FreeBSD Mastery: Specialty Filesystems

Lucas

Table of Contents

FreeBSD Mastery
Specialty Filesystems

Michael W Lucas

Tilted Windmill Press

FreeBSD Mastery: Specialty Filesystems

Copyright 2015 by Michael W Lucas (https://www.michaelwlucas.com).

All rights reserved.

Author: Michael W Lucas

BSD Daemon copyright 1988 by Marshall Kirk McKusick. All rights reserved.

Technical Review: Edward Tomasz Napierała

Copyediting: Lindy Lou Losh

Cover art: *Beastie sort de l'Opéra*, illustration copyright © 2015 Eddie Sharam, after *Le Sortie de l'opéra en l'an 2000*, 1882, by Albert Robida

ISBN-13: 978-0692610442

ISBN-10: 0692610448

Tilted Windmill Press

For Liz

Acknowledgements

I planned to spend six weeks writing this book. Six months later, when I finally typed the last words, I looked back at the plan and laughed maniacally. FreeBSD has an incredible variety of special-purpose filesystems, and getting a handle on them took far longer than expected (mainly because I'm not nearly as smart or efficient as I think I am). Some of these filesystems will solve problems you never knew you had. Some of them will make you glad you don't have the problems these filesystems solve.

As overdue as this book is, it would have been completely impossible without all the people who offered advice, suggestions, and feedback. These people include John W. De Boskey, Lars Engels, Peter Jeremy, Alex Kozlov, Marie Helene Kvello-Aune, Ruslan Makhmatkhanov, Ravi Pokala, and Dag-Erling Smørgrav, all of whom were kind enough to review this manuscript before it reached you.

I specifically want to thank John Hixson for patiently helping me sort out the ~~madness depravity~~ complexities of the NFSv4 Access Control Lists.

Finally, Edward Tomasz Napierała, author of many of FreeBSD's newer filesystem features, carefully reviewed this book. Trasz has the most charming way of telling me how wrong I am, and he had frequent opportunity to simultaneously exercise that talent and practice patience.

This book was written on hardware from iXsystems, using test systems from iXsystems—meaning that I didn't have any hardware problems. Which was good, because I had quite a sufficient number of problems understanding all these filesystems…

Chapter 0: Introduction

Storage: a hole with no bottom, into which you pour data.

The last 50 years of computing have seen countless types of data storage hardware. Modern flash drives improve on spinning disks, which in turn improved on punch cards, which improved on tape, which was a lot better than carved marble tablets. BSD operating systems, and FreeBSD in particular, have been around for most of that time, and at one time had ways to utilize many different data storage methods. Support for that hardware remains long after the hardware was in common use. Even paper tape support lingers in the bcd(6) command.

Once you pick a physical medium, though, you have to decide how you're going to use it. Different operating systems have evolved entirely different filesystems to stuff data onto each sort of media. Your FreeBSD installation might need to interoperate with all of these, and more.

Then there are filesystems developed for special purposes, such as devfs and the Memory File System. The network opens up many storage possibilities, such as iSCSI and the Network File System and various replication technologies.

To truly master FreeBSD, you need to understand many of these. That's where this book comes in. You won't find information about GEOM and the Unix File System here, or about ZFS, or about how to implement filesystems in code. Those topics have their own books. This book is for all the other filesystems, storage methods, and data storage tricks.

Prerequisites

As this book is for filesystems meant for special situations, it has perhaps the widest range of prerequisites of any book I've ever written.

While the individual topics in this book aren't difficult, they presume you understand FreeBSD's GEOM storage subsystem. You don't need to understand how to write storage classes, but I assume you're familiar with commands like glabel(8) and `geom show`. Partitioning is another complex topic you need to bring to the table. Some filesystems assume a disk uses Master Boot Record (MBR) partitions, while others work only on Globally Unique ID (GUID) Partition Tables (GPT). Similarly, you'll need to understand the basics of the Unix File System (UFS), including critical system files like */etc/fstab*. We'll also cover some ZFS issues, where ZFS interoperates with other filesystem attributes.

This book pulls in knowledge from the whole spectrum of systems administration practice. If you want to use FreeBSD's iSCSI support, you'd best know something about TCP/IP and storage hardware. Using the Filesystem in Userspace (FUSE) support with the SSH module means you should understand starting and stopping services as well as SSH.

Filesystems require support in the kernel. That support might be included in the GENERIC kernel or a kernel module. You need to understand how to load, unload, and view kernel modules.

If you're sketchy on any of these prerequisites, you can find extensive documentation online or in other books.

This book addresses FreeBSD 10.2 and newer. Many topics also work on older releases, but FreeBSD 10 includes several new filesystem features, such as FUSE and iSCSI. Those features have grown their own features as FreeBSD 10 has progressed.

Finally, any complex work with filesystems requires shell scripts. This book includes several to get you started, but you'll need to modify them to fit your environment. For your convenience, the included scripts are also available at the author's GitHub site (https://github.com/mwlucas).

FreeBSD Mount Commands

We normally use the `mount` command to attach a filesystem to the directory tree. Using multiple filesystems changes how you use mount(8).

The `mount` command assumes that any local disk partitions use UFS. If you try to mount a different filesystem you'll get an error.

```
# mount /dev/da4s1 /mnt
mount: /dev/da4s1: Invalid argument
```

The device */dev/da4s1* exists, but it contains neither a UFS nor a UFS2 filesystem. You need to tell mount(8) what kind of filesystem this partition contains. Use either the `-t` command-line argument or a filesystem-specific `mount` command.

Every filesystem has a name, listed either in mount(8) or in the filesystem manual page. (Every filesystem has a manual page, in section 5). Give the filesystem name as an argument with `-t`.

```
# mount -t msdosfs /dev/da4s1 /media
```

Many filesystems (but not all) also have a filesystem-specific version of the mount command, which you could use instead.

```
# mount_msdosfs /dev/da4s1 /media
```

If a non-UFS filesystem has an entry in */etc/fstab*, mount(8) can mount that filesystem by mount point. We'll see example */etc/fstab* entries throughout this book.

```
# mount /media
```

Unmount everything with umount(8). The filesystem type doesn't matter. Give it a directory with something mounted there, and `umount` will gleefully disconnect it.

About This Book

We'll start with filesystems you might find locally attached, such as the Linux filesystem, ISO 9660 (used for CDs), and removable media. Then we'll cover logical filesystems, which exist only in the memory of a running system. This includes memory filesystems, devfs(5), union mounts, and more. We'll proceed to network filesystems, including NFS, iSCSI, and CIFS. Finally we'll discuss the automounter and FUSE, glues you can deploy to hold your storage together.

Chapter 1, "*Foreign Filesystems,*" covers filesystems from other operating systems. FreeBSD can access filesystems such as Microsoft FAT and certain types of Linux EXT and ReiserFS, to varying degrees. It can also access cross-platform filesystems like those on CD-ROMs and Blu-ray disks.

Chapter 2, "*devfs,*" discusses the device filesystem used for */dev*. FreeBSD's device filesystem dynamically manages device nodes. You can control how the system manages hardware and make FreeBSD take action when hardware appears or disappears.

Chapter 3, "*Namespace Filesystems,*" covers informative logical filesystems like the process filesystem. FreeBSD also includes special-purpose filesystems that offer views into file descriptors and Portable Operating System Interface (POSIX) messaging queues.

Chapter 4, "*Rearranging Filesystems,*" describes null mounts and union mounts, two ways to reuse and redirect traditional filesystems.

Chapter 5, "*Memory Filesystems,*" gives an overview of two different memory-based filesystems. You'll learn when to use each, as well as using memory disks to view disk images.

Chapter 6, "*Network File System,*" discusses the venerable NFS protocol. FreeBSD supports NFS from the old-fashioned version 2 up to NFS version 4.1.

Chapter 7, "*Common Internet File System Client,*" covers FreeBSD's built-in support for accessing file shares served by Microsoft Windows-style CIFS servers.

Chapter 8, "*iSCSI,*" discusses management of Internet SCSI, or accessing disk-like devices over IP. FreeBSD includes a high-performance iSCSI initiator and target.

Chapter 9, "*GEOM_GATE and HAST,*" gives an overview of FreeBSD's custom methods for sharing disk-like devices.

Chapter 10, "*Networked Disk Failover,*" discusses creating highly available storage with tools like iSCSI or HAST, building on top of the Common Address Redundancy Protocol (CARP).

Chapter 11, “*NFSv4 Access Control Lists*,” covers the industry standard extended permissions scheme created for the latest version of the Networked File System.

Chapter 12, “*Filesystem Glues*,” discusses ways to simplify systems administration through filesystem-related tools. FUSE, or Filesystems in User Space lets you use third-party modules to access unsupported filesystems, or transform data so that it’s presented as a filesystem. Finally, FreeBSD can automatically mount removable and dynamic filesystems, a very useful feature for workstations.

This book doesn’t cover tape storage. Or punch cards. They’re too special even for a book on specialty filesystems.

Chapter 1: Foreign Filesystems

FreeBSD can read and write filesystems used by other operating systems. You can grab disks from non-FreeBSD machines, attach them to your FreeBSD host, and read files from them. As far as this book is concerned, any permanent physical storage that uses a filesystem other than UFS or ZFS is a *foreign filesystem*. FreeBSD supports several foreign filesystems, but they might behave differently on FreeBSD than on their native operating system. Linux filesystems don’t support FreeBSD-style file flags, and FAT-derived filesystems don’t support file permissions or ownership.

This chapter discusses the various Microsoft FAT filesystems, the ISO 9660 standard used by CDs, the Universal Data Format used by large removable storage, and the Linux EXT and ReiserFS filesystems. We’ll start with the basics of using removable media.

Removable Media

Modern computers have many sorts of removable storage media, from CDs and USB devices to entire hard drives. Some of you have floppy disks lurking around, probably for reflashing antediluvian embedded devices.

Working with removable media is almost exactly like using permanently attached storage, but slightly more complicated. Any one piece of removable media is probably used only occasionally, though, and sometimes doesn't have a useful—or any—filesystem or partition table.

Removable Media Device Node

Using removable media requires knowing the media's device node. The device node depends on the type of device. USB drives, both spinning and flash, show up as a `/dev/da` device. Optical disks like CDs and Blu-ray appear as `/dev/cd`, even if they're USB devices. Floppy disks are `/dev/fd0`.[1] The hard part is figuring out the device number. If you have only one CD device, it's almost certainly `/dev/cd0`, but hard drives are more difficult.

The easiest way to identify the device node of a removable storage device is to check `/var/log/messages` or the console when you plug the device in. Some people, like the tech editor of this book, prefer to read through `geom disk list`. The kernel will log the name assigned to the new device.

Once you identify the device node, check for partitions on the device. Some smaller removable storage, like floppies and small USB drives, put the filesystem straight on the physical device. Others use partition tables. The easiest way to check for partitions is to look for partition device nodes, but tools like `gpart show` also work.

The mount point can be any existing directory on the system. FreeBSD includes a `/media` directory intended for removable media, and it also offers `/mnt` for whatever random short-term mount you need. You can mount a partition on any directory you like, however.

Attach the device or partition to the directory tree point with mount(8).

```
# mount /dev/fd0 /mnt
```

Use the proper mount(8) flags or mount program for the type of filesystem on the device.

Normally, only root can mount filesystems. If you want to grant unprivileged users the right to mount removable media, set the sysctl vfs.usermount to `1`.

Removing Media

Removing the hardware beneath a mounted filesystem can cause data loss or program crashes, and can even panic the system, depending on the disk’s filesystem and what was happening on the disk. To safely remove media from your system, first unmount all filesystems on the media. Optical trays won’t open until the disc is unmounted, which is a helpful reminder, but nothing prevents you from yanking out a USB flash drive.

Using Floppy Disks

Floppy disks work just like any other removable media, except that they must be formatted before use. This low-level format prepares the disk to receive a filesystem. Floppies do not normally use partition tables. While the floppy disk factory almost always formats the disks before shipping them out, perfectly good floppies often need reformatting.

Start by performing a low-level format with fdformat(1). This program requires only two arguments: the size of the floppy and the device name. Here I format a standard 1.44 MB floppy.

```
# fdformat –f 1440 /dev/fd0
Format 1440K floppy ‘/dev/fd0.1440’? (y/n): y
```

The fdformat(1) command runs more slowly than newfs(8). Go get a cup of coffee and come back.

At this point you can create a filesystem on the floppy.

You cannot run `fdformat` on a USB floppy drive. If a floppy disk needs reformatting and you have a USB floppy drive, throw the disk away.

MSDOS Filesystems

The File Allocation Table (FAT) is probably the lowest common denominator of filesystems. FAT dates back from the Microsoft DOS days, but has been updated over the years to support more modern systems.

Today, FAT is mostly used for removable media such as USB flash drives and, if you're in a *very* dark place, floppy disks. FreeBSD has excellent support for reading, writing, and creating the various versions of FAT, lumping them all together as *msdos* filesystems.

Reading MSDOS Filesystems

Tell mount(8) that a FAT filesystem is of type *msdos*, or use the mount_msdosfs(8) command to mount a FAT partition.

```
# mount -t msdosfs /dev/da4s1 /media
```

You don't need any special arguments to mount older FAT types such as FAT16 or FAT12; FreeBSD automatically detects the FAT version on the media.

You can mount FAT devices through their label names as they appear in */dev/msdosfs*.

Creating MSDOS Filesystems

Use newfs_msdos(8) to create a FAT filesystem. For most modern storage media, you'll put the filesystem on a partition. Floppy disks do not use partition tables, and some older entertainment devices that use FAT don't support partition tables. Put the filesystem right on the raw disk in these cases.

Here I create an MBR partition table and a FAT partition on the 2 GB USB flash drive da4. (I chose to use MBR rather than GPT so that this drive is compatible my old television.)

```
# gpart create -s mbr da4
da4 created
# gpart add -t fat32 da4
da4s1 added
```

Now that this disk has a partition table, I can create a FAT32 filesystem on it.

FreeBSD can assign a label to a FAT filesystem. Use `-L` and the label name. A FAT label can be up to 11 characters. Legitimate characters include alphanumerics, spaces, and a couple dozen basic symbol characters.

```
# newfs_msdos -L pertwee /dev/da0s1
/dev/da0s1: 1953048192 sectors in 30516378 FAT32 clusters (32768 bytes/cluster)
BytesPerSec=512 SecPerClust=64 ResSectors=32 FATs=2 Media=0xf0 SecPerTrack=63 Heads=255 HiddenSecs=0
HugeSectors=1953525105 FATsecs=238410 RootCluster=2 FSInfo=1 Backup=2
```

This FAT32 filesystem is ready for use. I now have a flash drive to hold *Doctor Who* seasons 7 through 11.

If you need to create an older version of FAT, use the `-F` argument and the FAT version number. FreeBSD supports three versions of FAT: FAT12, FAT16, and FAT32.

```
# newfs_msdos -F 12 /dev/da4s1
```

Yes, some networks (often telecom) have hardware so old it only speaks FAT12. And you thought your servers were old!

Fixing FAT Filesystems

FreeBSD is pretty good with FAT, but not everyone is so careful. If you have a device with a corrupt FAT, fsck_msdosfs(8) might be able to fix it.

# fsck_msdosfs /dev/da4s1 ** /dev/da4s1 ** Phase 1 - Read and Compare FATs ** Phase 2 - Check Cluster Chains ** Phase 3 - Checking Directories ** Phase 4 - Checking for Lost Files 1 files, 2074816 free (64838 clusters)

If fsck_msdosfs(8) finds an error, it offers you the option of "fixing" it. You should end up with a usable filesystem, but not necessarily all the files you'd hoped to recover. As with any filesystem, your ability to recover a FAT with complex corruption is limited only by your understanding of FAT.

MSDOS and /etc/fstab

List FAT filesystems in */etc/fstab* to easily mount them. Here I list my new FAT device with a mount point of */media*.

```
/dev/da4s1 /media msdosfs rw,noauto 0 0
```

Note that I used the `noauto` mount option, so that the system won't automatically mount this device at boot. I don't want my system to hang during boot because I unplugged the USB drive!

Other MSDOS Hints

FreeBSD's mount_msdosfs(8) includes several rarely used features that might help you when working with older FAT filesystems, such as managing Windows 95's long filenames. Always check the man page when you have a FAT issue.

While FAT filesystems don't support UTF-8 character sets or other locales, you can have FreeBSD map between the DOS codepage and these character sets. You must load the kernel module *msdosfs_iconv.ko*, and configure your locale. Use the `-L` flag to set your locale and `-D` to specify the DOS codepage, as discussed in mount_msdosfs(8).

If you need to handle a whole bunch of FAT disks, check out the mtools package. It contains several flexible tools for working with FAT filesystems without mounting them.

CDs: ISO 9660

The ISO 9660 was developed to support CDs. FreeBSD supports reading and writing CD images, as well as burning those images to disk. The overwhelming majority of CDs are formatted with ISO 9660, which FreeBSD calls *cd9660*. DVDs and Blu-ray disks use UDF, discussed in the next section.

Normal CDs are read-only media. You cannot rewrite most CDs. Rewritable CDs exist, but they have short lives and are mostly supplanted by more robust USB flash drives, so we're not going to discuss them.

The write-only nature of CDs can be an advantage in some environments, however. You can be confident that the files you burned to a CD, labeled in your clumsy I-use-a-keyboard-not-a-pen handwriting, won't be tampered with. A CD won't catch viruses from an infected client computer. While you can throw a switch to flip some USB flash drives to read-only mode, that still leaves room for human error to flip the switch back at an inconvenient moment.

We're not going to cover multisession disks. Multisession disks let you append data to the end of a read-write optical disk, rather than overwriting the existing filesystem. Using multisession CDs depends largely on your hardware.

Creating a CD has two parts: making an image of an ISO 9660 filesystem, and burning that image to disk. We'll cover each separately.

Multiple CD Drives

In systems with multiple CD drives, figuring out which piece of hardware corresponds with which device node can frustrate anyone. If you have multiple */dev/cd* devices, the easiest way to identify which device node goes to which drive is to use cdcontrol(1).

Close the doors of all your CD trays. For each device node, run the `cdcontrol eject` command. Give the device node with the `-f` flag, like so.

```
# cdcontrol -f cd1 eject
```

And voilà! The tray of the hardware associated with */dev/cd1* pops open.

The cdcontrol(1) command also has a `close` function, but not all CD drives support automatic closing.

My examples assume that you're using the CD drive */dev/cd0*.

Reading CDs

To mount a CD, tell mount(8) to mount filesystem type cd9660.

```
# mount -t cd9660 /dev/cd0 /media
```

ISO 9660 neither needs nor wants partitions, but on rare occasions you might find an optical disk burned with a partition table. If you have trouble mounting the disk, check for a partition table.

CD drives can also have a permanent entry in */etc/fstab*.

```
/dev/cd0 /cdrom cd9660 ro,noauto 0 0
```

I assign this CD drive a mount point of */cdrom*. If I want to use */media* for the CD drive, I'll need a different mount point for any USB drives. Removable drives in */etc/fstab* should always use the `noauto` option, so a missing CD won't make the system hang at boot. All CD mounts are read-only.

Burning CD Images to Disk

So you have an ISO image and want to get it on disk. Use cdrecord(1) from the cdrtools package to burn ISO images to disk. Give the image file as an argument.

```
# cdrecord FreeBSD-10.4-RELEASE-amd64-disc1.iso
```

If you have multiple CD drives, add the `-dev` flag and the device name.

```
# cdrecord -dev=cd1 FreeBSD-10.2-RELEASE-amd64-disc1.iso
```

It might take a while for cdrecord(1) to burn the disk, depending on the CD drive's speed.

Creating ISO 9660 Images

Burning existing images is fine, but FreeBSD also lets you create images for CD drives from your own files. Use the mkisofs(1) utility included with cdrtools to transform a directory tree into a filesystem image.

Here I've put the files and directories I want to burn in the directory *$HOME/cdfiles*. Take a look at what's in that directory.

```
# ls -la cdfiles/
```

```
total 167
drwxr-xr-x   2 root  mwl      5 Aug 25 14:30 .
drwxr-xr-x  21 mwl   mwl     84   14:24 ..
-rw-r--r--   1 mwl   mwl  50709 Mar 13  2014 exposition.zip
-rw-r--r--   1 root  mwl  50718 Aug 25 14:30 file1.zip
-rw-r--r--   1 root  mwl  50719 Aug 25 14:30 file3.zip
```

Now transform this directory into an ISO 9660 filesystem. Give the destination image file with `-o`.

```
# mkisofs -o cd1.iso cdfiles/
```

The image file is now ready to burn to CD. Before burning it, however, double-check the contents by mounting the disk image as discussed in Chapter 5. You'll see that the contents are slightly different.

```
# ls -la /media/

total 159
dr-xr-xr-x   1 root  wheel   2048 Aug 25 14:30 .
drwxr-xr-x  28 root  wheel     35 Aug 25 11:47 ..
-r-xr-xr-x   1 root  wheel  50709 Mar 13  2014 expositi.zip
-r-xr-xr-x   1 root  wheel  50718 Aug 25 14:30 file1.zip
-r-xr-xr-x   1 root  wheel  50719 Aug 25 14:30 file3.zip
```

The ownership has changed. If the directory had symbolic links, they'd be missing. And one file has a truncated name!

The ISO 9660 standard imposes a limit of eight characters on a filename, plus a period and a three-character extension. It has a maximum depth of eight directories, a maximum of 65,535 directories, and assorted other annoying limitations. Also, ISO 9660 does not reflect all of POSIX's filesystem semantics—or all of the Microsoft filesystem standards. Various extensions support these features, however.

ISO 9660 Extensions and Features

Using any ISO 9660 extension requires specifying the extension when creating the ISO image. You cannot retrofit extensions onto an existing ISO, as the information the extensions support doesn't exist in the ISO. The two most common sets of extensions are Joliet and Rock Ridge.

Microsoft created Joliet extensions to support Windows filenames and attributes. Filenames on an ISO with Joliet extensions can be up to 103 characters long, and directories can run more than eight layers deep. Activate Joliet extensions with the `-J` flag.

Rock Ridge extensions are designed for Unix-like systems. They also allow longer filenames and greater directory depth, but also encode Unix attributes like file ownership, symbolic links, and more. Enable Rock Ridge extensions with the `-R` flag.

The El Torito extensions mark an ISO as a bootable device. You still have to put a boot loader and an operating system in the ISO. Enable El Torito with `-b` and the path to the boot loader. See "Bootable CDs" later this chapter for more information.

The presence of any of these extensions will not interfere with reading the CD on other operating systems. Windows systems can open CDs created with Rock Ridge extensions, but ignore the Unix-style permissions and symbolic links. Unix hosts can open Joliet CDs, but won't use Windows-specific information. FreeBSD's CD driver handles all these extensions by default.

If a CD will be used only on FreeBSD hosts, you can completely bypass all ISO filename restrictions with `-U`. The image will probably violate ISO 9660, but retain all filename information.

I normally create my ISOs with both the Joliet and Rock Ridge extensions. Unix hosts will get the full filename and other POSIX metadata, while Microsoft systems will at least get the complete name.

```
# mkisofs -JR -o cd1.iso cdfiles/
```

Burn the image to disk and you're ready to go.

Bootable CDs

In addition to the directory tree, bootable CDs need a boot loader, an operating system kernel, and some kind of userland.

Specify the boot loader with `-b`. FreeBSD includes a boot loader for running FreeBSD from CD, */boot/cdboot*. Unlike many other boot loaders, *cdboot* does not emulate a floppy disk, so you'll have to also add the `-no-emul-boot` flag.

Copy the kernel and userland into your source directory tree before creating the ISO. A FreeBSD boot loader expects to find a kernel and any kernel modules in the directory */boot/kernel*. You can change this, if you hack the loader internals. The userland can be small, but to actually use a bootable CD you'll want basic programs such as those found in */bin* and */sbin*.

Here I have my complete directory tree in the directory *cdfiles*, including a copy of the boot loader. I create the bootable ISO with mkisofs(1).

```
# mkisofs –R –no-emul-boot –b cdfiles/boot/cdboot –o cd1.iso cdfiles
```

The easiest way to create a custom FreeBSD boot CD, including custom installation media, is the `make release` process.

Sadly, CDs are approaching obsolescence. DVDs and Blu-ray are the more modern optical media.

DVD and Blu-ray: UDF

You can use the CD-style ISO 9660 filesystems on any optical media up to 4.3 GB. This means you can use a CD filesystem on a single layer DVD. Larger optical media, like a multi-layer DVD or Blu-ray, uses the Universal Data Format (UDF) filesystem or a hybrid ISO 9660/UDF filesystem. Like ISO 9660, UDF is a read-only filesystem. You can create UDF images and burn them to disk, but you can't edit an existing image.

FreeBSD supports reading from and writing to UDF disks. (UDF is a read-only filesystem, but FreeBSD can write a filesystem to a disk, exactly like an ISO.) You can use DVD and Blu-ray disks to store and transfer data. Writable optical media is not a durable long-term storage medium, but in some environments it's the best option.

UDF support is *not* the same thing as reading commercial DVDs and Blu-ray disks. Commercial optical disks, like those used for films, are encrypted. There is no freely available solution for watching DVD or Blu-ray disks on any open source operating system. An Internet search will expose many methods for breaking much of that encryption, but that's beyond the scope of this book.

We're not going to spend time exploring the specifics of the DVD-Video format, a specific arrangement of files suitable for a Blu-ray or DVD player. That quickly becomes complicated, and you can find extensive cross-platform literature on DVD-Video. Use third-party tools like the dvdauthor package to create such filesystems.

Finally, as with CDs, we're not going to cover multisession disks. Multisession disks are where you append data to the end of a read-write optical disk, rather than overwriting the existing filesystem. Hardware support for multisession disks is mixed, and DVD-RW and DVD+RW have different restrictions on their use. If you really want to dive into multisession disks, check the FreeBSD Handbook for all the tedious details. Instead, we'll cover what all the UDF users must know.

UDF Hardware

DVD disks are standardized. And you get several standards to choose from.

The most common standards for DVD disks are DVD-R and DVD+R. Each also comes in a read-write format, DVD-RW and DVD+RW. DVD drives work with one of these, but might not support both. Most DVD drives manufactured today are classed as DVD+/-, and support both media types. Before experimenting with reading and writing DVDs, be sure that your hardware and blank media are compatible with each other. Burning a DVD-RW on a drive that supports only DVD+RW will not make you a happy sysadmin.

You’ll burn files and images to DVD-RW and DVD+RW differently. Be sure to follow the instructions for your blank media.

You’ll also find a *very* few DVD-RAM disks, which are read-write DVD disks. DVD-RAM drives and players were manufactured in small numbers, and are difficult to find nowadays. A DVD-RAM disk can be rewritten at slow speed 100,000 times. (Faster writing speeds reduce the number of rewrites.) A DVD-RAM is expected to be stable for much longer than either other type of disk. With its high rewrite tolerance, a DVD-RAM is best managed like a regular hard drive. Partition it with gpart(8), create a file system with newfs(8), and stash your data.[2]

Blu-ray disks are also standardized. Unlike DVD, there’s only one standard. Lucky you.

