c*, but it's not standard practice to do so.

Creating Disklabel Partitions

The simplest way to create partitions is to use the same interactive disklabel editor that we used to install OpenBSD. Give the disklabel editor the `-E` flag and the disk name:

```
# disklabel -E wd1
Label editor (enter '?' for help at any prompt)
>
```

Now you can add, remove, and edit partitions, just as in Chapter 3.

Throughout the rest of the book, we'll edit disklabels as needed to change partition and filesystem characteristics.

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

Backing Up and Restoring Disklabels

Before messing with a disk, back up its disklabel so that you can fall back to the old label if you screw up. You can back up the disklabel with this command:

```
# disklabel wd1 > wd1.disklabel.saved
```

To apply a saved disklabel to a disk, give `disklabel` the `-R` flag, the disk device, and the label file:

```
# disklabel -R wd1 wd1.disklabel.saved
```

This writes the saved label to the disk. You can use saved disklabels to duplicate partitioning across identical disks.

Now that you have partitions, let's put a filesystem on them.

The Fast File System

OpenBSD's filesystem, FFS, is an improved version of the filesystem shipped with BSD 4.4. FFS is sometimes called UFS (for Unix File System), and many system utilities still use UFS.[2]

FFS is designed to be fast, reliable, and able to handle the most common situations effectively while still supporting weird configurations. By default, OpenBSD tunes FFS for general use, but you can optimize it to fit your needs—whether you need to hold trillions of tiny files or a half dozen 30GB files. You don't need to know much about FFS internals, but you should at least understand blocks, fragments, and inodes.

FFS Versions

The original FFS was written in the 1980s and included hard-coded limits that were ample for the day. Filesystems could have up to 2^{31-1} blocks, or just under a terabyte (TB). In 1983, a 1TB filesystem was unthinkable. In 2013, 1TB drives are cheap.

For larger file systems, we have FFS version 2. FFS2 can support filesystems up to 8 zettabytes—unthinkable by 2013 standards. (FFS2 is likely to reach other limits before hitting the filesystem size limit, mind you.) OpenBSD supports both FFS and FFS2.

The i386 and amd64 boot floppies support only FFS, not FFS2. The installation CD, however, supports both. Most machines that need to boot from floppy don't need FFS2, and probably don't have a BIOS that can support 2TB drives anyway. The filesystem creation program `newfs(1)` is smart

2. OpenBSD is not the only operating system that still uses the BSD 4.4 filesystem or a descendant thereof. If a Unix vendor doesn't specifically tout its "improved and advanced" filesystem, it's almost certainly running a derivative of FFS.

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

enough to use FFS2 on filesystems large enough to need it, so for most installations, you don't need to worry about the difference between FFS and FFS2.

NOTE *In the exceedingly unlikely event that you actually require FFS2 on a machine that must be installed via floppy, be sure to format the critical system partitions of root (`/`), `/var`, and `/usr` as FFS, not FFS2. Use FFS2 only for partitions that are not critical to the system. Otherwise, you won't be able to use the installation disk for upgrades or emergency repairs.*

Blocks, Fragments, and Inodes

Both FFS and FFS2 are managed through blocks, fragments, and inodes. This arrangement isn't unique to FFS and FFS2; filesystems such as NTFS use data blocks and index nodes, too. The indexing system used by each filesystem is largely unique, however.

Blocks

Blocks are sections of disk that contain data. Files are placed in one or more blocks. OpenBSD's FFS uses a default block size of 16KB, or eight times the fragment size, whichever is smaller. Not all files are even multiples of 16KB, so leftover bits go in *fragments*. A fragment is one-eighth of the block size, or 2KB by default. A 20KB file fills one block and two fragments.

Inodes

Inodes, or index nodes, contain basic data about files, such as the file's size, permissions, and the list of blocks that contain the file. Collectively, the data in an inode is known as *metadata*, or data about data.

Superblocks

You'll also see references to *superblocks*, which are blocks that contain vital information about the filesystem's size and specifications. Superblocks are so important that FFS makes many backup copies of them. If you need to meddle with superblocks, you've probably done something wrong or your filesystem is FUBAR.