Mounting UDF Disks

Mount UDF disks with mount_udf(8). You’ll need two arguments: the device node and the mount point. Here I mount a Blu-ray at */dev/cd0* (containing all of the FreeBSD releases I use in my test lab) on */mnt*.

```
# mount_udf /dev/cd0 /mnt
```

There are no UDF-specific mount options. While you can theoretically use some generic options from mount(8), I don’t know of any case where they are useful. UDF is a read-only filesystem.

Creating UDF Images

Use mkisofs(8) to create a UDF disk image. You’ll need the `-udf` and the `-iso-level 3` flags, in addition to the `-R` and `-J` needed for an ISO 9660 image.

```
$ mkisofs –R –J –udf –iso-level 3 –o imagefilename files
```

Here I bundle up all of the FreeBSD releases in */home/mwl/udf* into the UDF image.

```
$ mkisofs -R -J -udf -iso-level 3 -o FreeBSD-releases.udf /home/mwl/udf/*
```

You’ll get a bunch of spammy status messages saying exactly how far along the filesystem creation process is, and then a nice summary.

If you have any doubt about the quality or contents of your UDF image, use mdconfig(8) as discussed in Chapter 5 to mount the disk image and make sure it’s correct. Once you’re happy, burn the image to disk.

Burning UDF Images

While mkisofs(8) can burn UDF filesystems to disc, it's generally recommended to use growisofs(1) from the dvd+rw-tools package. Despite the name, the software works on both DVD formats.

```
# pkg install dvd+rw-tools
```

One annoyance with `growisofs` is that you cannot run it with sudo(8). Filesystem tools like `growisofs` need to access arbitrary data on the disk, and running `growisofs` with `sudo` grants clever people unrestricted access to any file on the system. It's as if fsdb(8) checked to see if it was running under sudo before continuing. To burn a UDF image, you must actually be root. This means, you must run `su` and enter the root password—running `sudo su` will not work. The growisofs(8) command checks for the presence of the SUDO_COMMAND environment variable, so you can use a wrapper script to strip that variable, use `sudo su -l`, or pull out the shotgun and do `unsetenv SUDO_COMMAND`.

Once you have a clean root shell, use the `–dvd-compat` and `–Z` flags to burn your UDF image *imagefile* to the disk burner */dev/burner*.

```
# growisofs –dvd-compat –Z /dev/burner=imagefile
```

Yes, that's an equal sign between the disk device and the image file.

Here I burn the image *`FreeBSD-releases.udf`* to the Blu-ray writer at *`/dev/cd0`*.

```
# growisofs -dvd-compat -Z /dev/cd0=FreeBSD-releases.udf
Executing 'builtin_dd if=FreeBSD-releases.udf of=/dev/pass11 obs=32k seek=0'
/dev/pass11: pre-formatting blank BD-R for 24.8GB…
/dev/pass11: "Current Write Speed" is 6.1x4390KBps.
21659648/6213132288 ( 0.3%) @1.0x, remaining 28:35 RBU 99.9% UBU 3.0%
40435712/6213132288 ( 0.7%) @1.3x, remaining 22:53 RBU 99.9% UBU 100.0%
…
```

Go refill your caffeine. When you return, you should have a disk with a hybrid ISO 9660/UDF filesystem.

Dumping Files to UDF

I have a bias towards image files, but for one-time burning jobs you might not need them. You can dump a directory hierarchy straight through growisofs(8) onto disk, without creating an intermediate UDF image file. Instead of an equal sign and the image file, use a space and the path to the files.

```
# growisofs –dvd-compat –udf –iso-level 3 –Z /dev/burner /files/to/burn/
```

If I have a bunch of files in *`/home/mwl/udf`* that I want to burn to the Blu-ray burner at

/dev/cd0, I can do it like this.

```
# growisofs -dvd-compat -Z /dev/cd0 /home/mwl/udf/
```

The burn messages are very similar to those that show up in burning an image file, and the process takes about as long. It’s a good time to clean out that really scary bottom drawer where you spilled the cola a few years back. Eventually, you’ll have an optical disk containing your files.

Linux Filesystems

FreeBSD supports the Linux filesystems ext2fs, ext3fs, ext4fs, and ReiserFS. These filesystems support many of the same features as UFS, and readily map to FreeBSD features. Some filesystems are better supported than others, however.

In my experience, the best use for FreeBSD's Linux filesystem support is to migrate a Linux system to a FreeBSD one. While you can copy files over the network, on a large system it's far faster to remove the hard drive from one machine, plug it into the new host, and copy from the old filesystem to the new one. FreeBSD can run most Linux systems in a jail.

Linux filesystem support is also useful on a dual-boot desktop or laptop, so you can access any Linux files while running FreeBSD. (In all fairness, most Linux systems can also mount FreeBSD UFS partitions, and some of them can mount ZFS.)

We'll discuss ext and ReiserFS separately.

ext Filesystems

FreeBSD classifies all ext filesystems as type *ext2fs*. The ext3fs and ext4fs filesystems are expansions and variations on ext2fs. You must load the kernel module */boot/kernel/ext2fs.ko* before using any ext filesystem. There is no ext-specific mount command, so you must use mount(8)'s `-t` option to mount it.

```
# mount -t ext2fs /dev/da4p2 /mnt
```

FreeBSD has full read-write support for ext2fs.

While FreeBSD can both read and write ext3fs, some features are missing. FreeBSD can't journal ext3fs. Inodes greater than 128 bytes are not supported. Linux extended attributes also don't work, although many of those attributes don't directly map to anything on FreeBSD, so that's not terribly surprising.

FreeBSD mounts all ext4fs filesystems read-only. FreeBSD cannot write to ext4fs filesystems.

If you need to do a lot of work with ext filesystems, grab the e2fsprogs package. The tools let you create and fsck(8) ext filesystems.

ReiserFS

Many ReiserFS features don't map easily onto anything FreeBSD supports. While FreeBSD can mount ReiserFS volumes read-only, that support is best reserved for migrating from Linux, digital forensics, or other non-production uses.

FreeBSD does not have a ReiserFS-specific mount(8) command. You must use the `-t` option to mount a ReiserFS volume.

```
# mount –t reiserfs /dev/da4s1 /media
```

You cannot create, repair, or edit ReiserFS volumes on FreeBSD. And there's really no reason to add them to */etc/fstab*.

Now that you can manage local storage, let's go on to FreeBSD's only mandatory logical filesystem: devfs.

---

[1] Unless you have more than one floppy drive, in which case figuring out the device node is the least of your problems.

[2] You could theoretically use DVD-RAM as a production ZFS storage provider. If you actually do this, post about it and send me the link. I appreciate knowing when I'm a lunatic influence on people.

Chapter 2: devfs

Unix-like operating systems traditionally put the filesystem’s interface to the hardware in the */dev* directory. The special files in */dev* are called *device nodes*. Running commands on device nodes gives instructions to the hardware. You’ve seen device nodes like */dev/da0* or */dev/ada0p3*, where the device name indicates a disk or a partition thereof. You’ll also see device nodes for hardware like keyboards (*/dev/ukbd0* or */dev/kbd0*), serial ports (*/dev/cuaa0*), and more. You’ll also find device nodes for system functions, like the vaguely randomish number generator */dev/random*, terminal sessions (*/dev/ttyv0*), and so on.

FreeBSD creates and configures these device nodes automatically, using the device file system devfs(5) and the supporting process devd(8). With no special configuration, they automatically configure all typical system common features and functions. If you do anything different, though, you might need to reconfigure devfs(5). A chrooted process that can’t access */dev* might need a device node for logging, but it certainly doesn’t need access to disks or terminals or anything like that. A jail needs access to the device nodes for logging in and basic programs, but shouldn’t see any of the host’s underlying hardware. These cases, and more, require that the sysadmin reconfigure devfs to provide the needed infrastructure.

FreeBSD breaks device node management into three pieces: configuring devices present at boot, configuring devices that appear after boot, and managing device node permissions and access control.

Boot-time /dev Configuration

Sysadmins can change device nodes using normal system tools, changing their ownership and permissions as desired. But suppose an unprivileged process needs access to the serial terminal */dev/cuaa1*. I would normally add that user to the **dialer** group, but that grants access to all the serial ports. If you need tighter control, you can change the owner of */dev/cuaa1* to the unprivileged user. Or perhaps a piece of clunky software might need a device node accessible under a different name.

The problem is, devfs(5) is a logical filesystem. It's in system memory. When you reboot the server, all your manual changes to */dev* evaporate. The kernel and devfs create a pristine */dev* that reflects the initial hardware configuration.

While the kernel and devfs know how to do their job without any help from you, they read */etc/devfs.conf* at boot to see if you have any special instructions for them. Rules in *devfs.conf* have the format of:

```
action devicename desiredvalue
```

The *action* describes the desired change. The *devicename* entry is the device node you want to change, while the *desiredvalue* is a new value. Using these rules you can add new device nodes, change their ownership, and alter their permissions.

Devfs only reads *devfs.conf* at boot. To make a change take effect immediately, change the existing */dev* entries. Any time you change *devfs.conf*, reboot to verify the configuration.

New Device Nodes

The `link` action creates a symlink between a new device node and an existing node. This lets you create aliases for existing devices.

```
link cd0 cdrom
link cd0 acd0
```

Devfs creates two new device nodes, */dev/cdrom* and */dev/acd0*, that point to the CD drive at */dev/cd0*. Some software expects the CD drive to be at */dev/cdrom*, and I don't feel like tweaking those preferences. And for many years, FreeBSD's ATA CD drives lived at */dev/acd0*, and sometimes my fingers revert to that while my brain is otherwise occupied.[1]

Device Owner

Change a device node's owner and group with the `own` action. Here, we let user **mwl** have absolute control of the CD drive.

```
own cd0 mwl:mwl
```

Many programs that work on device nodes have internal checks to verify that they're running as root before doing anything. I've seen more than one program shut itself down when it's not run as root, without even trying to access the target device nodes. Changing the device node ownership won't make those programs work.

While you can change the owner of a device node alias, that change applies only to the symlink, not the target node. Always change the owner of the actual device node.

Device Permissions

The *mode* keyword changes the permissions on a device node. Give the desired permissions in octal form.

```
mode cd0 0664
```

This grants read and write access to the owner and group owner of */dev/cd0*.

As with ownership, only change the permissions of the actual device node, not any aliases.

Dynamic Hardware

Devfs dynamically creates new device nodes when new hardware gets plugged in, and deletes device nodes when the hardware is removed. Much hot-pluggable hardware is very simple—when a new USB keyboard appears, FreeBSD attaches it to the console.

Other hardware requires more complicated configuration. The device daemon devd(8) automatically runs userland programs when hardware appears, disappears, and changes state. You can have, say, FreeBSD automatically run dhclient(8) when you plug in your laptop's Ethernet interface, or change the permissions on a USB-to-serial-port adapter.

The devd(8) daemon reads its configuration from `/etc/devd.conf` and any file ending in `.conf` in the directory `/usr/local/etc/devd`. As `/etc/devd.conf` is a system file and upgrades overwrite it, I strongly recommend placing your local rules in `/usr/local/etc/devd`. You could use a single file for your rules, but if your rules get complicated using a separate local rule file for each type of device helps.

Devd Rules

Devd watches for four separate types of events: *attach*, *detach*, *nomatch*, and *notify*. Each of these events triggers a rule of that type. It also has an *options* statement, which controls how devd(8) itself behaves.

Attach rules trigger when hardware matching the rule is attached to the system. When you plug in a USB flash drive, an attach rule can mount the drive and scan for viruses.[2]

Removing hardware from a system triggers *detach* rules. The kernel automatically notices when you remove hardware and removes the resources from the system, so detach rules are uncommon. There's no need to remove the network configuration from a nonexistent interface.

A *nomatch* rule triggers when you plug in new hardware that doesn't match any device driver. Generally speaking, hardware without a device driver is unusable.

The *notify* rules apply when the kernel sends a matching event notice to userland. These messages normally appear in `/var/log/messages` and on the console. The message that a network interface has come up is a notify event.

Each rule also has a priority, with 0 being the lowest; devd(8) processes only the highest matching rule, skipping any lower-priority matching rules.

Finally, a rule has a bunch of matching terms, and an action to take. If the matching terms match the device, the rule type matches the event, and no higher-priority rule has kicked in, devd(8) triggers the specified action.

Here's a sample rule.

```
notify 0 {
  match "system" "IFNET";
  match "subsystem" "!usbus[0-9]+";
  match "type" "ATTACH";
  action "/etc/pccard_ether $subsystem start";
};
```

This is a notify rule, so it activates when the kernel notifies the userland of an event. But as a priority 0 rule, it triggers only if no rule of higher priority matches the specified criteria.

The matching terms appear within brackets. For this rule to trigger, it must match the system IFNET. The device subsystem must match the regular expression `!usbus[0-9]+`. This boils down to "the device driver can't start with usbus," which excludes USB devices. This rule matches on an ATTACH, or when the kernel notifies userland that hardware has been attached to the system. In English, this rule fires when the kernel announces that a non-USB network device is plugged in. It's intended for the removable network cards found on older laptops.

The last line of this rule (other than the brackets) gives an action to take. This rule fires up the network configuration script for removable network cards.

Taken all together, this rule says "When you plug in a removable non-USB network card, configure it."

Match Statements

You can match almost any aspect of a device in a devd(8) rule. I'm not going to list all the possible matches here, as that would be pages and pages of stuff you'll almost never use. Read devd.conf(5) for the complete list. In general, though, you can match devices by manufacturer, vendor, model, serial number, network interface media, parent device, hardware revision, and so on.

Some rule types permit additional terms. In attach and detach rules, for example, you can use *device-name* as an alias for *match device-name*. These are optional.

I could fill this entire book with more examples. The best way to understand *`devd.conf`* rules is to read the examples in *`/etc/devd.conf`*.

Creating devd(8) Rules

I strongly recommend never writing your own devd(8) rules from scratch. Look at */etc/devd.conf*. Find a rule that does something like what you want. Copy that rule into a *.conf* file in */usr/local/etc/devd*. Edit that rule until it does what you want.

In my opinion, the easiest rules to write are `notify` rules. You can read */var/log/messages* or check the console to see exactly what messages the kernel passes to userland, and use them as match terms.

But easy is boring, so we'll create a simple `attach` rule.

Flash Drives

If I plug in a flash drive, I probably want it mounted on */media*. This isn't always true, but it's common enough that I'm willing to have it be the default. I'd much rather manually unmount */media* once every ten inserts than manually mount it nine times out of ten.

On a simple system with SATA drives the flash drive probably appears as */dev/da0*. On a more complex system it might be any device node. Insert the flash drive to see where it shows up on your system.

```
attach 10 {
  match “device-name” “umass1”;
  action “sleep 2 && mount -t msdosfs /dev/da4s1 /media”;
};
```

Some USB drives need a second or two to wake up when you plug them in, so I added the sleep(1) command. You could also call an external script.

FreeBSD can't prevent you from removing a mounted hard drive, but it can complain. Here we automatically force unmounting */media* when devd(8) detects the device removal.

```
detach 10 {
  match “device-name” “umass0”;
  action “umount -f /media”;
};
```

You don't have to mount the drive, however. I once had to prepare 40 flash drives for a large FreeNAS deployment. A *devd.conf* rule and a simple script let me tell a minion “when the light stops flashing, put the next one in” so I could return to playing FreeCell. If your organization's operations or security policies say “don't plug USB media into running production servers,” a *devd.conf* rule that calls newfs(8) can enforce that. (By the time FreeBSD could run `newfs`, USB's inherent security flaws could have already destroyed your server. Disconnecting USB ports is more secure, and leaves more flexibility for future maintenance than filling the ports with superglue.)

CARP Failover

The Common Address Redundancy Protocol (CARP) lets multiple hosts share one IP address. When one host dies, another host takes over the address, keeping the service alive. Some services need a quick kick when failing over. Often you need to tell the backup database that where it was once the student, it is now the master.

CARP events generate kernel notifications, which means you can use `devd` to tell the application what's happening. Here's a *devd.conf* configuration to run a script when the host takes over as the CARP master.

```
notify 30 {
  match "system"          "CARP";
  match "subsystem"       "[0-9]+@[0-9a-z]+";
  match "type"            "MASTER";
  action "/usr/local/scripts/carp-up";
};
```

This is a notification rule, which means it triggers when the kernel sends a notification message to userland. We have more complicated matching terms, because we only want to run the script if this specific interface goes up. If an interface becomes the CARP master, `devd` runs the script */usr/local/scripts/carp-up*.

When the interface goes down, `devd` runs a second command to inform the host it's now the backup.

```
notify 30 {
  match "system"          "CARP";
  match "subsystem"       "[0-9]+@[0-9a-z]+";
  match "type"            "BACKUP";
  action "/usr/local/scripts/carp-down";
};
```

This rule is almost identical to the first, except for the type of event and the script we run.

Debugging devd.conf

Debugging devd(8) rules can be difficult. I strongly recommend using logger(1) in action rules, so that the commands you run get sent to the system log. You might need to enable the system debugging log */var/log/all.log* in */etc/syslog.conf* to catch everything that's happening, or look at the system console.

Any time you need to perform an action when hardware state changes, consider *devd.conf*. Many functions that could go into a script are also accessible through devfs rules, though.

devfs(5) Rules

In addition to changing devfs with *devfs.conf*, and running commands on dynamic devices with devd(8), you can use devfs rules with */etc/devfs.rules*. All device nodes, whether present at boot or dynamically added, are subject to *devfs.rules*. Rules let you set ownership and permissions on device nodes, and make device nodes visible or invisible. In addition to the configuration file, you can add devfs rules at the command line.

Just like *rc.conf* and *periodic.conf*, FreeBSD has a default *devfs.rules* file in */etc/defaults/*. Upgrades overwrite this file. Put your own devfs rules in */etc/devfs.rules*. Entries in */etc/devfs.rules* override rules in the defaults file by rule number—that is, rule 5 in */etc/devfs.rules* overrides rule 5 in */etc/defaults/devfs.rules*.

devfs Rule Design

Every devfs rule starts with a unique name and rule number, given between square brackets. You can refer to this ruleset by name elsewhere in *devfs.rules*. The number is only a unique identifier—devfs rules don't have a priority or processing order outside the rule. The rule then has a statement adding an action this ruleset takes.

Here's a complete devfs rule from the default *devfs.rules*.

```
[devfsrules_hide_all=1]
add hide
```

This rule is named devfsrules_hide_all, and is rule number 1. It adds one action to this ruleset: *hide*. It hides every single device node, giving you an empty */dev*.

Rule Content

All devfs rules begin with the word *add*, adding a rule to the ruleset. Then there's 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, whatever the rule does. Here's a complete devfs rule.

```
add path cuau* user mwl
```

This rule declares that the user **mwl** owns all device nodes with a name beginning with cuau. These device nodes go to serial ports. On a multi-user system this would be a bad idea. On my laptop, where I'm the only one who should ever use the serial ports, it's not terrible.

Devices specified by path use shell regular expressions. To match a variety of devices, use an asterisk as a wildcard, as in the cuau example above. You can use the wildcard in the middle of the string as well, or give an exact device name. If I wanted to change only the second serial port on my laptop, I could specify device cuau1.[3]

The *type* keyword specifies the type of devices this rule applies to. Valid types are *disk* (disk devices), *mem* (memory devices), *tape* (tape drives), and *tty* (terminals and pseudoterminals). I rarely use the *type* keyword precisely because it's so broad. While I might want a particular disk to have a different configuration, I don't want *all* my system's disks reconfigured.

If you include neither the *path* nor *type* keywords, devfs applies the rule to all device nodes. If you don't know why this is a bad idea, try it once on a test machine. It's a vital part of your education.

The rule's action can be any one of *group*, *user*, *mode*, *hide*, and *unhide*.

The *group* action lets you set the device node's group owner as an additional argument. Similarly, the *user* action assigns the device node's owner. I changed the owner in our first example, but here I change the group owner of the same device.

```
add path cuau* group wheel
```

The *mode* action lets you assign octal permissions to the device node.

```
add path da4 mode 664
```

The *hide* keyword lets you make device nodes disappear, and *unhide* shows them again. Programs cannot use hidden device nodes. You can use *hide* and *unhide* for chroots and jails, where the system should have access to only a small subset of the device nodes.

Nesting Rules

One ruleset can include other rulesets. Consider this default ruleset.

```
[devfsrules_jail=4]
add include $devfsrules_hide_all
add include $devfsrules_unhide_basic
add include $devfsrules_unhide_login
add path zfs unhide
```

This ruleset is designed for jails. It's assigned the name devfs.rules_jail, and is assigned ruleset number 4. The "add include" statements pull in other rulesets, named *devfs.rules_hide_all*, *devfs.rules_unhide_basic*, and *devfs.rules_unhide_login*. These rules are defined earlier in the default rules. It also explicitly unhides the device nodes under `/dev/zfs`.

You might need a slightly different ruleset for a particular application. Some apps that run in a chroot(8) need a syslog device socket, or */dev/log*. You could create your own ruleset that refers to the hide_all rule, and then unhide the log device.

```
[devfsrules_logonly=100]
add include $devfsrules_hide_all
add path log unhide
```

Similarly, you might have a jail that needs access to a specific device. You can refer to the jail devfs rule and add the device you need. Here I have a jail used for serial port connections.

```
[devfsrules_serialjail=100]
add include $devfsrules_jail
add path cuau* unhide
```

Nested rules let you create any variation you want, while letting changes percolate through the ruleset.

Mounting /dev with Rules

You probably won't want to change the devfs rules for */dev*. The main system should have access to all the devices on the system. Applications like jails mount their own */dev* filesystems appropriately. If you need to mount a device filesystem for an application or a chroot, however, make an entry in */etc/fstab*.

```
devfs /var/app/dev devfs rw,ruleset=101
```

To use a ruleset, use the *ruleset* mount option and the ruleset number. Be sure there's no space between the comma and the ruleset.

Devfs at the Command Line

You can use the devfs(8) command to add, remove, and change devfs rules at the command line. For example, running `devfs rule apply hide` removes all your device nodes. Go ahead, try it. Very few people have a use case for this feature, so I'm not going to cover it, but if you are one of those folks read devfs(8).

If you decide to play with this, be sure to use the `-m` option to specify a mount point for a non-default device filesystem—otherwise, you'll be working on the default system */dev*, and you might need to reboot your system to recover.

Now let's discuss some FreeBSD supporting filesystems.

---

[1] Don't ask what I'm preoccupied with. The answer would not make you a happier person.

[2] Why scan for viruses on a FreeBSD machine? I'm not worried about the FreeBSD machine. But USB flash drives carry files between machines. I'd prefer to identify any viruses before infecting my Blu-ray player or the garbage

disposal.

[3] Yes, my laptop has multiple serial ports. It also has multiple hard drives, 64 GB RAM, and four processors. I have a very large lap.

Chapter 3: Namespace Filesystems

FreeBSD includes several smaller filesystems that present internal system information in a filesystem manner. These filesystems are almost always intended to support or debug specific software. The process filesystem procfs(5) displays process information, letting you examine files rather than using tools like ps(1). Some Linux software also expects to find hardware information available as the system filesystem. FreeBSD also supports the POSIX message queue filesystem with mqueuefs(5) and the file descriptor filesystem fdescfs(5).

FreeBSD doesn't enable any of these filesystems by default. Load or mount them only if an application requires them.

Running `mount` normally requires some kind of storage device behind it. Namespace filesystems don't have any kind of backing store, however. The `mount` command expects an argument in the space where the backing store should be. For namespace filesystems it's customary to use the type of filesystem as that argument, but in reality `mount` ignores that argument.

Process Filesystems

The process filesystem procfs(5) displays the system's current processes as files. Using procfs(5) lets you gather process information with programs like cat(1) and grep(1), rather than using process-specific tools like ps(1). Process information is security sensitive, and procfs across operating systems has a long history of security problems, so FreeBSD does not use procfs by default. All information available in procfs is also available through the preferred sysctl interface.

Some add-on packages, notably software ported from Linux, are designed to use procfs, however. And procfs is useful on FreeBSD jail hosts—not the individual jails, but the OS instance that supports those jails. Sysadmins managing these systems need procfs.

Mounting procfs(5)

The process filesystem is normally mounted at */proc*. The mount point exists in a default install, even though procfs isn't mounted. You can temporarily mount */proc* from the command line.

```
# mount -t procfs proc /proc
```

To make FreeBSD mount */proc* at boot, make an */etc/fstab* entry.

```
proc /proc procfs rw
```

Now that you have */proc*, let's poke at it a little.

What's in /proc?

Looking at */proc* shows a bunch of numbered directories. Each represents a single process ID—that is, the information for process 844 is in */proc/844*. Files in that directory are owned by the process owner. Each process has different files that represent the process' state. Read procfs(5) for all the painful details.

The odd directory is */proc/curproc*. This gives information on the process that's accessing */proc/curproc*. Programs use */proc/curproc* to check their own condition. If you examine */proc* interactively, using tools like ls(1) and more(1), */proc/curproc* describes that tool's process. If you want to examine your shell process try something like `ls /proc/$$`, which the shell expands for you before firing up the child process.

The process filesystem exposes operating system internals. You should never assume that a procfs for one operating system will match that used by another. Notably, FreeBSD's procfs is incompatible with Linux's.

/proc and /sys for Linux

FreeBSD’s Linux mode lets you run native Linux software on FreeBSD. Rather than emulating Linux, FreeBSD supports multiple kernel interfaces. FreeBSD directs Linux software to the Linux interface. Some Linux programs run faster on FreeBSD than on their native Linux.

Linux programs expect to have a supporting Linux userland. Running Linux on FreeBSD requires having a subset of a Linux userland in */compat/linux*. Similarly, some Linux software might requires a Linux */proc*, or even a Linux system filesystem.

Linux /proc

Linux makes extensive use of */proc*. Linux procfs is not completely compatible with FreeBSD’s, however. Some files in the Linux procfs have the same names as those in FreeBSD, but include different information, follow a different format, or cover concepts not used in FreeBSD. A FreeBSD process filesystem doesn’t have */proc/cpuinfo*, for example, as FreeBSD has no concept of bogomips. There’s no */proc/version* on FreeBSD, and if there was, it wouldn’t contain the same information as */proc/version* on Linux.

FreeBSD includes a Linux-style procfs for use by software expecting a Linux process filesystem or software run in Linux mode, linprocfs(5). Linprocfs requires a kernel module, *linprocfs.ko*, which loads automatically when you mount the filesystem.

Do not mount a Linux procfs on */proc*. FreeBSD software that can use procfs would think that it’s a FreeBSD procfs. Exposure to Linux procfs would distress such software, and your world has enough unhappy software. Mount Linux procfs at */compat/linux/proc*.

```
# mount –t linprocfs linproc /compat/linux/proc
```

To automate mounting Linux procfs at boot, make an */etc/fstab* entry.

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

FreeBSD’s Linux emulation layer transforms the filesystem for Linux software, fooling it so that it finds a Linux procfs in */proc*.