Creating FFS Filesystems

Use `newfs(8)` to create FFS and FFS2 filesystems and make sure that the disk has a disklabel. The `newfs` command takes one argument: the partition device node.

```
# newfs wd1a
/dev/rwd1a: 16383.9MB in 33554304 sectors of 512 bytes
81 cylinder groups of 202.47MB, 12958 blocks, 25984 inodes each
super-block backups (for fsck -b #) at:
```


Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

```
 32, 414688, 829344, 1244000, 1658656, 2073312, 2487968, 2902624, 3317280,
3731936,
 ...
```

You'll see details about the filesystem size, how many blocks it includes, and so on. The location of each superblock backup is printed as `newfs` proceeds. (When computers and disks were much slower, this told the operator that the computer was actually doing something and hadn't seized up.)

The partition size determines which filesystem `newfs` uses. Partitions smaller than 1TB are formatted with FFS; larger partitions with FFS2. If you want to specify a particular filesystem format (yes, you can even specify the old-fashioned 4.3BSD format if you like), use the `-O` flag. It makes no sense to demand an FFS filesystem on a large partition, but you might have a reason to use FFS2 on a small partition.

```
# newfs -O 2 wd1a
```

If you think you need to specify which filesystem format to use on a new filesystem, you're probably wrong.

FFS Mount Options

OpenBSD can handle FFS partitions in several special ways, controlling what sorts of changes the filesystem supports and what sorts of files may exist. These are called *mount options*. You can specify mount options either when you mount partitions on the command line, as we'll discuss in "Mounting and Unmounting Partitions" on page 140, or in */etc/fstab*.

Mount Options and /etc/fstab

Specify a filesystem's mount options in a comma-separated list in the fourth field of the filesystem's */etc/fstab* entry. For example, here's an */etc/fstab* entry for the partition that contains my */home* directory:

```
244f6d3acd6374ad.k /home ffs rw,nodev,nosuid,softdep 1 2
```

I've specified the options `rw` (read-write), `nodev` (device nodes forbidden), `nosuid` (`setuid` programs forbidden), and `softdep` (soft updates). I'll cover these and other common mount options, and explain why you might want to use them.

Read-Only Mounts

If you only want to read the contents of a partition, and never write to it, you can mount the partition as *read-only*. In most cases, this is the safest way to mount a disk because you cannot alter the data on the disk or write any new data. If a filesystem should never change, mounting it read-only might make sense.

Read-only mounts are especially valuable when a particular filesystem is damaged. While OpenBSD won't let you perform a standard read-write

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

mount on a damaged or dirty filesystem, it can often mount those filesystems read-only. This gives you a chance to recover some data from the partition. (Not a large chance, but a chance.)

To mount a filesystem read-only, use the option `rdonly` or `ro`.

Read-Write Mounts

If you want to both read from and write to the disk, you'll want to mount the partition as *read-write*. By default, OpenBSD mounts all partitions as read-write.

Use the option `rw` to explicitly configure read-write mounts.

On modern hardware, I recommend using soft updates in conjunction with read-write mounts.

Synchronous Mounts

Using a synchronous mount is the safest way to mount a filesystem. OpenBSD can read data from a synchronous-mounted partition as fast as the hardware permits. Whenever you write to the disk, however, the kernel feeds a chunk of data to the disk, waits to receive confirmation that the disk has accepted the data and written it to disk, and then tells the program that requested the write that the data is now on disk.

You should know that even if you're using a synchronous mount, most hard drives lie about whether they have actually written the data to disk. These drives perform *write caching*, where writes are cached in a small flash or RAM buffer on the disk itself before the drive actually writes the data. This raises the question: Is a synchronous mount really synchronous? Hard drive vendors usually claim that in the event of a power failure, these disks retain just enough power to write the cache to disk.

Although they provide the greatest data integrity in the case of a crash, synchronous mounts are slow. You might use synchronous mounts when data integrity is crucial, but in most cases, it's overkill and you have little ability to verify that the mount is truly synchronous.

Activate synchronous mounts with the `sync` keyword.

Asynchronous Mounts

To write data quickly, but with a higher risk of data loss, mount partitions asynchronously. When using asynchronous mounts, the kernel informs software that all disk writes are successful before the disk confirms that the data was written. This is fast, but a system failure can leave inconsistent data on your disk.

Asynchronous mounts are useful when restoring a filesystem from backup, because if you get a power failure halfway through the restore procedure, you'll need to start over anyway. Don't use asynchronous mounts in production if you care about your data or would object to re-creating the filesystem.

Activate asynchronous mounts with the `async` keyword.

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

Soft Update Mounts

Soft update mounts organize and arrange disk writes so that filesystem metadata remains consistent at all times. This gives performance similar to that of an asynchronous mount with the reliability of a synchronous mount. While that doesn't mean that all data will be written to disk—a power failure at the wrong moment will result in lost data—using soft updates prevents a lot of filesystem integrity problems caused by that lost data. It's not the default because some older, smaller hardware doesn't have enough memory to support it, but if you're using modern i386 and amd64 hardware, I recommend enabling soft updates for all FFS partitions.

To mount a filesystem with soft updates, use the `softdep` option.

"Don't Track Access Time" Mounts

FFS records the last time a file was read, executed, or otherwise viewed. Updating these access times consumes a small but measurable amount of disk I/O and performance. You can use the `noatime` mount option to tell OpenBSD to not update the access time on any file.

Using `noatime` makes sense on laptops, where minimizing power usage is critical. If you're tempted to use this option on your server to squeeze out a little extra performance, you should buy a faster disk instead. Some software, such as the Mutt mail client, will break if run on filesystems mounted `noatime`.

No Device Nodes Permitted Mount

By using the `nodev` mount option, you can tell OpenBSD to not interpret any device nodes on any given filesystem. Intruders can try to create "rogue" device nodes and use them to write files or attack the network, but if the kernel won't recognize those device nodes, it cuts off this whole category of attacks.

This type of mount is also useful if you have hard drives from multiple operating systems on your computer. For example, if you dual-boot OpenBSD and Linux on your computer, but you don't want to accidentally access a Linux device node when using OpenBSD, the `nodev` option will prevent you from doing so. (You might think you would notice that you had typed */linux/dev/hda* rather than */dev/wd1*, but never underestimate your ability to screw up.) In most cases, the partition containing */dev* is the only one that should contain device nodes.

Execution Forbidden Mounts

The `noexec` mount option prevents any binaries on the partition from being executed. Mounting */home* with the `noexec` option helps prevent users from installing and running their own programs, but for it to be effective, you'll need to make sure users can't install binaries in any shared areas, such as */tmp* and */var/tmp*.

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

Note that forbidding execution of binaries doesn't prevent users from running interpreted scripts from that partition. Maybe the users can't run a compiled C program, but if they can run `perl $HOME/rootkit.pl`, then `noexec` won't slow them down very much.

setuid Forbidden

The `nosuid` option disallows `setuid` behavior from programs on this filesystem. Many partitions should not have `setuid` files, and setting this is an easy way to disrupt them. OpenBSD sets this on partitions such as */home* and */tmp* by default. You must carefully place this option on all user-writable filesystems for it to prevent undesired behavior.

Do Not Automatically Mount This Filesystem

`noauto` isn't actually a mount option, but rather a way of telling OpenBSD to not mount a given partition listed in */etc/fstab* at boot. I frequently make */etc/fstab* entries for removable media drives, but the system should not try to mount these at boot. The boot will hang if a partition required by */etc/fstab* is not available, and I don't want my computer to refuse to boot just because I unplugged a flash drive.

Filesystem Integrity

Both versions of FFS go to a great deal of trouble to ensure that the data on disk is correct and intact. The blocks that contain a file should be recorded in an inode, the inodes should all be referenced by directory entries, and so on. When you remove a file, all references to that file should be removed.

After a system failure, however, data might not be consistent. Metadata might reference blocks that were previously erased; a file might be in a different location than the inode record specifies; and the filesystem might have all kinds of references pointing to things that have moved, changed, or disappeared. These inconsistent, or *dirty*, filesystems cannot be trusted and must be rationalized, or *cleaned*, before you can mount them read-write. If you mount a dirty filesystem read-only, it might only panic your system, but if you force OpenBSD to mount a dirty filesystem read-write, you will damage the dirty filesystem even more.