Linux /sys

Linux doesn’t only expose process information via a filesystem. It also exposes hardware information. Linux’s */sys* filesystem declares what hardware is available and how it’s configured. Like */proc*, certain Linux software expects to grub around in */sys* to find information and address hardware. You can use many pieces of Linux-only hardware configuration software on FreeBSD, if you mount a Linux */sys* using linsysfs(5).

On FreeBSD, */sys* is a link to the kernel source code. Mount the Linux */sys* under */compat/linux/sys*.

```
# mount -t linsysfs linsys /compat/linux/sys
```

Mount the Linux */sys* automatically at boot with an */etc/fstab* entry.

```
linsys /compat/linux/sys linsysfs rw
```

When you’re done configuring your hardware with that Linux-specific software, be sure to yell at your hardware vendor for not having a FreeBSD version of their utilities.

Both linprocfs(5) and linsysfs(5), as well as the FreeBSD-native procfs(5), are built on top of a filesystem layer called pseudofs(9). Pseudofs is a kernel programming framework for presenting filesystem information that changes every time you access it. Sysadmin never directly interact with pseudofs, but you’ll occasionally see references to it.[1]

Messaging Filesystems

Certain messaging and process management facilities are also implemented as filesystems or filesystem-like features. The most popular are the file descriptor filesystem and the POSIX message queue. You only need these filesystems if an application demands them.

File Descriptor Filesystem

The file descriptor filesystem fdescfs(5) gives processes filesystem-style access to their own file descriptors. File descriptors include standard input (descriptor 0), standard output (1), and standard error (2) plus any descriptors a program creates.

FreeBSD shows the three standard file descriptors under */dev/fd*. Processes can only see their own file descriptors in that directory. Programs can create dozens or hundreds of file descriptors, however, and some programs (notably Java) won't work without filesystem-level access to their descriptors. Viewing those additional file descriptors requires fdescfs(5).

Always mount fdescfs(5) at */dev/fd*.

```
# mount –t fdescfs fdesc /dev/fd
```

To automatically mount fdescfs(5) at boot, add an */etc/fstab* entry.

```
null /dev/fd fdescfs rw
```

You can now use Java.[2]

POSIX Message Queues

The POSIX inter-process communication (IPC) and semaphore standards include named message queues. FreeBSD gives you the option of exposing those queues as a filesystem.

Using message queues requires loading both the message queue filesystem mqueuefs(5) and the semaphore kernel modules. You don't need to actually mount an mqueuefs partition; the presence of the kernel module will let you use these message queues. Load the modules at boot with these *loader.conf* entries.

```
sem_load=YES
mqueuefs_load=YES
```

Mounting an mqueue filesystem lets you view the existing queues as files. The file's contents describe the queue's attributes. This might be useful for debugging. Here I mount mqueuefs at */mnt*.

```
# mount -t mqueuefs mqueue /mnt
```

Examine your queues, figure out your problem, and unmount the filesystem.

To automatically mount mqueuefs(5) at boot, use an */etc/fstab* entry.

```
null /mqueue mqueuefs rw
```

You can create POSIX queues by manually creating files, although this is strongly discouraged. POSIX message queues are designed for API use, not at the command line.

Now let’s take a look at how FreeBSD lets you rearrange filesystems on the disk.

[1] Do you really need to know this? Not really. But one day you’d stumble across pseudofs(9) and say, “Why didn’t that lazy Lucas cover that in his book?”

[2] I’m sorry.

Chapter 4: Rearranging Filesystems

Sometimes you’ll find that your filesystem isn’t laid out as you’d like, or you have needless duplication. While you can’t pick up and relocate UFS filesystems, and shuffling ZFS filesystems can incur penalties, FreeBSD includes two tools for arbitrarily rearranging and recycling filesystems: *null mounts* and *union mounts*.

Null mounts and union mounts are counterintuitive to many, and best understood by example.

Null Mounts

A *null mount*, sometimes called a loopback mount, adds a second way to access part of an existing filesystem. You might take */usr/home* and also make it available at */home*, or make */media* accessible as */usr/src*. Most mount commands work on partitions or disk images. Null mounts work on any directory. Create null mounts with mount_nullfs(8).

While you can do similar tricks with symlinks, commands that use the working path or use the system calls getwd(3) or getcwd(3) expose the non-symlinked path. With a null mount, this software sees the null mount path. Null mounts work with both UFS and ZFS. Users don't care so much about the true path, but applications like chroot(8) and jails most certainly do. A null mount can let you share a read-only ports tree among many jails on a system.

Understanding how null mounts change system behavior is easiest through example. My test machine puts user home directories at */home*. FreeBSD's installer puts home directories under */usr/home*. Here I use a null mount to make user home directories available in both locations. Let's start by using a symlink.

```
# cd /usr
# ln -s /home .
```

I can now access my home directory in both locations.

```
# cd /usr/home/mwl
# pwd
/home/mwl
```

The pwd(1) command understands that I'm in a directory tree that's rooted in */home*, even if I got there by going to */usr/home/mwl*.

Remove the symlink, and replace it with a null mount. The mount_nullfs(8) command takes two arguments: the existing directory tree, then the point where you want it to be mounted.

```
# mount_nullfs /home /usr/home
```

What impact does this have on users? I can now go to either */usr/home/mwl* or */home/mwl* and reach the same directory, just like with a symlink. Now that I'm using a null mount, the pwd(1) command shows my actual location.

```
# cd /usr/home/mwl
# pwd
/usr/home/mwl
```

While the real filesystem is rooted in */home*, pwd(1) saw only the null mount.

Many of us have closed-source programs or unfamiliar scripts with hard-coded paths. A null mount can keep that software working even as you move filesystems around your server.

You can use most standard mount(8) options with mount_nullfs(8). Use `-o` on the command line and list the options, separated by commas.

```
# mount_nullfs -o ro /home /usr/home
```

The `-o ro` option makes the null mount read-only. You can now edit your files under */home*, but the copy under */usr/home* is read-only. This particular example will really tick people off, so you probably shouldn't do that. But it might make sense to null mount a filesystem `noexec` or `nosuid`, depending on your use case. While `noatime` and `async` might sound sensible for your use case, they have no effect in null mounts.

You can perform null mounts automatically at boot with an */etc/fstab* entry.

```
/home /usr/home nullfs rw,noatime
```

If you decide to get into filesystem programming, null mounts are a great place to start. They contain the bare bones of a filesystem implementation. See mount_nullfs(8) for details.

Union Mounts

All FreeBSD file systems are *stackable*, which means you can mount one above another. Users can only see the filesystem on top. Again, this is better demonstrated than explained.

Suppose the directory */usr/src* contains the FreeBSD source code. Here I mount an unused, brand-new filesystem at */usr/src*.

```
# mount /dev/da0p1 /usr/src/
```

The */usr/src* directory now appears empty. I've stacked the new filesystem above the old. The original */usr/src* is still on the disk, but nobody can see it. Unmounting the empty partition reveals the code.

A union mount lets users see the contents of both filesystems simultaneously, in the same location. If I union mount a filesystem above */usr/src* and run ls(1), I'll see the contents of both filesystems.

Perform union mounts with mount_unionfs(8). The first argument is the filesystem that gets mounted on top; the second, the mount point and lower filesystem. Here's a rather messy but illustrative demonstration.

```
# mount_unionfs /home/mwl/ /usr/src/
```

I've mounted my home directory above the system source code. The */usr/src* directory now contains everything that was in */usr/src* before, plus everything that was in my home directory. So there's */usr/src/UPDATING* and */usr/src/.cshrc*.

I will never find anything ever again.

Union mounts work with both ZFS and UFS, but not both simultaneously. You can union mount a UFS partition over another UFS partition, or a ZFS dataset over another ZFS dataset. Union mounting UFS over ZFS or vice-versa appears to work, until you try to view or change the union-mounted data.

Union mounts have a bad reputation. The union mount feature was completely rewritten for FreeBSD 7.0, however, and it's more reliable than it once was. The mount_unionfs(8) manual page contains big scary warnings. The common cases, such as typical jail deployments, work fairly well and are widely deployed. Union mounts still do many things filesystem developers consider worrisome to awful, however. If you plan to use union mounts in a novel manner, test your system very carefully before deploying.

Upper and Lower Layers

When you query the filesystem, FreeBSD checks the upper layer first. If that fails, the query falls through to the lower level. Whenever you wonder why a union mount behaves the way it does, fall back to this fact.

Created files go in the top layer. With a union mount of */home/mwl* over */usr/src*, if I create the file */usr/src/test*, that file appears in the filesystem on top—*/home/mwl*. If I unmount the union, the file */usr/src/test* disappears with my *.cshrc* and all my other personal files.

Changed files also go in the top layer. If I edit */usr/src/Makefile* in this sample union mount, FreeBSD copies */usr/src/Makefile* to the top layer, creating */home/mwl/Makefile*. The edits go in the upper layer. When I disconnect the union mount, the original */usr/src/Makefile* is unchanged and the edited version appears in */home/mwl*.

When a file in each filesystem has the same name, only the top version is visible in the union mount. If I had */usr/src/UPDATING* and */home/mwl/UPDATING*, in this union mount you'd see only the version in my home directory.

The most common use for union mounts is jails. I'll use a jail example through the rest of this section. I've placed a basic FreeBSD install in */jails/basejail*. I want to use it as the bottom layer for */jails/jail1*, */jails/jail2*, and so on. These individual jail directories start off empty.

The mount_unionfs(8) command uses the first argument as the upper layer and the second argument as its bottom layer and the mount point. We need exactly the opposite behavior for reusing a read-only directory tree, however. The layer with the FreeBSD install needs to go on the bottom, so any changes can go into the top layer. We must use the top layer's mount point, however. Use the `-o below` mount option to achieve this.

```
# mount_unionfs -o below /jails/basejail /jails/jail1
```

The rest of this section discusses the behavior of union mounts with this configuration.

atime and Union Mounts

You'll quickly notice that the upper layer acquires a directory hierarchy. Reading a directory updates the directory's atime (access time). When you go into */jails/jail1* and run `ls /usr/bin`, a */usr/bin* directory appears in the upper layer. The directories in the upper layer have the correct atime for the union mount, while the lower layer keeps its own correct atime. These are called *shadow directories*.

Shadow directories can confuse you. Suppose you go to the lower layer and move */var/log/httpd/* to */var/log/httpd-old*. The union mount will still have a */var/log/httpd/* directory, because the shadow directory still exists. You'd need to delete that directory separately.

Most jails and virtual hosts don't need atime on their filesystem. You can disable atime in a union mount without disabling it in the lower filesystem by using the `noatime` mount option.

```
# mount_unionfs -o below,noatime /jails/basejail /jails/jail1
```

I have one virtual machine that does require atime, but that's because I read mail locally on the machine. Web, database, and application servers generally don't require it.

Deleting and Renaming Files

Removing files from a union mount behaves differently depending on the underlying filesystem.

A union mount on top of ZFS will not let you delete or rename files from the lower layer, as ZFS lacks whiteout support. The rm(1) command appears to work, but actually fails silently. If you try to move a file with mv(1), you'll create a duplicate of the file.

For most applications, an inability to change the lower layer is a benefit rather than a problem. If you're using one FreeBSD installation beneath dozens of jails, you don't want any of those jails to change the underlying installation.

UFS supports *whiteouts*, which allow a union mount's upper layer to hide a file visible in the lower layer. Removing a file from the lower layer of a UFS-backed union mount actually creates a new file in the upper layer of the same name but with inode number 1. Inode 1, by definition, is not part of any file. If a file contains inode 1, it doesn't exist. The lower layer's copy of the file is hidden, not removed.

Whiteouts mean that files can be undeleted. The `–W` flag to rm(1) attempts to undelete whited-out files. Give the filename as an argument.

```
# rm –W /etc/motd
```

The undelete exposes the original file from the lower layer. I could edit */etc/motd* within the jail and save my changes, creating a file in the upper layer. Removing that copy adds a whiteout for the lower layer. If I undelete the file I don't restore the edited file; rather, I see the lower layer's */etc/motd*.

A union mount defaults to always creating a whiteout for a removed file, even if the file does not exist in the lower layer. While whiteouts use very little space, if you're using a union mount on an embedded device you need every scrap of space you can scrounge. Set the whiteout option to `whenneeded` to have union mounts only create whiteouts when required.

Now that you can rearrange your on-disk filesystems any way you choose, let's create some filesystems out of pure memory.

Chapter 5: Memory Filesystems

FreeBSD lets you create filesystems backed by system RAM, for short-lived memory disks. Reading and writing from memory is much faster than accessing files on disk, which makes memory-backed filesystems a fantastic optimization for certain applications. As with everything else in memory, though, at system shutdown you lose the filesystem.

FreeBSD supports two different RAM-based disks: *tmpfs* (pronounced “temp f s”) and *memory disks*. While they have similar concepts behind them, the underlying code is completely different, and they serve different roles. Use tmpfs(5) for memory-backed filesystems on long-running systems. Memory disks are more flexible but more suited for short-term use (unless carefully configured) or mounting disk images.

tmpfs(5)

The “tmp” in tmpfs doesn’t mean “temporary.” It literally means *tmp*, as in */tmp*. Use tmpfs(5) for a speedy memory-backed */tmp* and similar filesystems. Don’t deploy tmpfs everywhere you see a path with *tmp* in it, though—remember, data in */var/tmp* is intended to survive a reboot. You might also use tmpfs for application lock files and other ephemeral data where speed would improve performance. While older versions of tmpfs had problems, as of FreeBSD 10 it’s widely deployed and considered production-ready.

When you mount a tmpfs(5), it automatically creates a filesystem.

```
# mount -t tmpfs tmpfs /tmp
```

You could also use an */etc/fstab* entry to have your tmpfs automatically mounted at boot.

```
tmpfs /tmp tmpfs rw,mode=1777 0 0
```

If users are permitted to mount filesystems (by setting the sysctl vfs.usermount to `1`), they can create and mount tmpfs filesystems. Machines with this setting generally belong to a single user, however.

Let’s consider some tmpfs options.

tmpfs Size

Left to its own devices, a tmpfs has a maximum size equal to the amount of available RAM plus the amount of swap space. Copying a sufficiently large file to your tmpfs partition would exhaust all system memory. This would be bad.

Use the mount option *size* to set a maximum size of a tmpfs. Here I create a 1 GB tmpfs, mounted at */mnt*.

```
# mount –o size=1g -t tmpfs tmpfs /mnt
```

If you mount multiple tmpfs filesystems without a maximum size, they will all think that their maximum size is equal to the entire available memory and swap space. This means you could have two half-full tmpfs instances and run out of memory.

One way you can reduce the odds of memory exhaustion is to limit the size of files that can go on a tmpfs with the `maxfilesize` option. Most files in */tmp* or */var/tmp* are tiny, and on most systems none should exceed a few megabytes. Here I set a maximum file size of 10 MB on any file on this tmpfs.

```
# mount -o maxfilesize=10m -t tmpfs tmpfs /tmp/
```

Copy a large file to your tmpfs, and you’ll get a “File too large” error.

If you want to restrict how many files can exist on the tmpfs, consider restricting the number of inodes. While every file and directory needs inodes, the number of inodes does not directly correlate to the number of files and directories. You'll want to experiment with this restriction to see if it works for you and how it needs adjusting.

```
# mount -o inodes=10k -t tmpfs tmpfs /tmp
```

Ten thousand inodes should suffice for */tmp* on most systems.

tmpfs Permissions

You can also control the ownership and permissions on a tmpfs with the `uid`, `gid`, and `mode` options. An actual */tmp* directory should be world-writable with the sticky bit set, so be sure to use the option `mode=1777`. If the tmpfs is for a specific user or application account, assign that user ownership of the tmpfs.

For more complicated memory-backed disks, consider a traditional memory disk.

Memory Disks

A *memory disk* is an ephemeral storage device. Despite the name, a memory disk is not always a chunk of memory being treated as a disk. It can be such a device, but it might instead use a file or swap space as a backing store. The storage device disappears at a reboot.

FreeBSD supports four types of memory disks: *malloc-backed*, *swap-backed*, *vnode-backed*, and *null*.

Malloc-backed disks are pure RAM. Even if your system runs out of memory, FreeBSD will absolutely not page or swap out a malloc-backed disk. Using a large malloc-backed disk is a great way to exhaust system memory. Malloc-backed disks are most useful for swapless embedded devices.

Swap-backed disks are mostly memory, but FreeBSD can shuffle them out to swap space if necessary. This is the safest way to get a high-performance memory disk.

Vnode-backed memory disks are built on top of files on the disk's filesystem. Using a file for backing a memory disk is the safest way to use a memory disk, but it's slower than just having the files on the partition's filesystem—reads and writes have to traverse the memory filesystem, which in turn calls up the file on the disk. Two layers of filesystems are involved. Vnode-backed memory disks are useful for mounting disk images, however, as well as for testing.

A *null* memory disk discards everything sent to it. Any writes are successful, while any reads return only zeroes. I won't cover null devices, as I prefer to lose my data by hand.

Creating and Mounting Memory Disks

The mdmfs(8) utility is a convenient wrapper for creating common memory disks. You can get down and dirty with programs like mdconfig(8) and newfs(8), but `mdmfs` bundles everything up for you. You only need to know the size of the disk you want to use, the type of memory disk, and the mount point. The mdmfs(8) command defaults to swap-backed memory disks, the most common.

Unprivileged users cannot create memory disks, even if they're allowed to mount removable media.

Use the `-s` flag to specify the filesystem size. Here I create a 16 MB swap-backed memory disk on */mnt*.

```
# mdmfs -s 16m md /mnt
```

The `md` option above means “I don’t care which memory disk device node this gets, just give me the next free one.”

You can also view what `mdmfs` does behind the scenes by adding the `-X` option.

```
# mdmfs -X -s 16m md /mnt
DEBUG: running: /sbin/mdconfig -a -t swap -s 16m
DEBUG: running: /sbin/newfs -U /dev/md0
DEBUG: running: /sbin/mount /dev/md0 /mnt
```

All the other `mdmfs` command-line options are about changing the behavior of those other programs.

Memory Disk Memory Use

Traditionally, a swap-back memory disk never returned used memory to the system. Once you wrote to the memory disk, that memory was consumed. If you needed a large memory disk, you had to permanently allocate memory for it. This was one of the reasons FreeBSD included tmpfs(5).

If the filesystem on the memory disk supports TRIM, though, FreeBSD 8.1 and later will return unused memory to the system. Enable TRIM in `mdmfs` with the `-t` flag. TRIM is probably important only for long-lived large memory disks.

mdmfs Options

We have many mdmfs(8) options to tweak the underlying memory disk. Most of us don’t care about the disk geometry, or the emulated disk rotation speed, or the number of blocks per inode in the temporary filesystem. Here are the most commonly useful options.

To use a malloc-backed memory disk, add the `-M` flag.

To put a UFS filesystem on an existing file and mount it, add the `-F` flag and the path to the file. This is destructive—any filesystem that exists on the disk will be destroyed. If you want to mount an existing UFS image file without destroying the contents, add the `-P` flag.

```
# mdmfs -PF ufsfs.img md /media/
```

Rather than using the next available memory disk device node, you can specify a device node at the command line. Here, I tell `mdmfs` to use device md13.

```
# mdmfs –s 16M md13 /mnt
```

The new filesystem uses the newfs(8) defaults, which might not really make sense for a memory disk. Specifically, soft updates are not useful on an ephemeral filesystem. Neither is a noasync mount, as you can't recover this disk after a crash. Disable soft updates with -s. Give any mount options with -o.

```
# mdmfs -S -o async -s 16m md /mnt
```

Specify a user and a group to own the memory disk with -w. You must specify both the owner and the group—listing only one or the other is an error. Similarly, use the -p flag to define permissions on the new device.

```
# mdmfs –p 700 -w mwl:mwl -s 16m md /home/mwl/mnt
```

The user has full ownership of this memory device.

Destroying Memory Disks

While mdmfs simplifies creating memory disks, you'll need mdconfig(8) to destroy them. Destroying the memory disk frees the memory used by the device. To find the disk device, run mount and find the partition containing the memory disk.

```
/dev/md86 on /mnt (ufs, asynchronous, local)
```

The */mnt* partition is built on memory device */dev/md86*. Unmount the partition, and then destroy the device with mdconfig(8).

```
# umount /mnt
# mdconfig –d –u 86
```

Unmounting with umount(8) works exactly like any other UFS filesystem. Use mdconfig(8) to directly manage memory devices. The -d flag means destroy, and-u gives a device node number. The above example destroys memory disk /dev/md86. The memory used by the device is now available.

Viewing Memory Disks

Memory disks can accumulate, and after months you might forget which disk does what and how it's configured. The -l option to mdconfig displays all live memory disk device nodes. Add the -v flag to show the details on each.

```
# mdconfig -lv
md0 swap 16M
md1 vnode 690M /home/mwl/ FreeBSD-disc1.iso
```

This host has two memory disks, one swap-backed 16 MB filesystem and one attached to an ISO.

Filesystems in a File

While mdmfs lets you use a filesystem on a file, the file must exist first. Use dd(1) to create files for filesystems. The `dd` command is a generic tool for copying and altering files, but these days it’s most commonly used to create disk images.

Using `dd` demands using a little math. It copies a certain number of blocks of a given size. You need to figure out how many blocks of that size you need to generate a file of the size you want. There’s a whole game to selecting a block size, and with some research you can choose the most efficient block size, but I find using 1 MB blocks gives reasonable performance on modern hardware. So, if you want a 1 GB disk image, how many 1 MB blocks do you need? Hardware manufacturers would say 1,000, while most sysadmins would say 1,024. I’m going to duck that question entirely and create a 512 MB file.

Once you know how many blocks you need, you need a place to get those blocks from. The fastest source of irrelevant stuff is the empty device, */dev/zero*. Here I create a 512 MB image file full of stuff.

```
# dd if=/dev/zero of=ufs2.img bs=1m count=512
```

The `if=` gives the input file, or the source I’m copying from. The `of=` defines the output file, and `bs` is the block size, while `count` is the number of blocks. So I’m copying 512 1MB blocks of… absolutely nothing. The result is a 512 MB file.

If you need large files, many gigabytes or even terabytes, `dd` can run very slowly. FreeBSD lets you create an empty file of arbitrary size, or a *sparse file*. A sparse file is marked as having a certain size, but it only takes up a single block on the disk. Sparse files grow once you put stuff in them. Use the `-s` option to truncate(1) to create a sparse file. Give it one argument, the size of the file.

Here I create a 512 MB sparse file.

```
# truncate –s 512M sparse.img
```

The result is a file that claims to be the desired size.

```
# ls -lh sparse.img
-rw-r—r— 1 root mwl 512M Sep 2 13:39 sparse.img
```

Using ls(1) or du(1) to get the number of disk blocks used for this file, however, gives a different story.

```
# ls -s sparse.img
1 sparse.img
```

This file uses one filesystem block. On this disk, that’s 512 bytes.

Sparse files are great. You can fit trillions of these tiny files in a lowly gigabyte of disk space. When you write data to sparse files, however, they expand. You can fill up your physical disk without creating any new files just by using sparse files to back filesystems. Creating a filesystem on this file with `mdmfs` inflates the underlying file, as does every disk write.

Sparse files never shrink. They can only grow. Even so, if you need to fit a mostly empty 10 TB disk image on a 2 TB ZFS pool, you might find them useful.

Memory Disks and /etc/fstab

Listing memory disks in */etc/fstab* tells FreeBSD to automatically create them at boot. The device name is `md` and the filesystem type is `mfs`. In the options column, list any command-line mdmfs(8) options used to create this device, as shown here.

```
md /mnt mfs rw,-s16m
```

When mounted from */etc/fstab*, memory disks are world-writable and have the sticky bit set. These permissions are appropriate for */tmp*, but not so much for anywhere else. Specify the permissions as a mount option, but remember that */etc/fstab* does not tolerate extra spaces. Separate the various options with commas, and eliminate any spaces between the command-line options and their arguments.

```
md /home/mwl/mnt mfs rw,-p700,-wmwl:mwl,-s16m
```

Disk Images

You can mount UFS disk images with `mdmfs`, but it's far more common to mount ISO or UDF images. While FreeBSD's tar(1) can extract files from ISO images thanks to libarchive[1], that's often overkill. Also, ISO images are read-only. You can inflict fumble-fingered damage on files extracted with tar(1), but ISOs resist tampering. Also, libarchive cannot extract files from pure UDF images. Sometimes your simplest choice is mounting a disk image.

To mount a disk image, use mdconfig(8) to attach the file to a vnode-backed memory device. When you give only a filename as an argument, `mdconfig` assumes you want to use the file as a filesystem.

```
# mdconfig FreeBSD-10.2-RELEASE-amd64-disc1.iso
md1
```

In response you get the device node for this memory disk, *md1*. Now use the proper mount command to mount this filesystem.

```
# mount –t cd9660 /dev/md1 /media
```

Remember that mount(8) assumes that all filesystems are UFS unless told otherwise.

When you're done accessing the disk image, unmount the image and destroy the memory disk device as you would any other memory device.

Disk Images and /etc/fstab

You can specify a memory device in */etc/fstab* and attach it to a disk image. Give the filesystem type as *mfs*. Specify the filesystem is read-only, and use the `-F` flag to give a path to the disk image. At boot this entry creates memory device *md10*, attaches it to */iso/stuff.iso*, and mounts it at */stuff*.

```
md10 /stuff mfs ro,-F/iso/stuff.iso 0 0
```

The system will need access to the disk image early in the boot process. If the image isn't available, the process will hang. You might need to use `noauto` for disk images in a home directory, for example.

By combining the `noauto` mount option and autofs (Chapter 12), the system can mount the image only when needed and automatically unmount it when it's not in use.

You can also use `mdconfig_md` entries in */etc/rc.conf* to configure memory disks at boot, and then use simpler */etc/fstab* entries, but that method is less flexible than pure */etc/fstab* entries.

That takes us through every filesystem that runs on the local machine. Let’s head out into network-aware filesystems.

[1] All hail libarchive!

Chapter 6: Network File System

FreeBSD supports the venerable Network File System (NFS) out of the box. NFS lets you share mount points and directories on one machine with other hosts. Entire books have been written about NFS, its various versions, and the good and bad points. This book doesn’t take you deep into the inner workings of NFS, but instead focuses on establishing and managing NFS services.

NFS uses a client-server model. A server offers filesystems to other computers. This is called NFS exporting, and the offered filesystems are called *exports*. NFS clients can mount exports much as they would mount any other filesystem.

NFS was not designed as a secure protocol. Do not put NFS servers on the Internet without a packet filter or firewall. Merely restricting access at the NFS level is utterly insufficient—you must prevent random Internet hosts from poking at the host’s Remote Procedure Call (RPC) services. If you use one of FreeBSD’s built-in packet filters, be sure to restrict NFS access by IP address as well as port number.

Additionally, NFS is not encrypted. Anyone with packet sniffer access to your wire can see files as clients access them. Encrypting NFS requires Kerberos, which could be an entire book in itself.

Use NFS to share files between Unix-like systems, or with non-Unix clients that have NFS capability.

NFS Versions

NFS comes in three versions: NFSv2, NFSv3, and NFSv4. To a sysadmin, versions 2 and 3 are very similar. Most hosts can negotiate which of these versions to use.