At boot, OpenBSD performs a minimal inspection and cleaning, or *preening*, of the filesystems and will automatically correct any minor problems found. If preening cannot fully clean the filesystem, the boot will hang until you intervene.

When confronted with a dirty filesystem, you have a few options: use the filesystem checking tool `fsck(8)`, debug the filesystem with `fsdb(8)` and `clri(8)`, or throw the filesystem away and run `newfs(8)`. Most of the time, you'll attempt to repair the filesystem with `fsck`. Using `fsdb` successfully requires more knowledge about FFS innards than I possess, so I recommend it to only those who really want to develop an in-depth knowledge of FFS and have

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

a whole bunch of time to devote to it. Rebuilding the filesystem with `newfs` destroys everything on the filesystem, but it's a decent choice for partitions that contain only ephemeral data, such as */usr/obj*.

You can use `dump(8)` to copy the damaged filesystem before trying any of the repairs. This gives you the option to fall back to the current state if attempts at repairing the disk fail. (If you have to do this, though, you should probably reevaluate your backup strategy.)

Running fsck

If you try to mount a dirty filesystem either at boot time or during routine operation, you'll see a message that looks like this:

```
/dev/rwd1a: UNEXPECTED INCONSISTENCY; RUN fsck_ffs MANUALLY
```

The `fsck(8)` program is a frontend for several filesystem-specific integrity-checking programs. When you run it, `fsck` identifies the type of filesystem and calls the correct integrity checker for you. Run `fsck` by giving it the device name of the filesystem you want to check:

```
# fsck /dev/wd1a
```

You can use either the raw or cooked device name; `fsck` is smart enough to use the raw node even if you give the cooked device name.

Examining the filesystem can take quite a while, so be patient.

When run on a dirty filesystem, `fsck` will probably find a number of problems: blocks that have become disassociated from their inodes, inodes that reference empty blocks, and so on. It can often make a good guess as to how everything fits together.

When `fsck` finds a problem that it isn't absolutely sure about, it will suggest a fix and ask if you want to make the change. If you answer `y`, `fsck` makes the change. If you answer `n`, `fsck` leaves the filesystem unchanged. If you tell `fsck` not to make the change it suggests, the filesystem will still be dirty, and you'll need to fire up `fsdb` or `clri` and make the change you think more appropriate.

Sometimes, `fsck` can't identify the name or directory of a file recovered from a damaged filesystem. These files go into the partition's *lost+found* directory (for example, */usr/lost+found*). You'll need to use programs such as `grep` and `strings` to try to identify these files by their contents.

Blindly Trusting fsck

Those of us who lack the skills to debug a filesystem find ourselves in a difficult situation, where we can either accept that `fsck(8)` knows what's best or just restore from backup. If your filesystem was performing a lot of disk I/O just before system failure, `fsck` might need to make dozens or hundreds of changes. You could spend an hour sitting at the console pressing `y` repeatedly.

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

If you decide to trust `fsck` and hope it's right, run `fsck -y`. This means "answer `y` to every question." You might wind up with the entire contents of the filesystem in the *lost+found* directory, or you might lose every file on the filesystem. But unless you're intimately familiar with the innards of FFS, you would need to restore from backup anyway.

If you run `fsck` and realize partway through that you would like to answer `y` to all the questions that follow, enter `F`. That tells `fsck` to answer `y` to all remaining questions.

At the end of the procedure, you've either recovered your system or need to restore from backup.

What's Currently Mounted?