NFSv2 is a fairly minimal implementation, dating from the years when people were delighted just to get file sharing working at all, regardless of performance.

NFSv3 contains many incremental improvements over NFSv2 and boasts greatly improved performance. Most of these improvements don’t require special configuration, however. You can choose to tweak settings, but you don’t have to specifically enable NFSv3 features.

NFSv4 is an entirely different beast, and breaks many of the long-standing rules of NFS. When people say “NFS,” they almost certainly mean versions 2 or 3. I’ve also heard versions 2 and 3 referred to as “traditional NFS.” Someone who means NFSv4 generally says “NFSv4.” This chapter starts with versions 2 and 3. Once you understand them, we’ll cover the changes that make NFSv4 unique.

While almost every Unix-like operating system supports some version of NFS, each of those operating systems implements it slightly differently. If you have multiple operating systems on your network, don’t be shocked if some of the hosts need tweaking to interoperate nicely with the other hosts. The FreeBSD-net@FreeBSD.org mailing list archive contains discussions on how to get the best performance out of just about every possible combination of operating system and NFS versions. If you have trouble, check there first.

NFSv2/NFSv3 Protocol

To a sysadmin, NFS versions 2 and 3 appear very similar. FreeBSD uses the same server software for both versions of NFS.

Traditional NFS is stateless. An NFS server does not track how a client is connecting or what it’s accessing. Rebooting an NFS server won’t crash the clients. The clients won’t be able to access files on an unavailable NFS server, but once the server returns it’ll pick up where it left off. Other network file sharing protocols are not always so resilient.

Statelessness causes its own problems, however. For example, clients cannot know when a file they currently have open is modified by another client. And a net-booted diskless host will probably hang when the server disappears.

We’ll configure the server first, then discuss the client.

Enabling the NFS Server

Enable NFS with the following */etc/rc.conf* options. While not all environments require all of these features, enabling them provides the broadest range of NFS compatibility and decent performance.

```
nfs_server_enable=YES
rpcbind_enable=YES
mountd_enable=YES
rpc_lockd_enable=YES
rpc_statd_enable=YES
```

These services work together to provide NFS exports. Those learning NFS get the best results by rebooting the server to ensure everything starts in the correct order. People familiar with NFS can start the `nfsd`, `lockd`, and `statd` services to activate NFS. The sockstat(1) program should show `rpc.lockd`, `rpc.statd`, `mountd`, and `rpcbind` listening to the network. If you don't see all of these programs running, check */var/log/messages* to see why they didn't start.

The key NFS program is rpcbind(8). If `rpcbind` won't start, try running it in debug mode with `-d`. It won't detach from the terminal, but prints status messages instead. Terminate the debug-mode rpcbind(8) with `CTRL-C`, like any other program.

How NFS Services Work

NFS exports require support in the kernel. While the NFS server and client are compiled into FreeBSD's default kernel, if your custom kernel lacks that support, enabling the NFS server loads the kernel module.

Clients start a connection by contacting the server's rpcbind(8) server. This daemon maps RPC requests into local network addresses and ports. Clients contact `rpcbind` to request the IP and port of the NFS mounting daemon, mountd(8).

The mountd(8) daemon accepts and responds to requests from clients. It's what most people would think of as "the NFS server." It listens to a random high-numbered port. The only way a client can find the running `mountd` daemon is to ask the rpcbind server.

While rpc.lockd(8) and rpc.statd(8) aren't strictly necessary, they make living with NFS much nicer. The rpc.lockd(8) process ensures smooth file locking operations over NFS, so that clients can get an exclusive lock on a file. The rpc.statd(8) process monitors client connections. When an NFS client disappears, `rpc.statd` frees up resources dedicated to that client. These services impose a small amount of state on an otherwise stateless protocol.

NFS Server Options

You can change how the NFS server connects to the network, which versions of NFS it supports, and how many servers it runs. You can also fall back to the older NFS server if needed. Set all of these as options to nfsd(8), using the `nfs_server_flags` *rc.conf* option.

NFS can work over TCP or UDP. UDP is the traditional NFS transport protocol. TCP works better over lossy networks, and can better cope with varying network speeds. TCP is a stateful protocol, however. An interrupted TCP connection won't transparently resume. Use the `-u` flag to enable UDP and `-t` for TCP. FreeBSD uses TCP by default but enables both. If you add your own mount flags, you must specify which protocol you want the mount to use.

The NFS server, by default, listens to all IP addresses on a machine. This causes problems with UDP-based NFS. UDP is stateless, so replies to NFS requests can come from any IP address on the server, possibly confusing the clients. When an NFS server has multiple IP addresses, configure the NFS server to accept connections only on a single IP with the `-h` option and the desired IP.

Servers that process NFS traffic from many clients might need additional instances of nfsd(8) to support those clients. The `-n` flag specified how many copies of `nfsd` FreeBSD should start.

This *rc.conf* entry tells FreeBSD to support both UDP and TCP connections, bind to the IP address 203.0.113.99, and run six copies of `nfsd`.

```
nfs_server_flags="-uth 203.0.113.99 –n 6"
```

FreeBSD 9.0 and above has completely new NFS kernel support, written so that FreeBSD could support NFSv4. It's possible that the new server has bugs (although it was extensively tested). By using the `-o` flag, you tell `nfsd` to use the old NFS server. One way to determine if a problem is specific to a FreeBSD NFS server or somewhere else is to reproduce the problem with the old NFS server. If you find an NFS issue that doesn't exist in the old NFS server, be sure to file a bug report.

Your server is now ready to export filesystems or directories.

Export Configuration

You can configure exactly what a server may export to which clients. You could export all directories and filesystems on the entire server, but that's begging for people to take advantage of your server. If someone can edit any file on the server, she can also edit the exports configuration. An ideal NFS configuration permits as little access as possible while letting the server fulfill its duties and the clients reach necessary files. While clients might need the files under */home*, they probably don't need to remotely mount the NFS server's root filesystem.

FreeBSD lets you configure exports in two different ways. You can configure all exports in */etc/exports*. If a server uses ZFS, however, you can also configure exporting each dataset via the **sharenfs** property. A ZFS-based server creates the file */etc/zfs/exports* out of the property settings. Never edit */etc/zfs/exports* by hand—always use the dataset **sharenfs** property.

Choose one method of managing your NFS exports. Either edit */etc/exports*, or use `zfs` and */etc/zfs/exports*. Using both methods simultaneously will at best confuse and frustrate you, and at worst break everything. If you're stuck with both UFS and ZFS on one host, you might decide to use */etc/exports* for UFS and **sharenfs** for ZFS—you've already set yourself up for confusion by mixing filesystems, so why not go all the way?

Exports Entries

While the following discussion focuses on "the exports file," almost everything applies to exports managed with ZFS as well. We'll discuss the differences in "Managing NFS with ZFS Properties" later this chapter. Understanding those limitations on ZFS NFS requires understanding how to configure NFS using the traditional */etc/exports* method, however.

Each exports entry has up to three components.

- Directories or partitions to be exported
- Options and permissions on that export
- Clients that may connect

Each combination of clients and a disk device can have only one line in the exports file. If your NFS server has only one giant partition, as in the default FreeBSD UFS install, and you want to export both */home* and */usr/ports* to your clients, they must both appear on the same line. The two exports will have the same options and identical permissions.

NFS mounts do not cross partition boundaries. If a host has separate UFS partitions for */usr* and */usr/src*, exporting */usr* won't automatically export the */usr/src* partition.

Of the three parts of an exports entry, only the directory is mandatory. An exports line cannot contain symlinks or periods. Not sure if a directory path has a symlink? Run `pwd` in the directory to get that directory's true location, regardless of how you got there.

If I wanted to export my home directory to the entire Internet, I could use an */etc/exports* entry including only:

```
/home/mwl
```

This has no options and no restrictions on which hosts may mount the share. Your firewall or packet filter should protect you from random Internet probes, but it's still poor practice.

After changing the exports file, restart or SIGHUP `mountd`.

```
# service mountd reload
```

If `mountd` finds any problems, it makes a log entry in */var/log/messages*. The most helpful part of these log messages is the line number; `mountd` doesn't usually give any more detail than that. My most common mistake in */etc/exports* is using a path with a symlink.

NFS and Users

Traditional NFS communicates file ownership and permissions by UID numbers. A file isn't owned by user **mwl**—it's owned by user 1001. If those UID numbers map to the same username, great! If not, you might have a problem. Large NFS deployments normally have some means of synchronizing usernames and UIDs across all systems, such as LDAP or even `rdist`.

On a network where untrusted users have administrative privileges on their own machine, and could create accounts with any UID, most sites implement Kerberos or some other means of authentication control. On a smaller network, such as my test lab, synchronizing the password files on all machines usually suffices.

The **root** user is handled somewhat differently. An NFS server can't trust **root** on other machines to execute commands as **root** on the server. An intruder who penetrates an NFS client shouldn't automatically get **root** on the NFS server with it!

NFS maps requests from **root** on a client machine to a user with the UID and GID of -2 on the server. Traditional systems had 16-bit user IDs, so this became 65534, the user ID for the **nobody** account. That's why the highly unprivileged **nobody** user was originally created.

The **nobody** account seemed useful for a lot of server programs, though, so many programs appropriated **nobody** for their own use. Multiple security entities simultaneously using **nobody** creates security access issues. While most programs now expect to run under their own unprivileged user account, some random software on one of your systems probably runs as **nobody**. Even if you audit all your machines for software that runs as **nobody**, some package installed by a minion next week, next month, or next year will probably want to run as **nobody**. (I consider the **nobody** user tainted, and don't want it on my network.)

Modern Unix-like operating systems use 32-bit UIDs, so a client's **root** account maps to UID 4,294,967,294 on the server. FreeBSD has no such account in */etc/passwd*. I recommend creating an unprivileged user, **nfsroot**, explicitly for use by NFS for **root**. If possible, assign that user UID 4294967294.

Explicitly assign a root-to-nfsroot mapping with the `-maproot` option. Here I map the user **root** to this assigned user.

```
/home/mwl –maproot=nfsroot
```

If you truly want the client's root account to have root privileges on the NFS server, you can use `–maproot=0`. Diskless machines need this, so that they can put their root filesystem on the server.

You can assign group memberships for the **nfsroot** user in the exports file. This group membership can differ from that assigned in */etc/group*, letting you use share-specific group permissions. Specify groups by name or GID. Use colons to separate the groups from the username. Here, the **nfsroot** user is also a member of **www** and **staff**.

```
/home/mwl –maproot=nfsroot:www:staff
```

If you want to remove all group membership for this user for a particular share, give a colon and no groups after the NFS mapped root user.

```
/home/mwl –maproot=nfsroot:
```

Instead of remapping only the **root** user, you can remap all NFS users to a single account with the `–mapall` option. This gives all users identical access.

```
/home/mwl –mapall=nfsroot:
```

Assign group privileges as you would for `–maproot`.

You cannot arbitrarily remap user accounts to each other. In complex environments, be sure you synchronize your user accounts and UIDs on all machines on your network.

One thing to remember is that users in NFS can belong to no more than 16 groups. (Some older operating systems limit their NFS servers to even fewer groups.) Some operating systems can break that limit, but they break the NFS protocol in doing so. If a user can't access files with group-based access control, check the number of groups that they're in.

Exporting Multiple Directories

You might want to export multiple directories on one partition. List all directories exported to the same clients on one line in */etc/exports*, separated by spaces. Here I export four directories on this system's root partition.

```
/home/mwl /usr/src /var/log /var/db –maproot=nfsroot
```

Clients may mount any of these directories, and requests from root get mapped to the user **`nfsroot`**.

Perhaps you want clients to be able to mount any directory on a partition. Permit this with the `-alldirs` option. Here, users can mount any directory on the */home* partition.

```
/home -alldirs
```

The `-alldirs` option works only when the directory is a partition mount point. I could not specify a directory that's not a mount point and permit mounting any directory beneath it.

Long Lines

Do not use any identifiers, delimiters, commas, or other separators between the various parts of the line. Tabs are okay.

In the last example, having each directory on its own line would be easier to read, but as they're all on the same partition, we can't do that. The FreeBSD NFS folks could redesign */etc/exports* in a more structured manner, but then FreeBSD would have an exports file incompatible with any other Unix-like operating system. Yes, organizations frequently share the exports file between machines, even machines with different operating systems.

You can break lines up with a backslash, however. Once an exports line gets long enough, line breaks are almost mandatory.

```
/home/mwl \
/usr/src \
/var/log \
/var/db \
-maproot=65533
```

However you arrange your exports file, it's not fun to read. Choose your preferred pain.

Restricting Clients

By default, any host anywhere in the world may access exported NFS shares. To restrict access, list permitted hosts at the end of the */etc/exports* entry. Here I restrict access to my home directory to a single IP address.

```
/home/mwl 203.0.113.111
```

Restrict NFS mounts to clients on a specific network by using the `-network` and `-mask` qualifiers.

```
/home/mwl -network 203.0.113.0 -mask 255.255.255.0
```

Here, any host with an IP address beginning in 203.0.113 can access my home directory.

Those of you familiar with slash netmask notation can use that instead.

```
/home/mwl -network 203.0.113.0/24
```

The two examples above are functionally identical, but I find the second more readable.

You can have only one network statement per entry. To export to multiple networks, use multiple lines.

```
/home/mwl -network 203.0.113.0/24
/home/mwl -network 198.51.100.0/24
```

IPv6 addresses work exactly the same, of course. Here I let an IPv6 block access my NFS share.

```
/home/mwl -network 2001:db8:bad:code::/64
```

You can also specify hosts by hostnames or NIS netgroups. List each host by either short or long name at the end of the line, separated by spaces. Here I give three hosts access to the share.

```
/home/mwl -maproot=nfsroot blue agouti rex
```

Using hostnames creates dependencies on name resolution. Losing domain name service (DNS) is sufficiently unpleasant without adding NFS failures to that. Additionally, these names are not dynamic. The NFS server looks up the IP address of each host when you restart `mountd`. Changing a client's IP means both reloading the name service and restarting `mountd`.

Using individual IP addresses instead of hostnames in the exports file means that when you change a client's IP, you must change its entry in */etc/exports* and restart `mountd`.

Either way, managing per-host NFS assignments takes more work. Assign NFS permissions as broadly as possible without compromising security.

Invalid Combinations

One combination of a partition (or directories on a single partition) and a host (or group of hosts) can have only one line in */etc/exports*. A host with only one filesystem cannot use these exports statements. Remember, FreeBSD's installer creates one large filesystem by default when using UFS.

```
/usr/src –maproot=0 203.0.113.5
/home –maproot=nfsroot 203.0.113.5
```

We're trying to export two directories on the same partition, to the same host, with different options. This is invalid. You could export both directories to the same host with the same options, or you could move the two directories to separate partitions.

Similarly, you couldn't export these to different hosts with different options, as shown here.

```
/usr/src –maproot=0 203.0.113.5
/home –ro 198.51.100.6
```

This would work if */usr/src* and */home* were on different partitions.

Exporting only mount points prevents attempts to create invalid combinations of subdirectories and options. An NFS server should always have multiple partitions, rather than the single large partition of FreeBSD's default UFS install. If you use ZFS, manage NFS with ZFS properties as discussed later.

Other Server Options

You might want to give clients read-only access to an NFS share. The `-ro` option permits this.

```
/home –ro –network=203.0.113.0/24
```

Sometimes you'll have an NFS export configured that isn't always valid. You might export */media* to share your DVD drive, for example. If you have a disc mounted when you restart `mountd`, everything is fine. If you don't have any media in the drive, however, `mountd` logs complaints. You don't need those complaints—you know what the problem is. Add `-`

`quiet` to an export line to silence common warnings.

```
/media –quiet
```

You can combine multiple NFS server options on a single line by removing the leading hyphens and separating them with commas. Do not use whitespace. Here I set a share to read-only, set a `maproot` user, and silence common warnings.

```
/home –ro,maproot=nfsroot,quiet
```

As you cannot have whitespace between options, use the equals sign (`=`) to specify the user for the `maproot` option.

Now that you can configure your NFS server and its exports, let's play with performance a little.

Managing NFS with ZFS Properties

Using zfs(8) to manage NFS has distinct advantages. You can configure NFS on a per-dataset basis, and you don't need to restart `mountd` after each change. Command-line configuration is easier to automate, and many folks find it easier as well.

Use the **`sharenfs`** property to enable, disable, and configure NFS exports. Set this property to *on* to globally share a dataset and all its descendants. This is equivalent to listing the dataset on its own in */etc/exports*. Anyone can mount it or any of its child datasets, with no restrictions and no options.

```
# zfs set sharenfs=on zroot/home
```

Similarly, set it to *off* to unshare the dataset.

You probably want some NFS options on this export, however. Set **`sharenfs`** to the options you want for this share. Here I establish a maproot user and restrict clients to a single network.

```
# zfs set sharenfs="-network 203.0.113.0/24 -maproot=nfsroot" zroot/home
```

The problem with using ZFS to manage your NFS exports is that you must use the same options for all permitted hosts. That is, if most of your clients use the `-maproot=nfsroot` option, but you have one problem host that legitimately needs `-maproot=root`, zfs(8) can't help you. You must configure such exports via */etc/exports*. Similarly, you can define only one permitted network with ZFS properties.

I encourage you in the strongest possible terms to choose a single method of managing NFS and stick with it. Simpler configurations can use ZFS properties, but more complicated NFS setups probably want */etc/exports*. Using both */etc/exports* and **`sharenfs`** causes confusion. Really, don't do it.[1]

NFS Clients

Using the NFS client is much simpler than configuring the server. FreeBSD has the `nfs_client_enable` option in */etc/rc.conf*, but FreeBSD automatically starts all NFS client functions when you try to mount a share over the network.

Attach NFS exports to your client with mount(8), much like you would for any local filesystem. Instead of using a device name, use the NFS server's host name or IP address and the directory you want to mount. Here I mount the directory */home/mwl* from the server **agouti** onto the local directory */mnt*.

```
# mount agouti:/home/mwl /mnt
```

This mount uses whatever options the server enforces. If the server exports the directory read-only, the mount is read-only. You can add your own mount options, as we'll see below.

Available Mounts

One obvious question for an NFS client would be "what can I mount from that server?" The showmount(8) command can list all exports available to a client. Give the `-e` flag and the name of the NFS server. Here I ask the server **agouti** what it's willing to share with this client.

```
# showmount -e agouti
Exports list on agouti:
/usr/src 203.0.113.0
/usr/obj 203.0.113.0
/home 203.0.113.0
```

The `showmount` command does not display any server-side options, such as `-ro` or `-maproot`. Those details are not readily available to clients, although read-only exports are pretty easy to detect with touch(1).

NFS Mount Options

FreeBSD uses conservative NFS defaults, so that it can interoperate with any other Unix-like operating system. If your environment uses only newer systems, try various mount options to see if they improve system performance. Specify mount options with `-o`, as shown below.

Modern FreeBSD uses TCP by default. TCP has all sorts of automatic scaling features, and adjusts itself to provide reasonable throughput. Not all NFS hosts support TCP, however. To serve clients that can't do NFS over TCP, and to access shares on an NFS server that doesn't handle TCP, you'll need to use UDP mounts. The mount option `udp` enables UDP.

NFS servers can disappear from the network more readily than hard drives. Programs using the NFS share can hang until the NFS server returns, which might never happen. Use *interruptible* mounts to let you stop programs when the filesystem disappears. You can interrupt processes hung on unavailable but interruptible NFS mounts with CTRL-C. Make mounts interruptible with the `intr` option.

By using a *soft* mount, FreeBSD will notify programs that the file they were working on is no longer available. What programs do with this information depends on the program, but they will no longer hang forever. Enable soft mounts with the `soft` option.

You can also use the `rw` and `ro` options for read-write and read-only mounts.

Putting everything together, you could use soft, interruptible TCP mounts like so.

```
# mount -o soft,intr,tcp agouti:/home/mwl /mnt
```

You could put this in */etc/fstab* like so.

```
agouti:/home/mwl /mnt nfs rw,tcp,soft,intr 0 0
```

While simple NFS is pretty straightforward, you can spend many hours tuning it.

NFS Performance

NFS performance is an arcane topic, and best handled on a case-by-case basis. Different combinations of server and client might work best with wildly different performance settings. Don't be afraid to play with NFS settings and configurations to see if you can improve throughput for your environment. Most NFS tuning happens at the client; the server responds to the client's tuning requests.

To adjust performance, look at the mount options `rsize`, `wsize`, `wcommitsize`, `readahead`, and `readdirsize`. While you can set these options to any value, FreeBSD clips most of them at a maximum size for interoperability or protocol reasons.

The `rsize` and `wsize` options dictate the size of read and write requests. The `readdirsize` value is the size of directory read requests. With UDP mounts, the maximum value on NFSv2 is 8K, while NFSv3 permits 16K. NFSv4 doesn't use UDP.

NFSv3 cannot specify separate maximum sizes for TCP and UDP `rsize`, `wsize`, and `readdirsize`. Many servers cannot handle requests greater than 16K, however, so FreeBSD clips these values to 16K.

The server uses `readahead` to guess what the client's going to ask for next and load it up from the disk before the client asks for it. Play with `readahead` if you normally use NFS to access large files.

To see the actual values used by a mount, use `nfsstat -m`. Then measure performance, experiment, test, and test some more.

NFSv4

Now that you understand how to use NFS versions 2 and 3, let’s see how version 4 throws all the pieces up in the air and puts them together in a different way. NFSv4 adds three core functions to NFS: *access control lists* (ACLs), *delegations*, and *referrals*. It also adds state and changes how it identifies users.

Filesystem *ACLs* allow extended permissions settings, letting people add many different layers of permissions to files. ACLs are mostly useful for systems that offer file shares over both NFS and CIFS. This is the primary use case for FreeBSD’s NFSv4 implementation. Chapter 11 discusses ACLs in detail.

Delegations allow a client to edit a file locally and send the updates to the server. FreeBSD’s NFSv4 client supports delegations, but the server does not.

Referrals allow servers to transparently redirect requests to another server. Again, FreeBSD’s NFSv4 client supports referrals, but the server does not.

Unlike older versions of NFS, version 4 is stateful. The client bundles up a whole bunch of RPC calls in a single request and ships them to the server as a single unit, for improved performance. The server processes the requests in order. If one of the requests fails, the server cancels the rest of the requests in the queue and sends the results to the client. NFSv4 always runs over TCP.

Finally, NFSv4 adds the nfsuserd(8) daemon to identify users.

nfsuserd(8)

Older versions of NFS use an account's UID number to identify the user. NFSv4 identifies users by both their UID and a combination of the username and the NFSv4 domain name.

The NFSv4 domain name is a string of text that usually corresponds to the host's domain name. Left on its own, FreeBSD grabs the host's domain name and uses that as the NFSv4 domain name. If a host doesn't have a domain name, however, or if your NFS hosts have different domain names, you'll need to set the NFSv4 domain manually.

Different domain names can impact even small networks. My home network uses one domain for production[2] and a separate domain for testing. Even in my house, I need to pick an NFSv4 domain and tell my hosts about it.

Both the client and the server use the nfsuserd(8) daemon process username information. Use the `-domain` command-line argument to tell NFSv4 which domain to use. Your domain shouldn't change often, so set it in */etc/rc.conf*. Here, I both enable `nfsuserd` and tell it to use the **`mwl.io`** domain.

```
nfsuserd_enable=YES
nfsuserd_flags="-domain mwl.io"
```

Additional configuration depends if the host is an NFSv4 server or client.

NFSv4 Servers

Configuring an NFSv4 server requires enabling required services and configuring filesystems to export.

Enable the NFSv4 Server

An NFSv4 server needs the following *rc.conf* options set.

```
nfs_server_enable=YES
mountd_enable=YES
nfsv4_server_enable=YES
nfsuserd_enable=YES
```

NFSv4 listens to port 2049, making it possible to put a packet filter between an NFSv4 client and server, so long as port 2049 is open. You still need the general NFS support functions in nfsd(8), though, as well as mountd(8).

Hosts can serve and mount all three versions of NFS simultaneously.

NFSv4 Exports

Unlike older versions of NFS, NFSv4 exports a single filesystem and all directories beneath it. You can't export multiple disconnected filesystems with NFSv4. Each NFSv4 server has a single export root directory. Clients can mount any directory beneath that mount point. If you want to export disconnected parts of a server's directory tree, use null mounts (Chapter 4) to attach them to the NFS hierarchy.

Configure your NFS export in */etc/exports*, on a line starting with *V4:*. Then give the root directory and any NFS options. NFSv4 accepts the network restriction options from older versions of NFS, as well as the Kerberos security options, but other options like `-ro` or `-alldirs` have no effect. Here I export */home* to my network via NFSv4.

```
V4: /home -network 203.0.113.0/24
```

While you get only a single root directory, you can have multiple V4 export lines to support multiple networks, provided they all have the same root.

If you export a ZFS dataset, all of its child datasets are automatically exported. You must explicitly use the **`sharenfs`** property to de-export child datasets.

NFSv4 clients do not use the mount protocol required by older versions of NFS, but the server still needs to run mountd(8) to provide some supporting services. To make */etc/exports* changes take effect, use `service mountd reload` or `pkill -1 mountd`, exactly as you would for older NFS versions.

NFSv4 Clients

Enabling NFSv4 on an NFS client requires only running nfsuserd(8). All other dependencies get started automatically. Enable nfsuserd(8) in `/etc/rc.conf`.

```
nfsuserd_enable=YES
```

NFSv4 uses the host's unique identifier from gethostid(3). FreeBSD automatically generates this ID at boot, unless someone has deliberately disabled it. If the client's `rc.conf` includes `hostid_enable=NO`, you can't use NFSv4 without possibly corrupting files.

You can now mount NFS shares via NFSv4. FreeBSD defaults to trying to mount NFS shares with version 3, falling back to version 2 if that fails. The automation doesn't even consider NFSv4, so you must specify the NFS version as a `mount` option.

```
# mount -o nfsv4 agouti:/mwl /mnt
```

It is possible to use NFSv4 without running `nfsuserd`, but you'll get incorrect permissions on files. If you're running `nfsuserd` but still get incorrect permissions, check the NFSv4 domain name.

While FreeBSD's NFSv4 server does not support delegations, the client does. NFSv4 clients can edit files locally, reducing bandwidth and improving performance. If a client asks to access a file delegated to another client, however, the NFS server must be able to poke the delegated client and retrieve the current file. This is called a *callback*. To enable delegation support on a FreeBSD NFSv4 client, enable the NFS callback daemon nfscbd(8).

```
nfscbd_enable=YES
```

While a client can mount NFSv4 exports without using `nfscbd`, the client won't get any delegations.

While you can tune NFS to meet almost any situation, it's not the only network storage protocol out there. Let's move on to CIFS.

[1] Around now I usually come out in a cheesy long black hooded robe, point a single chubby finger at you, and say in the deepest voice I can manage: "Doom. Doooom…"

[2] "Production" on a home network being defined as "anything whose loss would interrupt Netflix."

Chapter 7: Common Internet File System Client

The typical office shares network drives via Microsoft's Common Internet File Sharing (CIFS) protocol. CIFS also gets called the Windows "Network Neighborhood," Server Message Block (SMB), and "why can't I map that drive?" While originally available only on Microsoft operating systems, these days anything with even modest aspirations as a corporate file server supports SMB and CIFS.

Strictly speaking, SMB is a lower-level protocol. Microsoft considers CIFS a particular dialect of that protocol.

FreeBSD includes the smbutil(8) program to find, mount, and use CIFS shares as a CIFS client. FreeBSD does not include a CIFS server in the base system, but the open source CIFS server Samba (www.samba.org) works very well on FreeBSD. Samba is a topic that fills books much larger than this one, so we won't go there.

Use FreeBSD's CIFS support to interoperate with existing Windows infrastructure. You wouldn't deploy CIFS to support Unix systems—use NFS for that instead.

Prerequisites

You need a few pieces of information before you can even try to mount a CIFS share. First, get a valid username and password with access to your share, along with the Windows domain name. Your host needs the ability to find the fileserver; the IP address of the Windows DNS server will do, or alternately the hostname or IP of the fileserver you want to mount a share from.

On the FreeBSD side, you'll need to load the *`smbfs.ko`* kernel module. This brings the modules *`libiconv.ko`* and *`libmchain.ko`* along for the ride. You can load these automatically at boot time with a *`/boot/loader.conf`* entry.

```
smbfs_load=YES
```

With this ready, you can configure CIFS.

CIFS Configuration

Unlike NFS or iSCSI, you can only manage CIFS with a configuration file. CIFS is potentially far more complex than either of those. The main configuration file is *`/etc/nsmb.conf`*, but users can have personal configurations in *`$HOME/.nsmbrc`*. Any configuration in *`/etc/nsmb.conf`* overrides a user's personal settings. Both files use the same configuration syntax and keywords, so while this book refers to *`nsmb.conf`*, a user can perform the same type of configuration in their personal config file.

The configuration file is divided into sections. Each section starts with a label, in square brackets. Settings that apply to every CIFS connection are in the [default] section. You'll create other labels to provide configuration settings by server, username, and share, like this.

[*server*]

[*server*:*user*]

[*server*:*user*:*share*]

Settings that apply to an entire server go into a section named after that server. Settings that apply only to a specific user on a server go in a section named after the server and the user. Settings that apply only to a single share go in a section named after the server, the user, and the share name.

Configure FreeBSD's CIFS client using the values from the CIFS host. You cannot log into a Windows host with your FreeBSD account credentials.

nsmb.conf Keywords

Configure CIFS by putting keywords and values in the appropriate sections. Assign values to keywords with an equals sign (`=`). While nsmb.conf(5) lists every valid keyword, here are the ones you'll almost certainly need.

workgroup

The *workgroup* keyword specifies the name of the Windows domain. Workgroups were the predecessor to domains. Almost all CIFS servers have a Windows domain.

```
workgroup=BigOrg
```

nbns

Set the IP address of a NetBIOS (WINS) nameserver with the *nbns* keyword. You can put this line in the [default] section or under a particular server. For most environments, use the IP of the domain controller.

```
nbns=203.0.113.8
```

While modern Windows networks use DNS to find hosts, WINS might still be useful on older networks. Many huge corporations still have WINS kicking around in back offices.

password

The *password* keyword lets you set a clear-text password for a user or share. Yes, clear-text. If you must store passwords in *`nsmb.conf`*, be absolutely certain that only **`root`** can read the file. On a multi-user system, storing passwords in *`$HOME/.nsmbrc`* is a terrible idea. Automatically mounting a share at boot requires using a hard-coded password, however.

FreeBSD lets you assign a scrambled password to this keyword. The scrambling is easily reversed, but protects the password against casual exposure. Scrambled passwords start with double dollar signs (`$$`). Scramble your password with `smbutil crypt`.

```
# smbutil crypt RealPassword
$$15e5a5b3b0c320f10e9f2f8db
```

Use `password` in a section with user information.

If the server's role requires that access, rather than using your own account, ask your Windows team for an account that only has access to mount the needed share.

Taken as a whole, for the user **`mwl`** on the server **`fileserver`** in the workgroup **`mwl`** you might wind up with an *`/etc/nsmb.conf`* much like this.

```
[default]
```

```
nbns=203.0.113.12
workgroup=MWL
[FILESERVER:mwl]
password=$$1744a412a293f03
```

Given this simple configuration, you can access and mount CIFS shares.

Using CIFS

Working with CIFS requires finding the host, mounting and unmounting shares, and understanding permissions and case.

CIFS Name Resolution

Before FreeBSD can mount a share, it needs to find the host the share is on. While Microsoft has used DNS for many years now, typical Windows environments often have a whole bucket of legacy protocols. Make sure FreeBSD's CIFS client can find the CIFS server with `smbutil lookup`.

```
# smbutil lookup fileserver
Got response from 203.0.113.77
IP address of fileserver: 203.0.113.77
```

A correct answer means that the simplest CIFS functionality works. Now let's access some data.

If you have trouble with name resolution, you might need to use the *address* keyword in *nsmb.conf* to manually set the IP address of any host you want to access. Address resolution should always work, but given how CIFS has changed in the last quarter century and how some organizations have structured their environments, it's nice to have the option.

Mounting CIFS

Mount a CIFS share with mount_smbfs(8). The mount must include the Windows username, as shown here.

```
# mount_smbfs //username@server/share /mountpoint
```

I want to mount the corporate share on the server fileserver, using the account **`mwl`**. This share will get mounted at */mnt*.

```
# mount_smbfs //mwl@fileserver/corporate /mnt
```

You'll get prompted for a password. Once you give a vaild password, FreeBSD mounts the share and you can easily access documents on the Windows server.

View all of the servers you're logged into with `smbutil lc`.

```
# smbutil lc
SMB connections:
VC: \FILESERVER\MWL
(root:wheel) 711
  Share: CORPORATE(root:wheel) 711
```

This FreeBSD machine is connected to one user account on one host.

Disconnect the CIFS filesystem with umount(8), exactly like any other filesystem.

CIFS File Ownership

Windows and Unix-like systems have utterly different permissions schemes. FreeBSD usernames map to Windows usernames only accidentally. Combined, these mean you need to pay careful attention to permissions when mounting CIFS shares.

A CIFS mount defaults to using the same permissions as the mount point. If I mount the share on */mnt*, the files will get the `root:wheel` ownership and mode 755 used by */mnt*. If mounted on */home/mwl/mnt*, the files will be owned by the user `mwl` and get whatever permissions the directory and umask dictate.

You can specify an owner and a group for mounted files with the `-u` and `-g` options to `mount_smbfs`. Similarly, use `-d` to change the directory mode and `-f` to change the permissions mode. Here I specifically assign my mounted files to be owned by the user and group `smb`, and change the permissions so only that user can access the files.

```
# mount_smbfs -u smb -g smb -d 700 -f 600 //mwl@fileserver/corporate /mnt
```

Despite any permissions you assign on the FreeBSD side, the Windows permissions also apply. If the Windows user you use to mount the share lacks permission to view or edit files, the FreeBSD host cannot view or edit those files.

CIFS and Case

Microsoft filesystems are case-insensitive. FreeBSD filesystems are case-sensitive. FreeBSD tends to leave the case on files as it finds it, but that might not be desirable. Automated processes might find it easier to identify files with consistent case. The `-c` flag tells mount_smbfs(8) to give everything a consistent case. Using `-c l` changes everything to lowercase, while `-c u` changes everything to uppercase.

CIFS and NFS let you access filesystems over the network. Now let’s make the network part of the filesystem itself, with iSCSI and HAST.

Chapter 8: iSCSI

Sometimes you want to give a host access to a disk over the network. This can let you exceed the host’s physical limitations. No room in the chassis for another disk? Add disk space over the network! Perhaps several hosts need access to the same hard drive, as is common for virtualization systems. Or you might have hardware that doesn’t have a local disk at all.

That’s where iSCSI comes in. Most storage devices use the Small Computer System Interface (SCSI) command set or a subset of those commands. While SCSI hard drives disappeared from new systems years ago, SCSI commands linger on in modern SATA and SAS drives. iSCSI, or Internet SCSI, tunnels SCSI commands inside TCP. Where protocols like NFS (Chapter 6) and CIFS (Chapter 7) require teaching the client about a whole new filesystem, iSCSI only requires wrapping standard SCSI commands inside a network connection. The client sees the iSCSI drive as a normal local drive and can use it accordingly. What could possibly go wrong? Well, the network is not a SAS cable—it’s not even a SCSI cable, so the answer is “quite a bit, really.”

FreeBSD 10 and newer has an integrated high-performance iSCSI stack, interoperable with all major vendors.

Storage and Network Speed

Networks are much slower than storage systems. Most modern computers use gigabit network interfaces, while their SATA or SAS systems run at six times that. Combined with the facts that not all gigabit hardware actually runs at gigabit speeds, and most hosts have four or more SATA ports, common disk activity can completely saturate a network interface.

If you start to have performance problems, separate your storage network from the rest of your network traffic. I only deploy networked storage on a private network. If I have only two machines sharing storage over the network, in a failover configuration, I'll connect a cable directly between them and put the storage traffic there. Using iSCSI or other raw disk protocols over the network demands careful attention to network performance.

iSCSI Essentials

iSCSI is something like a client-server protocol. The client is called an *iSCSI initiator*. iSCSI uses a special name for the client because it takes its terminology from SCSI. The client initiates, or causes, all activity—that's why it's called the initiator. The *target* receives the requests and acts on them. It's much like any other storage device: the operating system makes requests and the drive services them. FreeBSD initiators use the iscsid(8) daemon to connect to targets.

FreeBSD provides server-side iSCSI services with the CAM Target Layer daemon ctld(8). An iSCSI server exports storage devices to initiators. These storage devices might be actual hardware, ZFS volumes, or files on disk. Each storage device offered by the server is called an iSCSI *target*. One iSCSI server can offer as many targets as the underlying operating system or hardware can support.

An iSCSI *portal* is the IP address and TCP port where the iSCSI server accepts requests for iSCSI targets. (iSCSI defaults to TCP port 3260.) Portals are collected into *portal groups*, which define which IP addresses the portal may listen on. iSCSI is not a protocol for use on the public Internet. Always protect your storage with a firewall or packet filter.

iSCSI supports authentication via CHAP. You can disable authentication, require the initiator to authenticate, or require both the initiator and the target to authenticate. You can also disallow authentication, but that prevents anyone from talking to your targets.

Finally, iSCSI supports *discovery*, where an initiator asks an iSCSI portal group to list all accessible storage devices.

We'll start into all this with iSCSI device naming.

iSCSI Device Naming

The first thing most people choke on when they start with iSCSI is the naming scheme used for both clients and initiators. Looking at a string like iqn.1996-03.com.sun:01:3a7b… followed by a whole string of hexadecimal numbers and some more periods and then more random gobbledygook can be intimidating, annoying, and headache-inducing. iSCSI device naming doesn't have to be that difficult, however.

All iSCSI names start with the string *iqn*, for iSCSI Qualified Name, followed by a period.

Then there's the year and month the organization was founded, separated by a dash.

There's another period, and the organization's domain name in reverse order. A domain like michaelwlucas.com becomes com.michaelwlucas.

Last you have a colon and a name for a particular device on the iSCSI target.

Let's put this all together. I'm using the domain name mwl.io, registered in November 2013. My iSCSI devices will all have names starting with iqn.2013-11.io.mwl. Despite what many storage vendors claim, iSCSI devices don't need a big long unique identifier. Long identifiers make sense on large storage farms, but a simple name works better when learning. I'll call my first initiator *iqn.2013-11.io.mwl:host1*, and my first target *iqn.2013-11.io.mwl:target1*. I could use more meaningful names, like the initiator's hostname or the target drive's LUN, but these suffice to get started.

The important thing to remember about all these names is that they're private. Nobody else outside your organization will ever see them. If you have a horribly long domain name like michaelwlucas.com, don't use it in the iSCSI name. If I choose to use a short domain name that I don't own, or if I arbitrarily declare that my organization started on 2000-01, nobody will care.

We'll begin by creating a target, then an initiator. The initial configuration will get iSCSI working, but not very securely. We'll add security and discovery as we go on.

Target Setup

Start by creating a ctld(8) configuration file, */etc/ctl.conf*. This file should not be world-readable.

```
# cd /etc
# touch ctl.conf
# chmod 600 ctl.conf
```

Then enable `ctld` at boot.

```
# sysrc ctld_enable=YES
```

Now create your iSCSI configuration in */etc/ctl.conf*. You must reload or restart `ctld` for any changes to take effect. Like many other services, running `service ctld reload` checks the configuration file for errors. It will not restart the service with an invalid *ctl.conf*.

A working configuration needs at least two parts: a portal group and a target. All targets are members of a portal group, so create the portal group first.

```
portal-group group0 {
  discovery-auth-group no-authentication
  listen 0.0.0.0
  listen [::]
}
```

We start with the *portal-group* declaration, followed by the name of this portal group, *group0*. The braces that follow contain the configuration for the portal group.

iSCSI discovery happens at the portal group level. Discovery normally requires authentication, as we'll discuss later this chapter. In this example, we've deliberately disabled authentication for discovery.

We then list the IPv4 and IPv6 addresses this portal group listens on. The `0.0.0.0` means to listen on all of the host's IPv4 addresses. For IPv6, `[::]` means "all available IPv6 addresses."

Now that you have a portal group to put the target in, create the target.

```
target iqn.2013-11.io.mwl:target0 {
  auth-group no-authentication
  portal-group group0
  lun 0 {
    path /dev/zvol/data/disk1
    size 1T
  }
}
```

A target declaration begins with the *target* keyword and the target name. I've used the target name I created in "iSCSI Device Naming" earlier this chapter, iqn.2013-

11.io.mwl:target0. The braces that follow contain the target configuration.

Every iSCSI target needs an authentication option and a portal group. In this target, the *auth-group* keyword is set to `no-authentication`, meaning that this target does not require authentication. Anyone who can connect to the iSCSI network socket can attach to the storage devices in the target. The *portal-group* keyword gives the name of the portal group defined earlier, *group0*.

This target has one Logical Unit Number (LUN). Every target needs a LUN 0. FreeBSD won't mind if you skip LUN 0, but some initiators care a great deal. You can have any number of LUNs in the target, but any host that authenticates to the target can access every LUN in the target.

Each LUN has two statements: the path to the storage device to be exported over iSCSI and the size of that device. The LUN 0 device is using a zvol as a backing store.

This gives you a minimal iSCSI configuration that exports one device to the entire world, with no iSCSI-level security. Let's expand this into something you might want to actually use.

iSCSI Target Authentication

You probably don't want any random yahoo to connect to your iSCSI devices, even if you're inside your organization's security perimeter. Even if you trust everyone you work with, we've all had those moments when we hit ENTER and abruptly realize we've just ruined everything. Requiring authentication to access remote disks helps prevent daft mistakes.

iSCSI supports the Challenge Handshake Authentication Protocol, or CHAP, for username and password authentication. The iSCSI protocol uses two types of CHAP: plain old CHAP and mutual CHAP.

With standard CHAP, a username and password are required to access an iSCSI client. Clients must provide valid credentials to access an iSCSI target.

Mutual CHAP assigns usernames and passwords to both the iSCSI target and the initiator. With mutual CHAP, the initiator presents its own username and password and demands the target's username and password in return. Mutual CHAP might seem like overkill in your environment, but authenticating both initiators and targets reduces accidents and can help detect both misconfigurations and security incidents. If your usual iSCSI target suddenly forgets its own username and password, something is very wrong!

An iSCSI user's password is called a *secret*. Secrets are normally stored in plain text in configuration files. There are legitimate reasons for multiple people to have access to an iSCSI secret, so they aren't stored as hashes like passwords. Unlike passwords, you'll type an iSCSI secret only rarely—usually you copy and paste them. While the example secrets in this book are short and easily remembered, CHAP secrets should be at least 12 characters long and contain many sorts of different characters.

Configure authentication in *`ctl.conf`*, through authentication groups.

Authentication Groups

Multiple initiators connecting to a single target shouldn't all use the same username and secret. FreeBSD uses *authentication groups* to create collections of usernames and secrets. You assign an authentication group to a target, effectively saying "these users can connect to this target." While you can assign authentication information on a per-target basis, authentication groups let you reuse authentication across multiple portals and targets.

Authentication groups start with the string *auth-group* and a name. Each authentication group needs a unique name. Here I define an authentication group, *db*, with two valid usernames.

```
auth-group db {
  chap bugs daffy
  chap pinky brain
}
```

Yes, usernames and passwords are stored in plain text, in a text file. That's why `/etc/ctl.conf` needs to not be world-readable. Only users who should have access to the host's iSCSI secrets should be able to read the file.

Each target needs an authentication group, set by the *auth-group* keyword. Portals, similarly, require an authentication group for discovery, defined by the *discovery-auth-group* keyword.

Each authentication group can only contain one type of credentials. You cannot mix CHAP and mutual CHAP in one authentication group.

The ctld(8) daemon always creates, but does not necessarily use, the authentication group *no-authentication*. This group lets anyone connect to a target without authenticating. Similarly, the authentication group *default* prevents anyone from authenticating to the target. Configure any other authentication groups yourself.

CHAP Authentication

To define CHAP users, use the string `chap`, a username, and the user's secret. Here I define the user *pepe* and the secret *penelope*.

```
chap pepe penelope
```

CHAP user definitions must go inside an auth-group or target statement, as shown under "Authentication Groups" above.

Do not combine CHAP and mutual CHAP statements in a single auth group.

Mutual CHAP Authentication

Configuring mutual CHAP requires two usernames, each with their own secret. Use the *chap-mutual* keyword.

```
chap-mutual user secret target-user target-secret
```

Here the initiator has the username *mickey* and the password *minnie*. The target has the username *donald* and the password *goofy*.

```
chap-mutual mickey minnie donald goofy
```

Your own secrets should be 12 characters or longer and far more complex than these examples.

As with a standard CHAP statement, mutual CHAP can only be defined inside an authentication group or a target. You cannot combine `chap` and `chap-mutual` statements in a single group.

Restricting Initiators

In addition to requiring a username and password to access a target, you can also restrict which initiators can connect to which targets by using the initiator name and IP address.

Use the *initiator-name* keyword to define initiator names that can connect in an authentication group. If you have multiple permitted initiators, list each initiator name on its own line. Here I require the initiator to use the name I created earlier this chapter, as well as several similar initiator names.

```
auth-group db {
  chap bugs daffy
  initiator-name iqn.2013-11.io.mwl:host1
  initiator-name iqn.2013-11.io.mwl:host2
  initiator-name iqn.2013-11.io.mwl:host987
}
```

Most initiator software can set the initiator name, so restricting initiator names isn't a security measure. If someone has the target's CHAP username and secret, they can probably capture the initiator name as well. It's more of a way to prevent problems caused by human error.

Restricting initiators by IP address is probably more effective. Use the *initiator-portal* keyword and the permitted IP address or network block. If you have multiple permitted IP blocks, list each on its own line with its own *initiator-portal* keyword.

```
auth-group db {
  chap bugs daffy
  initiator-name iqn.2013-11.io.mwl:host1
  initiator-portal 203.0.113.0/24
  initiator-portal [2001:db8:bad:c0de::]/64
  initiator-portal 192.0.2.87
}
```

While restricting connections by IP address is probably more secure than limiting initiator names, it still doesn't replace firewalls or packet filters. You really need a network access control device between your targets and the public Internet.

Restrictions by IP or initiator name are usable only in addition to other authentication methods. You must either use some sort of CHAP or specifically allow access without authentication.

Portal Groups

A *portal group* is the glue attaching iSCSI targets to an IP address. It also lets you set the authentication needed for discovery.

Portal Address

While your iSCSI server might listen on all addresses on the system, sometimes you want specific targets available only on particular IP addresses. Use the *listen* keyword and an IP address.

```
portal-group group0 {
  discovery-auth-group db
  listen 192.0.2.1
  listen [2001:db8::ace]
}
```

This is useful for, say, migration situations. Imagine your iSCSI load has hit a point where you need to split the server into two hosts. Add a second IP address to the host. Configure half of your hosts to use the new IP as their iSCSI target. On migration day, move the new IP to the new host and plug the associated SCSI shelf into the new server.

Discovery Authentication

Use the *discovery-auth-group* keyword to attach one and only one authentication group to the portal group. When an initiator queries the portal to learn which resources are available, the portal will require a username and secret before sending any information.

Targets

Targets are where you attach on-disk storage to the iSCSI service. In addition to the back end storage, you can assign authentication credentials, aliases, and LUNs.

Each target starts with the *target* keyword and the target's iSCSI name. A working, useful target must contain an authentication configuration, a portal group, and at least one LUN. (You can write a *`ctl.conf`* entry that lacks some of these, but it won't be useful.)

Target Authentication

The simplest (and, in my opinion, best) way to configure authentication for a target is to use an authentication group, abstracting the authentication away from the target itself. The *auth-group* keyword tells the target to pull in an authentication configuration from a previously defined group.

```
target iqn.2013-11.io.mwl:target0 {
  auth-group databases
  …
}
```

You can configure authentication information directly within the target, however. Use the *auth-type* keyword to define the type of authentication needed to access this target. You can use either `chap` or `chap-mutual`, meaning that you get to set up usernames and secrets later in this target. If set to `deny`, nobody can authenticate to this target. Finally, `none` tells the target that it doesn't require authentication (but it might have other restrictions).

The `chap` and `chap-mutual` keywords work inside a target definition exactly as they do in an authentication group. You can use only one type of authentication for each target—that is, you can't set `auth-type chap` and then have a `chap-mutual` statement.

```
target iqn.2013-11.io.mwl:target0 {
  auth-type chap
  chap dbuser dbsecret
  …
}
```

A target can include only one type of username and secret authentication—either `auth-group`, `chap`, or `chap-mutual`. Multiple targets can use different authentication methods.

The `auth-type` keyword is most often optional. If you don't explicitly state an authentication type, the presence of `chap` or `chap-mutual` lets FreeBSD infer the authentication type. The main reason the `auth-type` keyword exists is so you can explicitly specify `auth-type none`.

If a target does not offer any authentication methods, all attempts to use the iSCSI device are denied. To allow access without authentication, use the built-in CHAP group `no-authentication`.

You can also use the `initiator-name` and `initiator-portal` options in a target, exactly as in an authentication group. Here I permit a specific IP with a particular initiator name to access the target without a password.

```
target iqn.2013-11.io.mwl:target0 {
  auth-type none
  initiator-name iqn.2013-11.io.mwl:host1
  initiator-portal 203.0.113.208
  …
}
```

If you configure an initiator to use authentication, but the target doesn't require authentication, the connection succeeds.

Portal Group

Every target must be attached to a portal group. Use the *portal-group* keyword and the name of a previously defined portal group.

```
target iqn.2013-11.io.mwl:target0 {
  portal-group group0

    …

}
```

Each target can belong to only one portal group.

Aliases

Add a human-readable description to a target with the *alias* keyword.

```
target iqn.2013-11.io.mwl:target0 {
  alias database-storage

    …

}
```

Aliases cannot contain whitespace.

You'll see the alias in a few iSCSI discovery tools. FreeBSD's initiator shows the alias in verbose mode.

Logical Unit Numbers and Backing Stores

A Logical Unit Number (LUN) is a storage device. It might be a ZFS volume, a chunk of disk space, or even a physical device. Each iSCSI target requires one or more LUNs. Any initiator that connects to this target accesses all the LUNs in the target. Define LUNs within `target` statements. Each LUN requires a path to the back end storage device. The size in bytes is optional.

Here I define a target with three LUNs.

```
target iqn.2013-11.io.mwl:target0 {
  alias database-storage
  auth-group db
  portal-group group0
  lun 0 {
    path /dev/zvol/db/disk1
    size 1T
  }
  lun 1 {
    path /db/disk2.img
    size 2T
  }
  lun 2 {
    path /dev/da0
    size 1T
  }
}
```

LUN 0 is a 1 TB ZFS volume, accessible through the device node */dev/zvol/db/disk1*. For optimal performance, create the volume with the **volmode** property set to *dev*. (Using a **volmode** of *dev* lets ZFS bypass GEOM and manage I/O sizes more flexibly. It also enables DPO and FUA cache control, letting ZFS tell the drive to not put select data in the write cache.)

LUN 1 is backed by a file on the filesystem, */db/disk2.img*. This might be on a UFS partition or a file on a ZFS dataset. It could be a regular file or a sparse file.

LUN 2 is a physical disk with the device node */dev/da0*.

Number other LUNs with any positive integers.[1] I didn't need to use LUNs 2 and 3 here, but I like nice short numbers. LUN numbers do not need to be sequential.

If you have many iSCSI devices served by files or zvols, seriously consider naming the backing store for each LUN after the LUN. Use zvols or files with commas in the name, such as *target1,l,0* for target 1, LUN 0. (Note that the middle character here is the letter l, not the number 1.)

Any initiator that connects to this target gets access to all three LUNs.

LUN Size

Each LUN can have a `size` statement, giving the size `ctld` reports to the initiator, in bytes. You can use abbreviations like G for gigabytes and T for terabytes. If you don't specify a size, FreeBSD will probe the device to figure out the size. I like the `size` statement because I'm forgetful.

The size given should be the same as the actual size of the storage device, but that doesn't always happen. In the example above, I claim that the disk behind LUN 3 is one terabyte. It's a physical 1 TB disk, which means that the manufacturer measured the size in base 10. FreeBSD measures disk sizes in base 2. Running `geom disk list da0` gives the actual size of this specific disk as 1000204886016 bytes, or 932 GB. I'd be better off listing the actual value here rather than rounding it off to the nearest terabyte. Remember, not all "1 TB" disks are the same size.

In addition to size and the backing store, you can set a variety of other LUN characteristics.

Blocksize

While physical hard drives have had 512-byte sectors for decades, newer "advanced format" drives have 4096-byte physical sectors. For physical disks and ZFS volumes, ctld(8) passes the physical sector size to the initiator. On file-backed iSCSI volumes, set the sector size with the *blocksize* keyword.

```
lun 2 {
  path /jails/disk2.img
  blocksize 4K
  size 2T
}
```

If you're using file-backed iSCSI storage rather than a zvol or a physical disk, setting the block size can help the iSCSI initiator size write requests correctly for the underlying storage.

Drive Information

SCSI commands expose a physical hard drive's model name, serial numbers, and other vendor-specific data. You can set this same information within an iSCSI LUN definition.

The ctld(8) daemon provides the *device-id* and *serial* keywords. Use *device-id* to provide something like a model name. The *serial* keyword is for anything like a serial number. Neither entry can have spaces.

Here I use `device-id` to let people who administer initiator hosts view what sort of storage the target uses. I've assigned a meaningless serial number, but you might use the serial number to pass other data to the client.

```
lun 0 {
  path /dev/zvol/jails/disk1
  size 500G
  device-id zvol
  serial 0123456789abcdef
}
```

On the client, the device ID is visible in the `lunname` and `lunid` values of `geom disk list`. The serial number is visible in the disk's `ident` string. Here I log into an initiator host and view this device as it appears on the local system.

```
# geom disk list da5
…
  descr: FREEBSD CTLDISK
  lunname: FREEBSD zvol
  lunid: FREEBSD zvol
  ident: 0123456789abcdef
…
```

The leading FREEBSD and the space always appear in the LUN name and LUN ID.

ctld(8) Performance

While `ctld` is designed to perform well, you can tweak a few things in its behavior.