While performing routine work, inevitably you'll need to check which disks are currently mounted and which are not. To see a list of all mounted filesystems and their mount options, run `mount(8)` without any options:

```
$ mount
/dev/wd0a on / type ffs (local)
/dev/wd0k on /home type ffs (local, nodev, nosuid)
/dev/wd0d on /tmp type ffs (local, nodev, nosuid)
/dev/wd0f on /usr type ffs (local, nodev)
/dev/wd0g on /usr/X11R6 type ffs (local, nodev)
/dev/wd0h on /usr/local type ffs (local, nodev)
/dev/wd0j on /usr/obj type ffs (local, nodev, nosuid)
/dev/wd0i on /usr/src type ffs (local, nodev, nosuid)
/dev/wd0e on /var type ffs (local, nodev, nosuid)
```

Both FFS and FFS2 partitions show up as `type ffs`. The word `local` means that the partition is on a physical drive attached to this machine. We covered the various mount options (`nodev`, `nosuid`, and so on) earlier in this chapter.

Note that `mount` displays the device node mounted at each partition, not the DUID. If you want to see the DUID of a disk, check the disklabel.

Mounting and Unmounting Partitions

To attach filesystems to your directory tree, or *mount* them, use `mount(8)`. If you've never manually mounted filesystems before, boot your OpenBSD machine into single-user mode (see Chapter 5) and follow along.

In single-user mode, OpenBSD mounts only one partition: the root partition, which it mounts read-only. The root partition contains just enough of the system to perform basic setup, establish core services, and find the other filesystems.

Because filesystems other than the root are not mounted, their content is not accessible. Look in, say, */usr* on a system in single-user mode, and you'll find that it's empty. OpenBSD hasn't lost the files; it just hasn't mounted the partition containing those files.

To get any real work done in single-user mode, you probably need to mount other filesystems.

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

Mounting Standard Filesystems

To manually mount a single filesystem listed in */etc/fstab*, give mount(8) the name of the filesystem you want to mount. Here, we'll mount our */usr* partition:

```
# mount /usr
```

This mounts the partition exactly as described in */etc/fstab*, with all the options specified therein.

To mount all of the partitions listed in */etc/fstab*, give mount the -a flag:

```
# mount -a
```

All of your filesystems (except those not listed in */etc/fstab* and those with the noauto option) should now be mounted.

Mounting at Nonstandard Locations

Perhaps you must mount a filesystem at a location not specified in */etc/fstab*. I do this most commonly when adding a disk to a machine. To mount a partition at a location other than that specified in */etc/fstab*, or to mount a partition without an */etc/fstab* entry, give the partition device name and the mount point.

```
# mount /dev/sd0d /mnt
```

You must use the full path for the device node, not just the brief device node name.

Instead of the path to the device node, you could use the DUID, a period, and the partition letter, but on the command line, that's more painful than using the path to the device node.

Unmounting Partitions

To disconnect a filesystem from the directory tree, use umount(8) on a mount point. (Note that there is only one n in this command.) Here, we'll use umount to unmount our */usr* partition:

```
# umount /usr
```

You cannot unmount filesystems that are in use by any program. Even a command prompt in the mounted directory will prevent you from unmounting the partition.

To unmount all partitions except the root partition, pass umount the -a flag:

```
# umount -a
```

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

As programs almost certainly have files open on every partition, this probably works only in single-user mode. Note that you don't need to unmount all partitions to leave single-user mode.

Mounting with Options

Suppose you pull a disk from a decommissioned OpenBSD machine and you need to retrieve some files from it. You want to mount the disk read-only so that you don't change any of the files on the disk. To manually mount a partition with options not specified in */etc/fstab*, use the `-o` flag.

For example, if the disk shows up as */dev/sd0* and you want to mount partition *a*, run this command:

```
# mount -o ro /dev/sd0a /mnt
```

To prevent old software from running on your newer system, it might be a good idea to use some of the options we covered earlier, such as `noexec`, `nodev`, and `nosuid`.

How Full Is That Partition?

To get an idea how much free space remains on your partitions, use `df(1)`. This program displays the total number of filesystem blocks on each partition, how many blocks are in use, and how many blocks are free. It also gives you the percent in use.