Each incoming TCP connection makes `ctld` spawn a child process. By default, `ctld` limits the number of child processes to 30. If you're running a busy iSCSI server with many initiators, you might need to increase this with the `maxproc` option.

```
maxproc 30
```

Setting `maxproc` to `0` allows unlimited child processes.

Initiators generate a nearly constant flow of traffic to their targets. Even if the iSCSI disk is idle, the target will receive traffic every few seconds. Initiators that stop sending traffic for more than a few seconds are probably gone. The `timeout` value lets you tell `ctld` how long to maintain an idle connection before dropping it. The default is 60, a reasonable average. Your equipment or environment might need a different timeout.

```
timeout 90
```

All performance and debugging options go at the top level of `ctl.conf`, not inside any braces.

Debugging ctld(8)

While the SCSI command set gets complicated, iSCSI itself it pretty straightforward. Even so, we sysadmins can discover many ways to mess up our iSCSI configurations. The ctld(8) debugging features usually give good insight into problems.

Use the `debug` statement to enable debugging. The default level, 0, provides only bare operational messages. A debugging level of 3 gives useful information when an iSCSI connection goes awry, so that's what I normally use.

```
debug 3
```

Information from `ctld` appears in */var/log/messages*. You cannot change the log facility and level (`daemon.debug`), so if you want to log them separately you'll need to split them out in syslogd(8).

Whenever an initiator fails to authenticate, or when the storage for a LUN is absent or faulty, ctld(8) logs the error. Most `ctld` error messages are actually fairly easy to understand. Authentication errors describe which authentication constraint was violated. LUN errors are more generic, but normally mean that you mistyped the path to the backing storage in *ctl.conf*.

I use a debug level of 3 during normal production use, to easily identify and resolve issues. Log space is cheap.

Configuring Initiators

The iSCSI initiator programs share a configuration file, */etc/iscsi.conf*. This configuration file was also used by the iSCSI initiator in older versions of FreeBSD, so its documentation includes references to the obsolete iscontrol(8). The configuration format deliberately did not change between the two sets of software, however.

The simplest uses of the iSCSI initiator do not require a configuration file. You can discover targets on a host, connect to and disconnect from them, and perform CHAP authentication without a configuration file. A configuration file permits use of more complex options like mutual CHAP, and allows more selective iSCSI management. In production, always use a configuration file.

Each iSCSI target gets a bracketed section in *iscsi.conf*.

```
target-nickname {
  TargetAddress = storm.mwl.io
  TargetName = iqn.2013-11.io.mwl:target0
  InitiatorName = iqn.2013-11.io.mwl:host1
  …
}
```

A target's entry starts with a nickname. This sample target is unimaginatively nicknamed *target-nickname*.

Inside the braces we have the various iSCSI settings and their values, as defined in iscsi.conf(5). We'll add more settings in the relevant topics, but the three here are very commonly used.

TargetAddress can be a hostname or IP address. It's the IP address of the iSCSI portal providing the target. My iSCSI server is called **storm.mwl.io**.

TargetName is the formal name of the iSCSI target you want to connect to. This connection is for the target name we created earlier this chapter.

InitiatorName is a name you've chosen to assign this initiator. If you don't assign a name for this initiator, FreeBSD creates one for you. Assigning an initiator name is a convenience for you.

A target definition's closing bracket must appear on a line by itself. Don't tack it onto the end of the last variable setting.

When using iSCSI disks, remember that the device node is even more dynamic than on normal hardware. The disk that is */dev/da5* today might be a completely different disk node tomorrow. Always manage filesystems on iSCSI disks with labels rather than by device node.

Enabling the iSCSI Initiator

FreeBSD provides iSCSI initiator services through iscsid(8) and iscsictl(8). The `iscsid` daemon handles connections, while `iscsictl` lets you issue iSCSI commands.

iscsid(8)

The iSCSI daemon iscsid(8) manages an initiator's iSCSI logins and discovery. Once `iscsid` establishes a connection, it hands that connection to the FreeBSD kernel. To enable `iscsid`, set `iscsid_enable` to YES in */etc/rc.conf*.

```
# sysrc iscsid_enable=YES
```

If you have trouble with iSCSI logins, you might try `iscsid` debugging mode. The `-d` flag tells `iscsid` to run in the foreground, and print all the debugging information to the terminal. Debug-mode `iscsid` terminates after handling only one connection. This won't interrupt existing connections, as `iscsid` handles only logins and discovery. Even without `iscsid`, established iSCSI connections remain up until something interrupts them. With `iscsid` running, interrupted connections automatically resume.

While `iscsid` normally gets its debugging level from *iscsi.conf*, you can override that setting on the command line with `-l` and a log level. This lets you debug specific problems without editing the configuration file. A log level of 3 will identify most problems.

Each iSCSI login spawns a child process. These child processes handle authentication, negotiate various iSCSI parameters, hand the connection off to the kernel, and exit. Normally, the maximum number of simultaneous child processes is 30. If you must process more than 30 iSCSI logons simultaneously, use the `-m` flag to set a new maximum. Setting the maximum to 0 removes any limits on the number of child processes.

Initiator Controller: iscsictl(8)

Configure and manage your iSCSI initiator through `iscsictl`. While you can perform some configuration at the command line, complex setups require the configuration file *`/etc/iscsi.conf`*.

An iSCSI initiator has three core functions: discovering targets, connecting to targets, and disconnecting from targets. You can also list existing iSCSI sessions.

Initiator Authentication

Making iSCSI connections and performing discovery most often requires authentication. You can set authentication options on the command line or in *`iscsi.conf`*.

By default, `iscsictl` does not offer authentication. If a target doesn't need authentication, don't include any authentication information. Offering authentication information to a target that doesn't need authentication might cause the connection to fail.

To authenticate with CHAP on the command line, use `-u` to define a user and `-s` to give the secret. Here I run an iSCSI command (discovery) with the user *pinky* and the secret *brain*.

```
# iscsictl -Ad storm -u pinky -s brain
```

To set CHAP authentication in *`iscsi.conf`*, you must set the *AuthMethod*, *ChapIName*, and *ChapSecret* variables. Here I set that same username and secret for the target nicknamed *db*.

```
db {
  TargetAddress = storm.mwl.io
  InitiatorName = iqn.2013-11.io.mwl:host1
  TargetName = iqn.2013-11.io.mwl:target0
  AuthMethod = CHAP
  ChapIName = pinky
  ChapSecret = brain
  …
}
```

If a target requires authentication, set AuthMethod to CHAP—even if the target uses mutual CHAP. And pay special attention to the username variable. There's a capital *I* in the middle of ChapIName.

You cannot perform mutual CHAP on the command line. You must set the remote user and secret in *`iscsi.conf`*, using the *tgtChapName* and *tgtChapSecret* values as shown here.

```
db {
  TargetAddress = storm
  TargetName = iqn.2013-11.io.mwl:target0
  AuthMethod = CHAP
  ChapIName = mickey
  ChapSecret = minnie
  tgtChapName = donald
  tgtChapSecret = goofy
}
```

This connects the initiator to the mutual CHAP target built earlier this chapter.

Authentication is only needed for discovery and connecting. You can disconnect without authenticating.

iSCSI Discovery

The simplest way to connect an initiator to a portal is to perform an *iSCSI discovery*. This is where the initiator goes to the iSCSI portal on the target server and says, “Hey, what LUNs can you tell me about?” If the initiator successfully authenticates, the portal returns a list of targets.

The `–A` flag tells `iscsictl` to add a session. The `–d` flag triggers discovery, and takes one argument: the hostname or IP address of the iSCSI portal group. Here I perform iSCSI discovery against the host **`data1.mwl.io`**.

```
# iscsictl –A –d data1.mwl.io
```

When the initiator gets the drive information, it automatically logs into the system and attaches all of the discovered LUNs to the local system. The only way to see what your initiator discovered is to view the existing iSCSI sessions.

Viewing iSCSI Session and Connections

To see all active iSCSI sessions, either run `iscsictl` without any arguments or use the `–L` flag. Either gives the same result.

```
# iscsictl -L
Target name Target portal State
iqn.2013-11.io.mwl:target0 storm Connected: da4 da5
iqn.2013-11.io.mwl:target1 storm Connected: da3
```

This initiator has connected to two targets on this host. You’ll see the target names, the portal, and the device node the local host has assigned to the drives in this target.

If you want more detail, add the `–v` flag to `–L`.

```
# iscsictl -Lv
Session ID: 21
Initiator name: iqn.2013-11.io.mwl:host1
Initiator portal:
Initiator alias:
```

```
Target name: iqn.2013-11.io.mwl:target0
Target portal: storm
Target alias: database-storage
User: pinky
Secret: brain
Mutual user:
Mutual secret:
Session type: Normal
Session state: Connected
…
```

The first item, the session ID, is a unique number assigned to this particular connection. If you disconnect from a target and reconnect to it later, you'll get the next free session ID number. ID numbers start again from 1 after a reboot.

The verbose listing includes the complete target name and the authentication information, including any secrets. You can use this information to build an *iscsi.conf* entry for this target. You'll also see the target alias assigned by the storage administrator.

To view details on a particular disk, fall back on disk management tools like `geom disk list` and gpart(8).

Connecting to Targets

One problem with discovery is that the initiator automatically connects to all available targets on the portal. Just because the host *can* connect doesn't mean it *should*, though. Once you know what targets you can access, `iscsictl` lets you attach them more selectively.

Adding an iSCSI session again requires the `-A` flag.

If the host doesn't have an entry for the desired target in */etc/iscsi.conf*, use the `-p` flag to specify the portal hostname or IP, as well as the `-t` flag to give the name of the desired target. Here I attach to the target *iqn.2013-11.io.mwl:target1* on the host **storm**.

```
# iscsictl -A -p storm -t iqn.2013-11.io.mwl:target1
```

If *iscsi.conf* has an entry for my target, I can connect by the assigned nickname. The nickname is the first item in the target description, immediately before the opening brace. Use `-n` to specify the nickname. The sample *iscsi.conf* entry above uses the nickname *db*, so let's connect to it.

```
# iscsictl –A –n db
```

The host connects to that target, and only that target.

To connect to all iSCSI disks configured in *iscsi.conf*, use the `-a` flag.

```
# iscsictl –A –a
```

Whenever an initiator connects to a target, it gets all of the disks in the target. There's no way to connect to only one disk of several in a target.

Disconnecting From Targets

Use the `-R` flag to remove connected targets. To get rid of all iSCSI connections, add the `-a` flag.

```
# iscsictl –R –a
```

To ditch one specific connection among several, specify the portal with `–p` and the target with `–t`. This works exactly like connecting to that same target, except it uses `–R` instead of `-A`.

```
# iscsictl -R -p storm -t iqn.2013-11.io.mwl:target1
```

Finally, for targets configured in *iscsi.conf*, specify a nickname to disconnect with `–n`.

```
# iscsictl –R –n db
```

You can now create and use iSCSI disks.

Initiator Tuning

The FreeBSD iSCSI initiator lets you adjust timing, failure, performance, and logging behavior through boot-time tunables. Although you can read these values through sysctl(8), you can only change them in */boot/loader.conf*.

iSCSI log messages normally appear in */var/log/messages*. The sysctl `kern.iscsi.debug` enables these messages. It defaults to 1, for *on*. To disable most iSCSI logging, set `kern.iscsi.debug` to `0`. You'll still get messages when you attach and detach drives, but your logs will be much smaller. Disabling logging also disables any hope of diagnosing problems, though.

The tunable `kern.iscsi.maxtags` puts a maximum limit on the number of I/O requests the initiator will have outstanding at any given time. The default, 255, is a reasonable setting. If you consider increasing this, spend time improving the performance of your storage network and/or your iSCSI target instead.

iSCSI Timing

You can adjust the time the iSCSI client spends waiting for various parts of the iSCSI process.

The tunable `kern.iscsi.iscsid` sets the number of seconds iscsid(8) waits for the target to respond to a connection request and establish a session. The default is 60 seconds. If your iSCSI server can't establish a connection within 60 seconds, something's wrong with your network or the target server.

Similarly, `kern.iscsi.login_timeout` sets the number of seconds iscsid(8) waits for the iSCSI target to process the login. The default is also 60 seconds. If the iSCSI target can't process a simple login within a minute, you probably don't want to try sending I/O through it either.

Finally, the `kern.iscsi.ping_timeout` tunable gives the number of seconds the initiator will wait for the target to respond to a NOP-Out request, or an "iSCSI ping." The default is five seconds. If the target can't service a "hello, are you there?" request within five seconds, you probably don't want to send any I/O to it anyway. Some iSCSI stacks (notably, particular Linux versions) do not support NOP-Out requests, however, and you might need to disable them for those stacks.

iSCSI Failure Behavior

It’s rare that all your systems will fail simultaneously. You can tell the initiator how to react when the target disappears.

By default, the FreeBSD initiator retains the device nodes for the missing drives. The iscsid(8) daemon keeps trying to restore the connection to the targets. In some settings it makes more sense to remove the device nodes, however, such as the failover discussed in Chapter 10. To make the initiator remove the device nodes when the target disappears, set the tunable `kern.iscsi.fail_on_disconnection` to `1`.

Now that you can compel iSCSI to obey your will, let’s move on to HAST.

---

[1] No, you can’t have LUN π.

Chapter 9: GEOM_GATE and HAST

In addition to exporting filesystems across the network, FreeBSD lets you export device nodes. Device nodes are very operating system specific—there's rarely a reason to, say, address a Linux device node directly on a FreeBSD host or vice-versa. But perhaps you need to access the CD burner in machine A from machine B, or you need to get a filesystem image on a host that doesn't have enough space to store that image, or you just need some temporary storage space. FreeBSD includes the geom_gate kernel module, or *ggate*, that can transport storage device nodes (disks and partitions, memory devices, and optical drives) across the network.

Exporting device nodes across the network makes all kinds of things possible. The most obvious is to mirror a disk partition between two hosts, so that the hosts constantly synchronize data on those partitions. FreeBSD's HAST, or *Highly Available Storage*, lets you synchronize disks across the network.

These disk device exports are neither encrypted nor authenticated. Anyone who can intercept the traffic can view the transactions. Very few network people can identify disk device transactions, so you get a small amount of security through obscurity, but never rely on an attacker's ignorance. If you don't trust the network layer, tunnel this traffic inside a VPN or over SSH. These exports use TCP port 3080 by default. If you deploy geom_gate or HAST on a public network, protect TCP/3080 with a firewall or packet filter. Better still, directly connect the mirrored hosts with a direct (crossover-style) cable or private VLAN.

geom_gate Drawbacks

The geom_gate protocol manages to combine the disadvantages of user space filesystems (Chapter 12) with the disadvantages of networked disks as used by iSCSI (Chapter 8).

Like FUSE, both geom_gate and HAST pass storage through userland programs. Userland programs occasionally crash and die. While these programs are pretty thoroughly tested and widely trusted, deploying userland programs for storage means that you very much need proactive monitoring and automated process recovery when things go wrong.

Similarly, accessing storage devices across the network increases network load. Occasional uses of geom_gate to access device nodes at a low priority shouldn't be an issue. If you're mirroring block devices across the network with HAST, or expect good performance from geom_gate, separate your storage traffic from your regular network traffic—not with a VLAN on the main network interface, but with a separate network interface and a separate switch. If you have two servers using HAST to back each other up, connect them with a private direct cable.

Many times, the filesystem is the wrong place to have redundancy. Clustering databases is best done in the database server, not the filesystem. If you truly need redundancy at the filesystem layer, though, FreeBSD supports you.

Exporting Storage Devices: geom_gate

The geom_gate GEOM class can export only optical drives, memory disks (disk images), disks, and partitions. You can't export tape drives, terminals, */dev/random*, or other such devices. If you can't mount it, you can't export it. You cannot export mounted devices.

geom_gate Server Setup

Exporting device nodes requires setting up an exports file and starting the ggated(8) daemon.

The file */etc/gg.exports* lists exported devices, each on its own line. Each entry has three parts.

```
host permissions device
```

The host can either be an IP address like 192.0.2.8, a network like (203.0.113.0/24), or a hostname (like www.mwl.io).

Exported devices can have one of three permissions: read-write (`rw`), read-only (`ro`), or write-only (`wo`).

The last entry is the exact device node you want to export.

Here, I export the disk device */dev/da0* read-write to the hosts on my local network.

```
203.0.113.0/24 RW /dev/da0
```

Once you have an exports file, start ggated(8). There's no startup script for this daemon, but it doesn't need any command-line arguments.

```
# ggated
```

Restart `ggated` any time you change */etc/gg.exports*.

You can now set up a client.

geom_gate Client Setup

The geom_gate client does everything from the ggatec(8) program. There is no configuration file. To attach to an exported device run `ggatec create` with two arguments: the hostname of the `ggated` server, and the device node.

```
# ggatec create server device
```

Our sample `ggated` server exported the device node */dev/da0*. The server's hostname is **`storm`**. Here I connect to that device node.

```
# ggatec create storm /dev/da0
ggate0
```

The ggatec(8) command returns the local device node (*/dev/ggate0*) that it has mapped to the remote device node */dev/da0*. You can use it like any other disk device.

```
# gpart show /dev/ggate0
=>        40  1953525088  ggate0  GPT  (932G)
          40  1953525088          - free -  (932G)
```

Create your partitions and filesystems exactly like you would a local disk.

```
# gpart add -t freebsd-ufs ggate0
# newfs /dev/ggate0p1
```

This works fine in the short term, but more complex setups might require more flexibility.

To disconnect this device, use the `ggatec destroy` command, the `-u` option, and the number of the geom_gate device. We created */dev/ggate0*, or device number 0. Let's detach it.

```
# ggatec destroy -u 0
```

To see all of the devices attached to the system, run `ggatec list`. Add the `-v` flag to see details of the assorted devices.

ggatec(8) Options

With ggatec(8) you can control the permissions on attached devices, the device number, and more. While the manual lists every option, here are the most commonly used.

Just because the server offers full access to a device node doesn't mean you necessarily want that access. Use the `-o` option to set the permissions on attached devices. Use `ro` for read-only, `wo` for write-only, and `rw` for read-write. Here I attach an exported disk read-only.

```
# ggatec create -o ro storm /dev/da0
```

If you're using multiple `geom_gate` devices on a long-term basis, you might want to assign specific device numbers to each of them. While you should always manage disks with labels, not all storage devices can have labels. You can label optical disks, but not the drives they're in. It's nice when, say, */dev/ggate0* is always the CD drive and */dev/ggate1* is always the DVD drive. Assign a specific device number with `-u`.

```
# ggatec create -u 5 storm /dev/da0
```

I normally use `-u` in startup scripts, where I plan to use a `geom_gate` device in the long term.

geom_gate Failures

Adding network to storage adds a delightful layer of uncertainty to your data. You can feel confident that a server's SATA cable won't decide to stop passing bits. The network is less certain. Using `geom_gate` adds userland programs into the mix, and userland processes have been known to choke on their own bile.

If a `geom_gate` component dies, storage requests back up on the client. Recover the broken connection by using the `ggatec rescue` command, the `-u` option, and the device number. Here I kickstart a new `ggatec` process for */dev/ggate3*.

```
# ggatec rescue -u 3 storm /dev/da0
```

When you rescue the process, any buffered storage requests get sent to the server.

If you specifically want long-term mirrored storage, check out Highly Available Storage.

Highly Available Storage

The ability to access disks across the network immediately leads to the idea of mirroring disks across the network. You could build your own networked mirrored disks with raw iSCSI or geom_gate, but FreeBSD includes the Highly Available Storage (or HAST) system engineered exactly for this.

HAST works as a GEOM layer. It sits on top of a storage provider such as a disk, ZFS volume, or image file. Each HAST instance is paired with a HAST provider on another system. The HAST support daemon, hastd(8), keeps the two HAST devices synchronized. Any change to the provider gets mirrored on the partner provider.

The sysadmin assigns one of the HAST servers the primary role and the other the secondary role. HAST offers a device node in */dev/hast* for each pair of mirrored storage devices. Use that device node as you would any other node—partition it, put a filesystem on it, use it as part of a ZFS pool, whatever.

When the primary host fails, the secondary host can make itself primary, mount the filesystem, and resume providing service. This failover is not automatic, but is pretty easy to automate through the Common Address Redundancy Protocol (CARP) and devd(8).

Before configuring HAST, consider carefully if you need disk-level redundancy. Does your app truly require synchronized filesystems on two machines? Or is HAST just easier to configure than real-time MySQL replication? Mirroring storage across the network introduces many possible points of failure. While your database software's replication methods might drive you to the edge of madness, recovering a wonky database is much easier than recovering a scrambled filesystem.

HAST is best suited to an environment where you have two physical servers and want to mirror a chunk of storage between them. If you have a larger infrastructure than two hosts, consider using two iSCSI drives on two servers as the data store and having multiple hosts configure them as ZFS or GEOM mirrors. If you need interoperability with other operating systems, use iSCSI instead.

Configuring HAST requires assigning devices to participate in the mirror, configuring and initializing the mirror, creating a filesystem, and configuring failover.

Provisioning HAST

Assigning storage for HAST should be easy, right? Assign a disk in each machine to be the storage provider and get on with your day. It can be that easy—if you want to have a really bad day.

Always consider what happens in a catastrophic failure. Suppose one of your HAST nodes catches fire and needs complete replacement. The replacement HAST secondary node must initialize its copy of the HAST data. HAST, unlike ZFS, does not understand the filesystem running on top of it. If you have a measly little 4 TB drive backing your highly available storage, the backup must transfer all four terabytes over the network. How fast is your network interface, 1 Gb/s? Your 4 TB drive holds roughly 32,000 gigabits. Assuming that your network interface can actually send a full gigabit per second (unlikely, but let's be generous), re-initializing that mirror will take almost nine hours.

Make your HAST providers no larger than necessary to fulfill their purpose. Perhaps you truly do need that 4 TB drive.

In most cases, I recommend using ZFS volumes as the back end for HAST storage. You can easily clone and snapshot zvols when (*not* if) something goes wrong. If you're using UFS, use a disk image file instead. For these examples, both of my HAST hosts are using a 1 GB ZFS volume, *zroot/hast1*. This application stores only the very latest or most vital data on this volume. This reduces recovering lost HAST nodes to seconds.

One goal in designing your HAST configuration should be to minimize the amount of metadata you need to synchronize. You could use four zvols as the backing store for HAST and then build a striped mirror on top of them, but then you must pass a bunch of ZFS metadata back and forth between the hosts. Networked disk is already slow—why make it slower? It's more efficient and faster to put a single redundant device beneath your HAST, and handle any filesystem redundancy locally. If you need multiple HAST devices as separate entities, though, go ahead and use them. Just don't do your striping or mirroring on the client side.

You can use HAST on larger filesystems, but this requires that you pay careful attention to filesystem integrity and system administration practice. A multi-terabyte HAST-backed filesystem can go right on a disk partition, but demands a configuration with the maximum number of safety checks possible. If restoring your redundant storage takes nine hours, you'll want to perform that restoration as rarely as possible.

Configuring HAST

Configure HAST with hastctl(8) and in */etc/hast.conf*. The configuration file should be identical on both hosts. I strongly recommend maintaining the file in a config management tool such as Ansible or Puppet to guarantee consistency.

HAST calls each mirror of block devices a *resource*. A minimal *hast.conf* describes a single resource, without any special options. Each host in the mirror has a subsection within the resource description. Here's a *hast.conf* for bare minimum replication.

```
resource hast1 {
  on www1 {
    local /dev/zvol/zroot/hast1
    remote 192.0.2.2
  }
  on www2 {
    local /dev/zvol/zroot/hast1
    remote 192.0.2.1
  }
}
```

This creates a single HAST device, called *mirror1*. Two hosts support this, **www1** and **www2**. You must use the actual hostname for each host. Each host has two mandatory settings, where to find the local and remote parts of the mirror. For this mirror I created the ZFS volume *hast1* on each host. I also give the IP address where the remote mirror is found. (These two hosts happen to be connected by a private cable.)

Some HAST configuration happens only within a resource entry. Other configuration can be outside any resource, and applies to all resources. The integrity options discussed later, for example, can be set on a global level so they apply to all resources. Options in a resource entry override global settings.

Use hastctl(8) to initialize the HAST resource on both nodes. Then enable and start hastd(8) on both hosts.

```
# hastctl create hast1
# sysrc hastd_enable=YES
hastd_enable: NO -> YES
# service hastd start
Starting hastd.
```

Use hastctl(8) to tell the primary node that it's primary for the resource hast1.

```
# hastctl role primary hast1
```

On the secondary node, use `hastctl` to set it as backup.

```
# hastctl role secondary hast1
```

The first time you create the device and assign roles, HAST quickly initializes your storage device.

HAST Status

To check the status of a HAST device, use `hastctl status`. If you add the name of a HAST resource, you'll see only that resource. Here I look at the sample HAST resource I just created.

```
# hastctl status hast1
Name   Status   Role     Components
hast1  complete primary  /dev/zvol/zroot/hast1  192.0.2.2
```

This shows a single resource, *hast1*. The status, *complete*, means that the two hosts are synchronized. We then have this host's role in the HAST mirror, *primary*, and the storage nodes that are part of this resource.

To see more complete statistics for a host's HAST resources, use `hastctl list` and (if desired) the resource name. This gives everything from the number of dirty blocks to the number of local write errors.

HAST Filesystems

Each HAST resource has a device node in */dev/hast*, named after the resource. Device nodes for a particular resource appear only on the primary host for that resource. Our sample resource, hast1, is available as */dev/hast/hast1* on the primary node.

Here I create a ZFS pool on the HAST resource

```
# zpool create hast1 /dev/hast/hast1
```

You can use newfs(8) if UFS is preferable. I strongly recommend using soft updates journaling (`newfs -j`) on UFS.

For UFS or ZFS alike, setting `noatime` helps reduce the amount of HAST traffic and makes it easier for your hosts to synchronize the HAST device.

When your primary and secondary hosts switch roles, the device node for the HAST resource appears on the secondary node.

HAST Failover

Switching a HAST host from the secondary role to primary requires demoting the primary node and promoting the secondary node.

The unceremonious way to demote the primary node is to yank the power. For a more decorous migration, though, start by deactivating everything using the HAST resource. Shut down programs writing or reading to the resource, and unmount the filesystem or export the zpool.

```
# zpool export –f hast1
```

You can then use `hastctl` on the primary node to tell it it’s now the secondary node.

```
# hastctl role secondary hast1
```

The host is now standing by waiting for *someone* to claim the primary for this resource. Both nodes will have a status somewhat like this.

```
# hastctl status
Name   Status  Role       Components
hast1  -       secondary  /dev/zvol/zroot/hast1  192.0.2.2
```

The status on both hosts is a dash, indicating that this resource is idle.

Over on the backup node, claim the primary role and import the pool.

```
# hastctl role primary hast1
# zpool import hast1
```

If you’re using UFS, run `fsck` before mounting the filesystem. While the filesystem should be clean after you manually switch roles, a `fsck` before mounting the filesystem is best practice. On small filesystems `fsck` doesn’t take long, and on large filesystems you really need to use soft updates journaling.

Synchronization and Integrity Options

While HAST's default behavior is acceptable for most uses, you can fine-tune synchronization between hosts to best fit your environment and requirements. You can adjust the replication mode, the use of checksums, compression, and more.

These options are set either per-resource or globally. Setting an option globally, outside of any resource, means that it applies to every HAST resource in the configuration file. Here I set the `replication` option globally.

```
replication fullsync

resource hast1 {
  on www1 {
```

…

Some resources might have different performance or integrity requirements than others. You can set these options on a per-resource basis. Here I set the `replication` option for the resource hast1.

```
resource hast1 {
  replication fullsync
  on www1 {
…
```

Set options wherever needed.

Replication Mode

The HAST replication mode controls when the operating system acknowledges writing the data to disk. Just as filesystems can work in synchronous mode, asynchronous mode, or something in between, HAST lets you decide when the kernel should tell the application that a write is complete. Replication mode heavily impacts the apparent speed of the storage system. Many programs that write to disk will not proceed to the next step until the storage system acknowledges receipt of the data. Experienced sysadmins call this "blocking on disk."

The default mode, *memsync*, acknowledges the write when the data is on the primary host's disk and in the secondary host's memory. The data doesn't need to be on the secondary host's disk, only in memory. The secondary host will store the data right away as part of normal operation, but the program can proceed without waiting for the disk.

The safest way to use HAST is *fullsync* mode. HAST only acknowledges receipt of a write when it's safely on disk on both the primary and secondary nodes. This guarantees the data makes it to disk. The writes must leave the primary node, traverse the network, pass through the secondary node, and get on the secondary node's disk before the application can move on. If your data is critical, you probably want fast disk hardware and fullsync mode.

Memsync mode is faster than fullsync, but increases risk. If the secondary node fails at exactly the wrong moment, that data won't get to disk. If the primary node fails while the secondary node is rebooting, that data might be entirely lost on both nodes. As HAST exists below the filesystem, missing data might corrupt the filesystem. When I truly need redundant block device storage on multiple machines, I use fullsync mode. If my application can't take the small performance hit, I reconsider my whole design.

HAST also has *async* mode, where it acknowledges data as soon as it's written to the primary node's disk. The data will be sent to the secondary node, and will get written to the disk there, but the program won't wait for that. Async mode dramatically increases the risk of data loss, as the primary node might fail before the secondary node even receives the data.

Every time I've used async mode, I've regretted it. The only time you should even consider async mode is if the secondary node is so far away from the primary that the speed of light induces unacceptable latency. Even then, I would encourage re-architecting your application rather than using async mode.

Set the replication mode with the *replication* keyword and the desired value.

```
replication fullsync
```

Replication mode is only one way to manage integrity, though.

Checksum

HAST can calculate a checksum for each chunk of data, and transmit that checksum with the data. The secondary host uses that checksum to verify that the data it writes to disk is the same as the data the primary host sent.

The HAST checksum is not intended for protection against a malicious attacker. Both the checksum and the data are transmitted in plain text on the same wire. An intruder who can alter that data can also alter the checksum to match. Checksums are useful for detecting transmission errors, however. HAST supports the CRC32 and SHA256 checksums.

Set the checksum with the *checksum* keyword.

```
checksum sha256
```

I always use a checksum in production. Most modern systems have processing power that far exceeds either their disk or network throughput. Calculating SHA256 checksums requires more processing power than calculating the CRC32 checksums. The CRC32 checksum is perfectly adequate to catch the common sorts of bit errors you'll probably encounter. Personally, I'm still a SHA256 bigot.

By default, HAST does not use any checksums. You must enable them if you want them.

Compression

Compression exchanges processor time for network bandwidth. If you have a lot of disk writes, compressing them can reduce the amount of bandwidth HAST requires so that it will actually fit on your network. Use the *compression* keyword to set a compression method.

HAST defaults to *hole* compression, which means that only blocks made up of all zeroes get compressed. Hole compression vastly accelerates resource initialization, as a brand-new disk device contains only zeroes. It doesn't do much for day-to-day performance, however.

If you want to routinely compress data, HAST supports the *lzf* compression algorithm. LZF gets used in many places, including in ZFS. Compression won't hurt, but it doesn't always help as much as you might hope. Enabling LZF in HAST when you have LZF compression on the ZFS filesystem on top of the HAST device won't do you much good.

You can also completely disable compression with the *none* option.[1]

```
compression lzf
```

I normally use LZF, because even when compression doesn't help it won't hurt.

Metadata Flushing

Hard drive write caches can make the hardware put data on disk out of order. HAST might tell the hard drive to write block A and then block B, but the write cache can get in the middle and decide to write block B first. If block B requires block A, though, and the system fails between writing the two blocks, the hardware has just corrupted the filesystem. This is most common with filesystem metadata.

The *metaflush* option tells `hastd` to automatically flush the write cache after every metadata update. HAST normally flushes the write cache after updating metadata. If the hardware doesn't support write cache flushing, though, HAST stops trying to flush it.

I’m not aware of any time you would want to set `metaflush` to `off`. I only bring up this option to encourage you to buy good hard drives that support write cache flushing.

HAST Networking

You can control how HAST listens to the network, where it sends connections, and the source address it binds to for outgoing connections.

The *listen* keyword tells `hastd` what address to bind to for incoming connections. This is the IP address and port where the secondary host listens for updates, and where the primary listens for acknowledgements. HAST defaults to listening to TCP port 8457 on all available IP addresses and interfaces.

You can use the `listen` option globally, or in a special per-server section of *hast.conf*. Here I lock `hastd` down to one IP address on each of my hosts.

```
on www1 {
  listen 192.0.2.1
  }

on www2 {
  listen 192.0.2.2
  }

resource hast1 {
  on www1 {
…
```

You can use IPv6 addresses as well, of course. And you can change the port by specifying the new port with a colon and the port number.

```
listen 192.0.2.1:8888
```

If you use the `listen` keyword globally, you're saying that all hosts listen to the network on the stated IP address. The whole point of IP addresses is that they're unique to a machine. A global `listen` keyword is only useful to change the TCP port `hastd` uses when your hosts' `hastd` attaches to all available IP addresses.

```
listen 0.0.0.0:80
```

I encourage you to create per-host entries and limit the number of addresses `hastd` listens on.

Within each node entry for each resource, you can control the address of the remote peer and the local address used to send to that peer. We've already seen the `remote` keyword, a necessary part of any HAST configuration. The *source* keyword goes right next to `remote`, and lets you dictate the source port for outgoing connections to that peer.

```
  on www1 {
    local /dev/zvol/zroot/hast1
    remote 192.0.2.2
    source 192.0.2.1
  }
```

You would need to use `source` if your host has multiple IP addresses on that subnet and the remote peer uses packet filtering to restrict connections to `hastd`.

Failures, Startup, and Split Brain

A HAST pair with only one working node enters the *degraded* state. If a HAST node reboots, the other node sees the HAST device as degraded.

```
# hastctl status
Name    Status   Role     Components
hast1   degraded primary  /dev/zvol/zroot/hast1  192.0.2.2
```

If the primary node fails, presumably you or the system will switch the secondary node to be primary until the other node comes back to life. Or, maybe not.

This uncertainty dictates a key aspect of HAST behavior. While you can start `hastd` on boot, HAST makes no assumptions about the peer state. At system boot, HAST devices enter the `init` state. They're ready for action, as soon as you tell them how they fit into your world.

```
# hastctl status
Name   Status  Role  Components
hast1  -       init  /dev/zvol/zroot/hast1  192.0.2.1
```

You must use `hastctl` after the system boots to tell each HAST device its role.

In normal operation, you'd think that a newly booted HAST server would assume a secondary role so that the primary can send all the updates needed to bring the freshly booted device up to date. That would work—except when both machines are newly booted. When both machines boot simultaneously they'll both sit there, wait for the other to claim the primary role, and whinge into */var/log/messages*. You need a way to for one node to claim the primary role. If you have a log-watching program, you could have it promote a preferred node when the relevant log messages appear. You could log in and manually promote one of the nodes to primary. FreeBSD can automatically promote one for you through CARP and devd(8), as discussed in Chapter 10.

HAST behaves cautiously because it is possible to have multiple hosts simultaneously claim the primary role, causing a *split brain* condition. Split brain is bad. Each host modifies the HAST device as it thinks it should, and sends updates to the HAST peer. The peer doesn't listen, because it thinks it's the master. The data on each copy of the device differs. It's your job to figure out how to resolve those differences.

If software can readily examine the data on the two HAST devices—say, if ls(1) displays what you need to know—you could programmatically resolve the difference with shell scripting. If they're more complex records, such as a database or logs, you'll need to analyze the data itself to capture and reconcile the changes. How you do that depends entirely on the data.

The split brain issue is why I recommend not using HAST as database storage back end. Clustered databases like PostgreSQL, and even MySQL or MariaDB, have built-in routines, well understood processes, or accepted hacks for resolving split brain problems. Filesystems do not.

In many cases, sysadmins struggle with data reconciliation, give up, and resolve the split brain by destroying one copy of the HAST device, re-initializing it, and letting HAST resync. They accept a small amount of data loss in the name of restoring service.

Once you've reconciled the data differences caused by the split brain, resynchronize the two HAST providers. You can't tell HAST to shuffle bits back and forth until the two devices are all caught up with each other. Rather, you must wipe out and re-initialize the HAST resource on the secondary host, forcing a complete resynchronization of the backup host. Put the resource in `init` mode, then rerun the `create` command and assign it the secondary role. Here I recreate the backup HAST resource hast1.

```
# hastctl role init hast1
# hastctl create hast1
# hastctl role secondary hast1
```

The primary feeds the secondary everything on the device, restoring integrity.

But it's best to manage HAST meticulously and not split the brain in the first place.

Commands on HAST Events

Use the *exec* keyword to tell HAST to run a command on an event. HAST runs this script when the resources change roles, at the initial `hastd` connection between hosts, when a `hastd` peer disconnects, when a secondary resource starts synchronizing with the primary, when a resource finishes synchronizing, an interrupted synchronization, and when `hastd` detects a split brain condition. Here I run the script */usr/local/scripts/haststates.sh* any time any event happens.

```
exec /usr/local/scripts/haststates.sh
```

You cannot have per-event scripts; one script must handle all of these events. HAST gives the script different arguments depending on the event.

The script runs immediately after the event. You cannot put commands to prepare for the event in the switch. For example, having the script export your HAST-backed ZFS pool when the host switches to the secondary role won't work—the pool's back end is gone when `hastd` runs the script! You could have the script import a ZFS pool when the host assumes the primary role, however.

Script Arguments and Execution

At a role change, `hastd` runs the command with four arguments: the word `role`, the resource name, the old role, and the new role. Both hosts run the script.

When `hastd` on two hosts first connect to each other, both run the script with two arguments: the word `connect` and the name of the HAST resource. The script gets run once for each resource the hosts share. Similarly, when two hosts lose their `hastd` connection, the script runs with the argument `disconnect` and the name of the resource.

The synchronization events run the script with the event name (either `syncstart`, `syncdone`, or `syncintr` for interruptions) and the resource name. Synchronization events only trigger the script on the primary node.

Finally, in the event of a split brain, the script is run with the argument `split-brain` and the resource name. The script runs on both primary nodes.

hastd(8) Scripts

A `hastd` script is a set of nested case statements. I recommend starting with a simple script on test servers to log these arguments and whenever the script is run. Abuse your test servers and see what sort of messages get logged.

```
#!/bin/sh
logger hastd event: $1 $2 $3 $4
```

Here I use the script for two functions. First, it mails the server's `root` account[2] whenever a HAST resource starts synchronizing. Ideally, you'd replace the email with an SNMP trap or another call to the monitoring system. Theoretically, HAST resources should always be synchronized. A deliberate sync is a hint that something's wrong with the connection between the two hosts. (You'll also see synchronization whenever the two hosts switch roles, so this notification does double duty.) Secondly, it imports a ZFS pool named after the resource when the host is promoted to primary role. That is, when this host becomes primary for the HAST resource *shared1*, it then imports a ZFS pool also called *shared1*. You could also start processes here. Last, all events get logged.

```
#!/bin/sh

HOSTNAME=$(hostname)

case $1 in

  role)
    case $4 in
      primary)
        sleep 3
        zpool import -f $2
        logger “importing pool $2”
        # start your programs here
        ;;
      *)
        ;;
    esac
  ;;

    syncstart)
      mail -s “sync started for $2 on $HOSTNAME” root <<EOF
HAST sync started, what happened?
EOF
      ;;
  *)
  ;;

esac

logger hastd event: $1 $2 $3 $4
```

Add other cases as needed for your environment.

Note that the script waits three seconds between becoming the master node and importing the ZFS pool. HAST needs a little bit of time to perform the switch and verify everything is caught up. If you have a large HAST device, test the switch and adjust the timing as needed.

If you’re using UFS on HAST, you can replace the zpool import with a call to `fsck` and a `mount` command. Giving the mount point and the HAST resource the same name will simplify your script.

The hast.conf(5), hastd(8), and hastctl(8) manual pages have even more on configuring and using HAST. To see how to automatically switch the primary device when a host fails, however, read the next chapter.

---

[1] I am not aware of any circumstances in which you would need to disable HAST compression, but I have faith you can find it. At 3 AM, while a bunch of angry people scream at you.

[2] Your server *is* configured to send mail to a human being with the job of reading it, isn’t it? Isn’t it?

Chapter 10: Networked Disk Failover

Running disks and filesystems over the network presents interesting possibilities. The most obvious is the ability to keep a data store running even when the host supporting it goes down. FreeBSD includes tools to let a backup host take over when a primary host fails, using standard hardware found in every datacenter.

When you first try networked disk failover, use a test environment. Be prepared to reboot. A lot.

Any type of failover automation requires that you be comfortable with basic shell scripts. This book provides scripts to get you started. They’re also available on the author’s GitHub site (https://github.com/mwlucas).

Clustering Risks

Any two-node clustering solution on commodity hardware has a risk of going some style of split brain. HAST can issue a literal “split brain” error, but iSCSI-backed high availability can have identical problems when multiple hosts mount the same UFS filesystem or import the same ZFS pool. A flaky network switch can not only interrupt client connectivity, but also convince each server that its peer has gone down and that it should become master. Merely unplugging a network cable can trigger this error. Certain stages of operating a two-node cluster really do require highly intelligent management.

Advanced cluster systems usually have at least three hosts involved in the cluster, so that they can use a quorum method for determining the master host. In a quorum system, a majority of the hosts involved must agree on which host is acting as master. One isolated host cannot declare itself to be the dictator. These systems fill a book on their own and have greater hardware requirements.

Look carefully at your hardware, and investigate FreeBSD’s support for it. Some hardware has solutions for these problems. For example, if you’re one of the lucky few with shared SCSI shelves, you could use SPC-3 SCSI persistent reservations via camcontrol(8) in your failover script. If you have multipath SCSI behind your iSCSI devices, investigate FreeBSD’s high availability support through Asymmetric Logical Unit Access (ALUA) as discussed in ctl(4).

Any specific hardware recommendations I would make would go obsolete before you could read this book, so I encourage you to do your own hardware research.

The systems described here are as safe as possible given the software’s limitations and commodity hardware. If you decide to build a two-host cluster, be prepared to recover should the worst happen.

Many people successfully use CARP for filesystem failover. They’ve thought long and hard about how to recover from problems. Be sure you do the same before deploying.

Failover Architecture

Failover automation is built on top of two key technologies, CARP and devd(8). How they get used depends on the storage back end used.

CARP

The Common Address Redundancy Protocol (CARP) is an OpenBSD creation for sharing IP addresses between machines. It lets network addresses float between machines as needed. CARP resembles Cisco's Virtual Router Redundancy Protocol (VRRP), but also includes checks for security and integrity. Additionally, CARP lacks the burden of patents, so you're free to use, distribute, or re-implement it for any purpose whatsoever.

Each host in a CARP cluster has a CARP attribute attached to one or more interfaces. Each interface with CARP continually announces its presence to the other hosts on the network. The hosts negotiate to see which one will be the *master*. While each interface running CARP has an IP address assigned just for that machine, the CARP master answers all requests for the floating CARP IP address. When the master host stops responding, another host in the cluster takes over service and claims the CARP IP.

Clients know only about the CARP IP address, not the individual machine's management address. When the address floats to the other machine, client requests follow it. The trick is to get the filesystem and server processes to follow it.

Devd

When a CARP interface goes up or down, taking over or relinquishing an IP address, it sends an event to devd(8). As Chapter 2 discusses, `devd` events can trigger commands. You can have FreeBSD activate storage when the CARP interface goes up and deactivate storage when the interface goes down.

Storage Back Ends

FreeBSD has two block storage systems that hosts can share: iSCSI (Chapter 8) and HAST (Chapter 9). Which you should use depends on your environment.

In a large environment, iSCSI is the best choice for networked storage devices. Have two separate servers offer iSCSI targets, and another two client-facing servers. The client-facing servers can use one drive from each iSCSI server to create a mirror, or multiple drives from each to create striped mirrors. The environment can withstand the failure of one iSCSI server and one client-facing server. So long as you have one server of each type active and working, the service remains up. You can easily expand this environment and replace hardware as needed. You will probably want a stand-alone Storage Area Network (SAN), to separate your storage traffic from the client and management traffic. For even higher availability, investigate multipath SCSI.

Using RAID-Z for high availability is possible, but requires more iSCSI targets on more independent servers. An eight-disk RAID-3 can withstand the loss of three disks. If you have two iSCSI servers, losing one server means losing four disks—and the pool. You could create a separate RAID-Z device on each iSCSI server, and then mirror the pools.

If you have only two hosts, HAST is your only real choice for failover filesystem. It would be possible to use iSCSI between only two hosts, each providing a drive to the other, but that reaches astonishing new levels of clunky. Ideally, the HAST hosts will have a private network connection between them, such as a crossover cable.

Let’s work through this architecture one piece at a time.

Configuring CARP

In FreeBSD 10 and later, CARP is an attribute attached to an existing interface. Older versions of FreeBSD (and all versions of OpenBSD) created a virtual interface for CARP, but the new model simplifies management on hosts with many interfaces. CARP is incredibly flexible, so we'll only cover its basic configuration. For far more details, see carp(4).

Start by loading the carp(4) kernel module at boot with a `loader.conf` entry.

```
carp_load=YES
```

The basic unit of CARP is the *virtual host*. A virtual host, and IP addresses attached to that virtual host, can float between machines. Each virtual host needs an ID number, or *VHID*. Most people start numbering VHIDs at 1.

Each VHID also needs a password. The password prevents rogue hosts on the same network from spoofing CARP announcements. Like iSCSI passwords, a CARP password is stored in plain text on each machine, as anyone who configures the host's network needs access to it.

Configure CARP as an Ethernet interface alias in `/etc/rc.conf`. Here interface em0 has its own address, but it also has a CARP VHID.

```
ifconfig_em0="inet 203.0.113.214 netmask 255.255.255.0"
ifconfig_em0_alias0="vhid 1 pass Gelato alias 203.0.113.219/32"
```

Both hosts need exactly the same CARP alias configuration. (They need unique primary IP addresses, of course.) You could add additional CARP configuration to both, such as hard-coding one host as the CARP master, but that imposes additional risks. See "Cluster Startup" later this chapter.

Reboot the first host. You should see the CARP information in `ifconfig`.

```
# ifconfig em0
  …
  inet 203.0.113.214 netmask 0xffffff00 broadcast 203.0.113.255
  inet 203.0.113.219 netmask 0xffffffff broadcast 203.0.113.219 vhid 1
  …
  carp: MASTER vhid 1 advbase 1 advskew 0
```

This host has its own IP address, 203.0.113.214. It also has the IP 203.0.113.219, as part of VHID 1. The last line shows that this interface is a CARP *master*, meaning that it's currently servicing requests for this VHID.

Reboot the second host. Its `ifconfig` output should show that host in the *backup* state. If the master fails, the backup will assume service.

Force a host to change roles by using `ifconfig`. Give the interface name, the VHID, and the new state.

```
# ifconfig em0 vhid 1 state backup
```

I normally run such commands on the master host to tell it to demote itself to backup status, but they work just as well on the backup.

If this is your first experience of CARP, play with it before continuing. Start an ongoing ping of your VHID IP. Reboot the master CARP host. Depending on your network switch and the server hardware, your client might drop a ping as the CARP IP address moves from one host to the other. When the host finishes rebooting, it will see that the other host has claimed master status and will take the backup role for itself.

Once you understand how VHIDs float between machines, we’ll configure `devd`.

CARP and devd(8)

On FreeBSD, CARP interfaces send events to devd(8) when they change roles. On older versions, when CARP ran as a virtual interface, those events were up and down notifications. Now that CARP is an attribute applied to an existing interface, those events specifically notify `devd` when the interface assumes the master or backup role. If you're unclear on how any of this works, review Chapter 2.

Processing these events requires custom rules to trigger shell scripts. I advise putting these rules in */usr/local/etc/devd/carp.conf*. As we don't have filesystem failover scripts yet, here I use these events to trigger a log entry.

```
notify 30 {
  match “system” “CARP”;
  match “subsystem” “[0-9]+@[0-9a-z]+”;
  match “type” “MASTER”;
  action “/usr/bin/logger devd carp up event detected”;
};

notify 30 {
  match “system” “CARP”;
  match “subsystem” “[0-9]+@[0-9a-z]+”;
  match “type” “BACKUP”;
  action “/usr/bin/logger devd carp down event detected”;
};
```

Have your hosts switch between the master and backup roles a few times. Run `tail –f /var/log/messages` in a separate terminal window, so you can see the logs appear. Once it's clear that `devd` runs commands as you expect, you're ready to actually implement filesystem failover.

Failover Scripts

No matter which storage back end you use for your high-availability networked disk, you'll need three basic scripts.

At system boot, you must put the storage in a usable state. This startup script can't make the newly booted host the cluster master—that's likely to cause a HAST split brain or a corrupt ZFS pool. But at boot you can safely attach iSCSI targets, or put HAST devices in a secondary role.

When a host becomes the CARP master, you'll need to activate the storage. HAST hosts must switch to the primary role and import or mount any filesystems. iSCSI hosts must import or mount any filesystems. Finally, this script needs to start any processes that use the filesystem.

When a host becomes the CARP backup, quickly stop any processes using the filesystem. Those processes are no longer serving client requests—the IP address clients go to for service is no longer attached to this machine. After terminating the processes, unmount or export the filesystem. If you're using a HAST device, the host switches the HAST device to the secondary role.

Before deploying failover scripts, test them. Use the script to demote the master to a backup. Then run the promotion script on the alternate host. Make sure that everything mounts automatically. Reverse the process: demote the new master, and then promote the backup. Remember, you can't have two master hosts simultaneously—that causes split brain problems or an otherwise corrupt filesystem. Once you're confident that the scripts work with your environment, edit */usr/local/etc/devd/carp.conf* to have CARP state changes automate the failover process. Then trigger failovers by changing CARP state. Reboot each host and see that the other takes over. Last, perform "unceremonious shutdown" tests. Pull the power cord on one host, and make sure that the other takes over.

Batter and abuse your test hosts as thoroughly as possible. Each test you perform before you deploy reduces the risk of data loss in production.

Here are some sample scripts for each storage back end. These scripts use ZFS, but you can easily modify them to use UFS. They conform to my scripting prejudices—specifically, they're small scripts that perform a single task each. The promotion and demotion scripts are based on Freddie Cash's HAST and CARP failover scripts. I've folded, spindled, and mutilated them so badly that while he certainly needs credit for his work, he no longer bears any responsibility for their failures. Feel free to rearrange them to suit your environment and prejudices. I did.

HAST Failover Scripts

When `hastd` starts, it places all HAST devices in `init` mode. They're ready to do something, as soon as you tell them what to do. This is for safety reasons; if two hosts both decide that they're the primary node for the same HAST devices, you and your server both develop a split brain error.

When you're designing a HAST-backed failover system, remember that HAST synchronizes devices in serial, rather than parallel. A host must synchronize every HAST device before switching roles. If a host has eight HAST devices, the slowest part of the failover process will take eight times longer than it would with only one device.

You can automatically put your HAST devices in the secondary role safely, however. If the other node has been running and has assumed the master role, everything will just work. If both nodes boot simultaneously, they'll both be in secondary mode. Services won't be up, but you won't have data loss. You must run `hastctl` after `hastd` starts, however. Here's a very stripped-down rc.d script that accomplishes this. I've skipped most of the rcorder(8) functions, as putting your HAST devices in a usable state isn't really a service.

```
#!/bin/sh
#

# PROVIDE: hastctl
# REQUIRE: hastd

hastctl role secondary hast1
hastctl role secondary hast2
```

Make an entry for each of your HAST devices. Or add loops, for fancy. If you have so many HAST devices that you need loops, though, reconsider your application architecture. It's best to put any ZFS redundancy below the HAST devices. Remember, the purpose of sending ZFS metadata over HAST is to make your redundant disk slower. Copy this script to */usr/local/etc/rc.d/hastctl* and make it executable.[1]

These failover scripts assume that your ZFS pools have the same name as your HAST resources. Each pool should have only one device beneath it—remember, any redundancy belongs below the HAST device.

If you use UFS with HAST, also name your UFS mount points after your HAST resources. Replace the zpool(8) commands with calls to umount(8).

For the master-to-backup script, you must set `resources` to the name of your HAST resources. You also must identify processes to stop. In this example I've used `httpd`, and stopped it by using service(8). If all else fails, your script can use `pkill -9` to terminate processes with prejudice. By the time this script runs, the CARP VHID is on the other host. This machine is no longer providing service.

```
#!/bin/sh

# The names of the HAST resources, as listed in hast.conf
# Use the same name for your pool or mount point

resources="hast1 hast2"

#logging
log="local0.debug"
name="hast-carp-demote.sh"

#terminate processes here

service httpd stop

#end of user configurable stuff
#do not go beyond this point

for disk in ${resources}; do


  #forcibly unmount the filesystem

  zpool export -f ${disk} 2>&1
  if [ $? -ne 0 ]; then
    logger -p $log -t $name "Unable to export the pool ${disk}."
    exit 1
  fi

  # Switch roles for the HAST resources
  hastctl role secondary ${disk} 2>&1
  if [ $? -ne 0 ]; then
    logger -p $log -t $name "Unable to switch role to secondary for resource ${disk}."
    exit 1
  fi

  logger -p $log -t $name "Role switched to secondary for resource ${disk}."
done
```

Copy this script to */usr/local/scripts/hast-carp-demote.sh* and make it executable.