One annoying thing about `df` is that it offers this information in 512-byte blocks by default. This was fine when disks were much smaller, but today, it's like measuring the distance of an airplane flight in yards. Some people have done this for so long that they automatically perform block transformations in the back of their mind.[3] For the rest of us, the `-h` flag tells `df` to provide human-readable output, such as megabytes or gigabytes, giving us something like this:

```
# df -h
Filesystem     Size    Used   Avail Capacity  Mounted on
/dev/sd0a     1005M   39.1M    916M     4%    /
/dev/sd0k     26.9G   27.0G   -104M    -1%    /home
/dev/sd0d      3.5G   12.0K    3.3G     0%    /tmp
...
```

You might wonder why the */home* partition in this example has negative free space. How is that possible? By default, FFS reserves 5 percent of each partition for moving files and reducing fragmentation. When you exceed 100 percent disk utilization, you begin tapping into this reserved space.

FFS performance degrades quickly when the partition is overfull. It's best to keep some free space on your disk so that FFS can defragment itself.

3. Hi, Henning!

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

You can reduce the amount of space FFS reserves, but doing so will impact performance. See tunefs(8) for instructions on how to shoot yourself in the foot.

What's All That Stuff?

When you see a partition is full, the obvious question is "What's filling up my disk?" Every hard drive I've ever owned has gradually filled up for no apparent reason. You can identify individual large files with `ls -l`, but recursively examining every directory in the filesystem is impractical and tedious (not to mention annoying).

To check the number of filesystem blocks used within each directory below the current directory, use `du(1)`.

```
$ du
164     ./.ssh
2       ./old
6       ./.mozilla/firefox/bcpuv16e.default/chrome
80      ./.mozilla/firefox/bcpuv16e.default/Cache/0/B0
354     ./.mozilla/firefox/bcpuv16e.default/Cache/0/31
28      ./.mozilla/firefox/bcpuv16e.default/Cache/0/7A
...
```

When I run `du` in my home directory, I get 700 entries; of those, 563 are related to some Mozilla tool. This kind of list intimidates the new sysadmin and makes the experienced sysadmin work too hard. Rather than cull through this list manually, tell `du` to show only the total for directories in the current directory, and then sort the output so that the largest directories appear first.

```
$ du -s * | sort -rnk 1
25224805  Dark_Shadows_Complete_Series
141104    mibs
14948     tarballs
4668      work
1864      pix
...
```

I now know why my */home* partition is full.

You can tell `du` to display human-readable values with the `-h` flag, but doing so will show values in a mix of gigabytes, megabytes, and kilobytes, making `sort` far less useful.

Setting $BLOCKSIZE

Many disk tools—including, but not limited to, `du(1)` and `df(1)`—display information in 512-byte blocks. If you're accustomed to working in blocks, you probably won't mind seeing them. If you're not used to working in blocks, however, they'll probably make you want to tear out your hair.

The environment variable BLOCKSIZE tells these programs to display information using blocks of a different size. If you set BLOCKSIZE to K, df and du will display totals in kilobytes. If you set it to M, these tools will show megabytes instead. Check your shell manual page or the dotfiles in your home directory for examples of setting environment variables.

Adding New Hard Disks

The OpenBSD installer walks you through formatting and partitioning your initial hard disks. If you need to add a disk to an existing system, however, you must run these commands yourself. The good news is that if you can install OpenBSD, you already know how to use the commands, and the only hard part is learning which commands to run.

I'll show you how to move */home* to a new disk as an example. You could create a new partition on your existing disk if you have some empty space, but that would eliminate the need for this example, so I'm going to pretend I never gave you that advice. (Also, moving partitions to a separate disk controller channel will improve performance.)

WARNING *Before touching anything involving disk partitioning or filesystems, back up your system. Verify that backup before starting. You have been warned.*

Creating an MBR Partition

The i386 and amd64 platforms require disks to have MBR partitions as well as OpenBSD partitions. A standard new disk needs a single OpenBSD MBR partition covering the entire disk. Passing the -i argument to fdisk does exactly this. Let's create a new MBR partition on wd1, our new disk:

```
# fdisk -i wd1
Do you wish to write new MBR and partition table? [n] y
Writing MBR at offset 0.
```

Once you have an MBR partition on your disk, you can create disklabel partitions.

Creating a Disklabel

All OpenBSD platforms use disklabel partitions. To activate the same disklabel editor we used during the install process, give disklabel the -E flag and the disk name:

```
# disklabel -E wd1
```


Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

This should look familiar from earlier in this chapter. Use the interactive disklabel editor to create your new partitions. For a single */home* directory, we'll use one large partition, *wd1a*. The new label should look like this:

```
#                size           offset  fstype [fsize bsize  cpg]
  a:         33543648               64  4.2BSD   2048 16384    1
  c:         33554304                0  unused
```

When you've finished editing partitions, check your work by printing the disklabel. This should also give you the DUID of the new disk.

When you're satisfied with the partitioning, use `newfs` to create a filesystem on the new partition:

```
# newfs wd1a
```

You're now ready to add the filesystem to your computer.

Moving Partitions

Moving data from one disk to another is slightly more complex than adding new partitions. You must first mount the new drive in a temporary location, copy files to that location, remove them from the old location, and mount the new drive in its previous home.

Our new */home* filesystem is on disk partition *wd1a*. The default "temporary mount" location is */mnt*, so mount it there. This is strictly temporary, so there's no need to mount it via the DUID or make an */etc/fstab* entry for this.

```
# mount wd1a /mnt
```

You can then use `tar(1)`, `cpio(1)`, or `dump(8)` and `restore(8)` to copy the files to the temporary location. Here, we copy everything in */home* to */mnt*.

```
# (cd /home && tar cf - . ) | (cd /mnt && tar xpf - )
```

You could also use `cp(1)` or `mv(1)` for this, but these commands don't guarantee that file permissions and ownership will copy intact. OpenBSD's versions of these programs have never given me errors when I copy or move files, but I've learned from other Unix-like operating systems that `tar` and `cpio` are both more reliable when moving entire file hierarchies. If you're using file flags for security (see Chapter 10), you must use `dump(8)` and `restore(8)` to retain those flags.

Using `tar` or `cpio` does not delete files from their original location. This means that if a user changes files in his home directory after you copy them but before you change the mount point, he will lose his changes as you shuffle disks around.[4]

Now update */etc/fstab* to reflect your new disk.

4. Presumably you warn your users before doing maintenance. Or at least *during* maintenance. Or . . . maybe afterward.

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas

Adding New Filesystems

Look at the disklabel for the new disk and get the disk's DUID. This new disk has a DUID of `fea9194ee78362d8`. Use the DUID and the partition letter to make an */etc/fstab* entry for your new partitions.

```
fea9194ee78362d8.a /home ffs rw,nodev,nosuid,softdep 1 2
```

You might want to keep the old partition available at a new location, such as */oldhome*.

If you're not sure about the mount options to use for your new partitions, the options `nodev`, `nosuid`, and `softdep` are generally safe. You probably want the partition mounted read-write (`rw`) as well.

Now unmount the old and mount the new.

```
# umount /home
# mount /oldhome
# mount /home
```

When you unmount a partition, `umount` doesn't check */etc/fstab*. You tell it to unmount a partition, and it unmounts that partition.

Stackable Mounts

OpenBSD filesystems are *stackable*, which means that you can mount one partition over another. The partition on top hides any files in the filesystem below.

Look at your system in single-user mode. By default, only the root partition is mounted. You can go look in the */home* directory, and it will be empty. There's no reason you can't put files in the */home* directory, even when */home* isn't mounted. Suppose you copy a couple of core files into */home* while in single-user mode, and then go into multiuser mode. All the usual partitions are mounted. If you then look in */home*, you won't find your core files.

What happened? Where did those files go?

The files are in the directory */home*, but on the root partition. The */home* partition is mounted above that directory, so the */home* partition obscures the files in the */home* directory on the root partition. To access those hidden files, you must unmount the */home* partition. Those hidden files continue to take up space on the root partition, however.

This happens more commonly when splitting a partition. For example, if you find that your */var* partition is too small, you might move */var/www* into its own partition on a separate disk. To free up space on the original */var*, delete the files you copied to */var/www*.

With the basics of filesystem management under your belt, you're now ready to look at some of OpenBSD's more interesting filesystem tricks.

Absolute OpenBSD, 2nd edition
©2013, Michael W. Lucas