The backup-to-master script below also requires that you set resources to the name of your HAST resources. This script doesn't actually mount any filesystems. Use a HAST events script, as shown in Chapter 9, to mount the filesystems. Start any processes either in the HAST events script or at the end of this script.

```
#!/bin/sh

# The names of the HAST resources, as listed in hast.conf
# Use the same name for your pool or mount point

resources=“hast1 hast2”

#logging
log=“local0.debug”



#end of user configurable stuff
#do not go beyond this point

name=“hast-carp-promote.sh”

logger -p $log -t $name “Switching to primary provider for ${resources}.”



# Wait for all “hastd secondary” processes to stop

for disk in ${resources}; do
  while $( pgrep -lf “hastd: ${disk} \(secondary\)” > /dev/null 2>&1 );
  do
    sleep 1
  done



  # Switch role for each disk
  hastctl role primary ${disk}
  if [ $? -ne 0 ]; then
    logger -p $log -t $name “Unable to change role to primary for resource ${disk}.”
  exit 1
  fi
done

logger -p $log -t $name “Role for HAST resources ${resources} switched to primary.”

#Let HAST script mount the filesystem

service httpd start
```

Copy this script to *`/usr/local/scripts/hast-carp-promote.sh`*.

Test your scripts as described at the beginning of this section. Once you’re confident that they work, you can edit *`/usr/local/etc/devd/carp.conf`* to trigger these scripts automatically.

```
notify 30 {
  match “system” “CARP”;
  match “subsystem” “[0-9]+@[0-9a-z]+”;
  match “type” “MASTER”;
  action “/usr/local/scripts/hast-carp-promote.sh”;
};


notify 30 {
  match “system” “CARP”;
  match “subsystem” “[0-9]+@[0-9a-z]+”;
  match “type” “BACKUP”;
  action “/usr/local/scripts/hast-carp-demote.sh”;
};
```

You now have HAST-backed failover.

iSCSI Failover

iSCSI failover is both simpler and more complicated than HAST-based failover. iSCSI-backed systems are far more flexible than HAST and allow much greater expansion. On the other hand, HAST is specifically designed as a failover protocol. Its very design lets you verify that storage is not in use before mounting it. Any number of hosts can simultaneously attach to iSCSI targets. Multiple hosts simultaneously mounting a UFS partition or using a ZFS pool on those targets will corrupt the filesystem. Not might—*will*. Those accesses might even panic the hosts, corrupt the buffer cache, spread the problem to other filesystems, and trigger a rain of frogs. It will unquestionably cause meetings, with you in the hot seat.

Performing iSCSI-based failover requires a protocol specifically designed for failover. You need to know, conclusively, that the other host is not using the disk before trying to import it. Fortunately, we have one. We’ll use HAST. We won’t use any data on the HAST device, but rather piggyback on the HAST protocol. You can find other options, including failover daemons written for this specific purpose, but they don’t have any features we need, and HAST is included in the base system.

Start by creating a very small HAST device. I’m using a 10 MB zvol as the backing store.

```
# zfs create -V 10M zroot/failover
```

With a HAST device like this, we don’t need much in the way of data integrity. The presence of the HAST device permits mounting the filesystem, not any data stored on the HAST device. Here’s a bare-bones but completely sufficient *hast.conf* file.

```
resource failover {
  on www1 {
    local /dev/zvol/zroot/failover
    remote 192.0.2.2
    }
  on www2 {
    local /dev/zvol/zroot/failover
    remote 192.0.2.1
    }
}
```

As with any HAST environment, you must have a solid and reliable network between the hosts. A crossover cable, as many other failover solutions require, would certainly not be amiss. But the data load is tiny, so you might be fine running this flag through the primary network interface. If your organization has invested lots of money in a big iSCSI array, though, you'll feel really embarrassed if a bad network on the front end causes ZFS corruption because you wanted to save a network card.

You'll need a startup script to put the HAST device in a backup state at system boot. (See "Cluster Startup" later this chapter for more discussion of system state at boot.) You'll also need to log onto the iSCSI targets. The cluster startup script *`/usr/local/etc/rc.d/cluster`* looks like this.

```
#!/bin/sh
#

# PROVIDE: cluster
# REQUIRE: hastd

hastctl role secondary failover
iscsictl -Aa
```

The host serving as HAST primary now has a device node, *`/dev/hast/failover`*. We'll use the presence of this device node as a trigger to mount or import the iSCSI-backed filesystem.

The backup-to-master iSCSI script must first check to see if the host still thinks it's a HAST backup node. Once it's no longer acting as a HAST backup, the host can promote itself to primary, import the pools, and start services. You'll need to edit this script to name your pools at the beginning, and start services at the end.

```
#!/bin/sh

pools="data1 data2"

#logging
log="local0.debug"
name="iscsi-carp-promote.sh"

#main script
logger -p $log -t $name "Becoming main storage, waiting for completed export"

while $( pgrep -lf "hastd: failover \(secondary\)" > /dev/null 2>&1 ); do
```

```
  sleep 1
done

hastctl role primary failover

logger -p $log -t $name “Clear to become main storage, importing ${pools}.”

for pool in ${pools}; do
  zpool import -f ${pool}
  if [ $? -ne 0 ]; then
    logger -p $log -t $name “Unable to import ${pool}.”
    exit 1
  fi
done

logger -p $log -t $name “Pools ${pools} imported.”

#start your services here

service httpd start
```

Copy this script to */usr/local/scripts/iscsi-carp-promote.sh* and make it executable.

The master-to-backup script here is simpler, and requires only that you set the name of your ZFS pools and configure commands to terminate processes using those pools. Remember, when the host switches to backup mode, clients are no longer accessing this host. You can ungracefully kill most software if necessary.[2]

```
#!/bin/sh

# The names of the ZFS pools available on iSCSI

pools=“data1 data2”

#logging
log=“local0.debug”
name=“iscsi-carp-demote.sh”

#terminate processes here, forcibly if needed

service httpd stop

#end of user configurable stuff

for pool in ${pools}; do

  #forcibly unmount the filesystem

  zpool export -f ${pool} 2>&1
  if [ $? -ne 0 ]; then
    logger -p $log -t $name “Unable to export ${pool}.”
    exit 1
  fi

  logger -p $log -t $name “Pools ${pool} exported.”
done

#if we get this far, we signal HAST that the other host can import.

hastctl role secondary failover
logger -p $log -t $name “HAST demotion triggered.”
```

Copy this script to */usr/local/scripts/iscsi-carp-demote.sh*. Test your failover by hand, as discussed at the beginning of this section. Once you’re certain they work, integrate them into devd(8) as shown here.

```
notify 30 {
  match “system” “CARP”;
  match “subsystem” “[0-9]+@[0-9a-z]+”;
  match “type” “MASTER”;
  action “/usr/local/scripts/iscsi-carp-promote.sh”;
};



notify 30 {
  match “system” “CARP”;
  match “subsystem” “[0-9]+@[0-9a-z]+”;
  match “type” “BACKUP”;
  action “/usr/local/scripts/iscsi-carp-demote.sh”;
};
```

You now have live failover of iSCSI-backed systems.

Cluster Startup

These scripts place a host in backup mode at boot. If you reboot one host at a time, everything fails over correctly. A cold start of both nodes, however, is more problematic. An individual system has no way to tell that the whole cluster has been cold booted.

FreeBSD starts its network early in the boot process. The network includes CARP, of course. It starts CARP before HAST or iSCSI are live. Do you want a rebooting server to automatically claim the master role from a fully functional peer? Probably not—it won't have the storage ready, or its applications, or even SSH. Debugging a troubled server means rebooting it multiple times.

When both nodes boot simultaneously, I recommend having a third party decide which node to make the master. This third party might be you. It might be a piece of software on another machine. Running the `promote` script on the select master node should bring it fully up and initiate the failover.

Are there alternatives? Sure. They all boil down to: what risks can you accept? You could write a system boot script to enable and configure CARP well after the host boots. You could declare that host A always claims the master role when it boots. Each of these has risks, depending entirely on your environment and architecture. Think carefully, and choose the most automated solution that imposes only acceptable risk.

To really dive into risk, let's talk about Access Control Lists.

[1] If I have to tell you how to make a file executable, you should not be implementing highly available filesystems.

[2] If you can't kill this particular software ungracefully, it's probably a database. And I warned you against running databases on filesystem-level failover back in Chapter 9, so I have a complete lack of sympathy for you.

Chapter 11: NFSv4 Access Control Lists

Using FreeBSD as a storage back end for a corporate NFS or CIFS file store requires replicating the organization’s structure and information control in file permissions. The traditional Unix user/group/everyone model just won’t cut it. You need an access control model that lets you construct permissions and privileges any way you need.

That’s where Access Control Lists, or ACLs, come in.

ACLs let you select from a list of privileges, and apply them to any system user or group, creating rules for who may access what. You want a file readable by anyone in the company, but editable only by the people in Billing, except for Fred, because he ticked off the chief accountant? An ACL can do that. Maybe you have some files that you want users to be able to write to, but not delete. That’s an ACL. Or perhaps you have dozens of users, each of whom gets slightly different access to a specific file. An ACL can do that. You’ll go totally bonkers from administrative overhead, mind you, but it’s possible.

While understanding ACLs is pretty straightforward, the actual practice of implementing and managing ACLs quickly gets complicated. The reason Windows NTFS ACLs are so frequently cursed is not because they’re on Windows. It’s because people use ACLs to map and emulate human relationships, and human relationships often merit cursing. Ponder any large organization you’re a member of, and all the ways different groups of people in that organization want to offer and control access. Imagine writing rules in software to emulate that. Plus, today’s heroic employee is tomorrow’s scapegoat, so you’ll have to redo everything at random intervals.

The key to successful ACL management is simplicity. Don’t assign ACLs just because you can. Assign them only when needed.

ACLs are always interpreted on the local host. If you’re sharing a filesystem via NFS or Samba, the client can think whatever it wants about the mounted share and can issue requests accordingly. The server interprets any ACL itself, though, and responds to the client as the ACL dictates.

ACL types

While many different styles of Access Control Lists have been implemented over the years, the three you'll encounter most often today are POSIX, NTFS, and NFSv4.

Portable Operating System Interface (POSIX) is a set of standards for operating system interoperability. One proposed POSIX standard, POSIX.1e, describes a set of Access Control List behaviors. The proposed ACL standard underwent several revisions, and many operating systems implemented one of those revisions. Everyone who implemented these draft ACLs assumed that they'd update their implementation when the final draft became a standard. At the end, though, POSIX.1e was not adopted as a standard. All of these operating systems thus have slightly different "POSIX" ACL implementations. FreeBSD supports POSIX.1e ACLs, but they're not recommended, and so I don't cover them. Anything that can be implemented with a POSIX ACL can also be implemented with an NFSv4 ACL.

Microsoft created NTFS Access Control Lists. The NTFS ACL model has more features and types of access control than POSIX ACLs, and is widely deployed in enterprise environments. Entries in an NTFS ACL are processed in order, on a first-match basis.

NFSv4 ACLs were created as part of the NFS version 4 negotiations. They closely resemble NTFS ACLs, and were specifically created to allow Unix-like hosts to better serve Windows clients. They're so close that NFSv4 ACLs can be used on a Samba server and managed via Windows security tools. Like NTFS ACLs, entries in an NFSv4 ACL are processed in order. Unlike other ACLs, each privilege is checked separately. The first matching rule for each privilege wins. A request can accumulate its needed privileges from several different access rules. (See "ACE Ordering and Deny" for more discussion.) Also like NTFS, NFSv4 ACLs default to denying access. If you need ACLs on FreeBSD, use NFSv4 ACLs.

FreeBSD's NFSv4 ACL implementation is deliberately modeled after that found in OpenSolaris-derived operating systems. The management interface differs, but the various permissions and ACL handling traces OpenSolaris fairly closely. Other operating systems implement these ACLs differently. Linux, for example, implements its NFSv4 ACL support atop POSIX ACLs, creating something compliant enough for most users.

This chapter does not cover everything there is to know about NFSv4 ACLs. The NFSv4 specification is about 135,000 words, or almost three times the size of this entire book, and the ACL details are layered throughout it. Much of that is written in pseudocode. Converting that to a systems administration text would fill many volumes, and would include lots of stuff a sysadmin doesn’t need (or want!) to know. The goal here is to orient you to how NFSv4 ACLs function and behave, so that you can make vaguely sensible design decisions and have a reasonable hope of identifying and solving common problems.

ACLs and Filesystems

All ACL implementations require support in the underlying filesystem. UFS supports POSIX and NFSv4 ACLs. Both UFS and ZFS support NFSv4 ACLs, but they behave slightly differently. I'll give the details in "ACL Inheritance" later this chapter, but here are the basics.

ZFS is preferable for NFSv4 ACLs, and ZFS enables them by default. ZFS offers two modes of passing directory ACLs down to files and subdirectories. Users who want the more popular way of inheriting ACLs, the one expected by Windows clients, need to set the properties **`aclmode`** and **`aclinherit`** to `passthrough` on any dataset before creating any files or directories that will need ACLs. Set this value before creating any files or applying any ACLs.

You can use NFSv4s ACLs on UFS filesystems. UFS filesystems support only one method of ACL inheritance, however, and while it's the less popular method, it might do for you. You must enable ACLs with the `nfsv4acls` mount(8) option.

ACL Format

An NFSv4 Access Control List contains a list of Access Control Entries (ACEs). Each ACE describes the permissions assigned to a single entity such as a user, a group, or everyone on the system. The easiest way to understand ACLs is to look at a simple one. Use getfacl(1) to view the ACL on a file or directory.

Here I have a file with very common permissions.

```
# ls -l file1
-rwxrw-r– 1 mwl mwl 0 Nov 16 13:16 file1
```

I have not assigned any special ACLs to this file. The owner can read, write, and execute the file, members of the owning group can read and write the file, and everyone else can read it. Let’s see how these permissions show up as an ACL.

```
# getfacl file1
# file: file1
# owner: mwl
# group: mwl
            owner@:rwxp--aARWcCos:-------:allow
            group@:rw----a-R-c--s:-------:allow
         everyone@:r-----a-R-c--s:-------:allow
```

Where did this ACL come from? It’s built out of the standard file permissions. Once I apply an actual ACL to this file, the filesystem permissions and the ACL diverge. This ACL contains three ACE entries, each with four colon-separated fields.

The first field, the ACL *tag*, gives the type of entity this ACE applies to. The tag for the first entry, `owner@`, says that this ACE is for the file owner. The second ACE tag, `group@`, means this ACL is for the group owner, while the third, `everyone@`, represents everyone on the system. (“Everyone” in traditional Unix permissions means “everyone except the owner and group owner,” while in NFSv4 ACLs it means “everyone *including* the owners.”)

The second field gives the ACL *permissions* assigned to the tag. The first three characters of each should look familiar; they’re the standard Unix-style permissions for this user or group. For example, the tag `owner@` (representing the file owner) gets `rwx`, just as you see in `ls -l`. These get a whole bunch of new possible settings, though, represented by the jumble of letters after the leading three. We’ll see what privilege each letter represents in “ACL Permissions,” later this chapter.

The third field gives any inheritance settings for the files and directories. Files and directories can inherit ACL settings from their parent directories or directories above them. As this file has built its ACL out of standard Unix-style permissions and there's no ACL on the parent directory, it hasn't inherited any ACL-specific information.

The last entry, the ACL *type*, says that matching items are either allowed or denied.

Now let's look at a file with an ACL.

```
# ls -l file2
-rwxrw-r—+ 1 mwl mwl 0 Nov 16 13:47 file2
```

Note that the standard Unix-style permissions are identical to the first file we looked at. The permissions end with a plus (`+`) sign, however. This indicates extended permissions, or an ACL. Let's look at these permissions. I'm adding the `–q` flag, to eliminate the commented-out information at the top.

```
# getfacl –q file2
          user:jkh:rwxp--aARWcCos:-------:allow
            owner@:rwxp--aARWcCos:-------:allow
            group@:rw-p--a-R-c--s:-------:allow
         everyone@:r-----a-R-c--s:-------:allow
```

These permissions are very similar to the ones for *`file1`*, but the ACL has an extra Access Control Entry. The first non-commented line has five fields. What appears as the first field in the later entries is two fields here.

In addition to the `owner@`, `group@`, and `everyone@` tags that map to traditional Unix permissions, the tag can specify a user or a group. These tags are followed by a *qualifier*, narrowing down exactly who this ACE applies to. The first entry in this ACL applies to the user **`jkh`**[1].

Reading this ACL, the user **`jkh`** has exactly the same permissions on this file as the file owner. We thus have four groups of permissions on the file, something normal Unix permissions don't permit.

The user identification part of an ACE—either the tag derived from Unix-style permissions, or the combination of a tag and a qualifier—is called a *principal*. Every ACE includes a principal, permissions, inheritance, and type.

ACL Permissions

What do all of those permissions letters mean, anyway? Let's dismantle them and find out.

Each permission can be represented by either a single character (such as `x` or `c`) or a name (such as `execute` or `read_acl`). Remembering what all these letters mean is annoying, though. If you use the `-v` flag to getfacl(8), you'll get the name of each privilege rather than the letter code.

```
# getfacl -qv file2

user:jkh:read_data/write_data/execute/append_data/read_attributes/write_attributes/read_xattr/write_x
…
```

Even the most cryptic names are slightly easier to remember than single letters.

You can use the long names in commands, rather than the separate letters. Separate long names by slashes, as shown above. The single-character privileges don't need any separators.

Here are the privileges grouped by general functions: reading, writing, deletion, and others. We'll also discuss privilege groups.

Read Permissions

These permissions control a user's ability to access information in a file.

The `read_data` (`r`) permission controls access to the file's contents, exactly like the standard Unix-style read permission. This permission usually, but not necessarily, comes with access to all the other read privileges.

The `read_attributes` (`a`) permission determines access to Unix-level file metadata, such as the inodes and size, rather than the file contents. If a principal lacks this permission, he can see that the file exists by using ls(1), but requesting details with `ls -l` generates an error. The user can see that the file exists, but nothing about the file. If the user has the `read_data` permission but lacks `read_attributes`, he can cat(1) the file, but not view the file's size, timestamp, ACL, and so on.

The `read_xattr` (`R`) permission is ignored on FreeBSD and Solaris. Theoretically, it lets the principal read extended attributes other than those used for ACLs. While UFS and ZFS do have extended attributes for storing special data, most sysadmins never need to access them.

The `read_acl` (`c`) privilege permits reading the special subset of extended attributes that contain ACL data. If a user has `read_data` but doesn't have `read_acl`, running ls(1) on the file both shows the file and gives an error. The `ls` command can't check the ACL to see if it should print the `+` sign after the permissions. Perversely, the presence of the error is a giveaway that the file *does* have additional attributes that the user is not allowed to see.

Write Permissions

These permissions control a user's ability to change a file.

The `write_data` (`w`) privilege is identical to the standard Unix-style write permission. The principal can change the file. This privilege usually, but not necessarily, comes with all of the other write permissions as well.

The `append_data` (`p`) privilege is intended to allow the principal to add data to the end of the file, but not otherwise change it. Both OpenSolaris and FreeBSD ignore this flag on files, and check for `write_data` instead. For directories, though, `append_data` lets the principal create a subdirectory. For this reason, it's a good idea to always assign `append_data` with `write_data`.

The `write_attributes` (`A`) permission lets the principal arbitrarily change the access, modification, and creation timestamps on a file. A user who can write to a file could programmatically change all of the timestamps on a file to, say, 23 February 1972, without otherwise accessing the file. Lacking this privilege doesn't mean that these times don't change when the user accesses the file; rather, it means that the user can't change them by means other than accessing, modifying, or creating the file.

The `write_xattr` (`W`) privilege is ignored on FreeBSD and Solaris. (In theory, it would allow the principal to write extended attributes on the file, other than ACLs.)

The `write_acl` (`C`) privilege allows the principal to edit the ACLs on a file.

The `write_owner` (`o`) privilege permits changing ownership on the file. Without this privilege, chown(8) and chgrp(1) will not work. It's disabled for normal users, for security reasons. You don't want users giving executable scripts to **`root`**.

Deletion Permissions

You can use ACLs to remove the ability to delete a file, through the **`delete`** and **`delete_child`** privileges.

The `delete_child` (`D`) privilege controls a principal's ability to delete files within a directory. By denying this privilege, you make the files within a directory non-deletable.

The `delete` (`d`) privilege controls a principal's ability to remove a file. Normally, this comes along with the ability to write to the file, and so it's not generally granted. Use `delete` when you want to override `delete_child` on a specific file in a directory.

So, a directory can declare all of the files in it irremovable, except for files specifically marked removable. Any user that can change the ACL on the file can add the `delete` privilege, so you'll need to block that access as well if you want files to be unmovable.

This sort of thing is why so many sysadmins drink.

Other Permissions

These two permissions don't really fit in anywhere else.

The `execute` (`x`) permission lets the user run the file as a program or open a directory. It's the same as assigning the execute bit with chmod(8). I would still encourage you to manage access to a program with groups rather than individuals, but using an ACL will let you give multiple groups execute permission. Just as with chmod(8), denying execute permission on an interpreted script (such as a shell or Perl script) doesn't prevent the user from running the script by calling it as an argument to the interpreter. You can disallow someone from running */home/mwl/script.sh*, but if they can read the file, they can run */bin/sh /home/mwl/script.sh*.

The `synchronize` (`s`) permission is ignored on FreeBSD and OpenSolaris-derived systems. It lets a file be accessed synchronously at the server and at the client simultaneously, which happens anyway as part of the filesystem.

Permission Sets

Some of these permissions make sense to be granted or revoked en masse. If a user should be able to write to a file, you'll want him to have all the privileges needed. Reading a file without errors requires the ability to see the file metadata. That's where the permission sets come in. Permission sets have no single-character representations.

The `full_set` includes all permissions. The principal is allowed or denied access to everything.

The `modify_set` grants the principal everything except `write_acl` and `write_owner`. He cannot change file ownership or the ACL on the file, but he can modify it any other way.

The `read_set` gives the principal the ability to read the file, its metadata, its ACL, and any extended attributes.

Finally, `write_set` lets the user write to the file, its metadata, and its extended attributes, but not its ACL.

Implicit ACEs for Owner

The file owner always gets the rights `read_acl`, `write_acl`, `read_attributes`, and `write_attributes`, even if they don't appear in any ACE in the ACL. Users cannot lock themselves out of their own files—that takes a sysadmin.

Setting ACLs

Now that you have some idea what these ACL privileges mean, let’s apply some. Use setfacl(8) to edit ACLs.

ACE order is crucial in interpreting ACLs, as discussed in “ACL Ordering and Deny” later this chapter. Many changes require that you pick a spot to insert a new ACE, or pick the number of an ACE you want to remove. ACEs in an ACL are numbered starting with 0, just like chapters in this book.

Adding ACEs

Use the `-a` flag to add an Access Control Entry to an ACL. This flag takes two arguments: the rule number where you want this ACE inserted and the ACE itself. You can give multiple ACEs, separated by commas. Multiple ACEs will be inserted in the order you give. Then give the file to be changed.

Here’s the original ACL for our example file.

```
# getfacl -q file2
          user:jkh:rwxp--aARWcCos:-------:allow
            owner@:rwxp--aARWcCos:-------:allow
            group@:r-----a-R-c--s:-------:allow
         everyone@:r-----a-R-c--s:-------:allow
```

The ACE for the user **`jkh`** is identical to that of the file owner.

I need to give the user **`phk`**[2] the same privileges as the user **`jkh`**, so I copy his ACE and change the principal, removing the dashes because I can. When adding ACEs, you can abbreviate user as `u`. I want to insert this ACE at line 1.

```
# setfacl -a 1 u:phk:rwxpaARWcCso::allow file2
```

This file now has the following ACL.

```
          user:jkh:rwxp--aARWcCos:-------:allow
          user:phk:rwxp--aARWcCos:-------:allow
            owner@:rwxp--aARWcCos:-------:allow
            group@:r-----a-R-c--s:-------:allow
         everyone@:r-----a-R-c--s:-------:allow
```

Users **`phk`** and **`jkh`** have the exact same privileges, which will probably annoy both of them.

Managing ACLs on a per-user basis is poor practice. Ideally, we’d have a system group for the role fulfilled by **`phk`** and **`jkh`**. Otherwise, your ACLs get very large very quickly. Any time a person changes job roles, you need to change the ACLs on however many files that user had access to. Managing everything with groups, even if the group starts with only one person in it, is far more sustainable and much less likely to drive the sysadmin bonkers. When adding an ACE, you can abbreviate group as `g`. Here I've created a system group for **`phk`** and **`jkh`**, and am adding a group ACE for their access. Rather than using the whole list of permissions, I'm using the privilege set `full_set`.

```
# setfacl -a 1 g:vermin:full_set::allow file2
```

Now that their privileges are expressed via a group ACE, I need to remove their individual ACEs.

Removing ACEs

Use the `–x` flag to remove an ACE from an ACL. You either need the ACE itself, or the rule number of the ACE. Remember, ACEs start numbering at 0. Here's our sample ACL now.

```
# getfacl –q file2
          user:jkh:rwxp--aARWcCos:-------:allow
      group:vermin:rwxp--aARWcCos:-------:allow
          user:phk:rwxp--aARWcCos:-------:allow
            owner@:rwxp--aARWcCos:-------:allow
            group@:r-----a-R-c--s:-------:allow
         everyone@:r-----a-R-c--s:-------:allow
```

To delete a rule by the ACE, give the ACE as an argument to `–x`. You can delete any dashes, and you can abbreviate `user` as `u`. Here I fry the access for user **`jkh`** by specifying his ACE.

```
# setfacl -x u:jkh:rwxpaARWcCos::allow file2
```

This ACE is no longer in the ACL.

I find it much easier to remove ACEs by number, rather than typing in the whole ACE at the command line. In the list above, the rule allowing **`phk`** access is rule 2. I removed an earlier rule, however, shifting all the rules up by one space. ACE 2 is now the one that gives the file owner access. Removing it will annoy the file owner. Always double-check the ACL before removing rules by number.

```
# setfacl -x 1 file2
```

User **`phk`** no longer has a personal ACE.

To completely wipe out the ACL, burning it to the ground until only traditional Unix privileges remain, use `setfacl –b`.

```
# setfacl -b file2
```

You can now start over and try to do it right this time.

You can also use the `-m` flag to edit an ACL. The `-m` flag automatically puts the new ACEs at the beginning of the list and modifies existing ACEs to match the ACE on the command line. Using `-m` is discouraged, as `-a` and `-x` are much more precise. It's not that the effect of `-m` is random, rather that the effect of `-m` *feels* random for anyone not intimately familiar with its innards.

ACL Files

You can apply an ACL from a file, or copy the ACL on a file to another file. Storing an ACL in a file is as simple as running getfacl(8) and redirecting the output to a file. It might make sense to add the `-q` flag, eliminating the commented information.

```
# getfacl -q file2 > acl1
```

You can then apply this ACL to another file by using the `-M` option to setfacl(8). Use the file containing the ACL as an argument to `-M`. This adds any new entries to the beginning of the target file's ACL. Here, I apply the ACL in the file *acl1* to *file4*.

```
# setfacl -M acl1 file4
```

This would allow you to set a common ACL on every file in a directory, but using inheritance for managing directories is much easier, as discussed later this chapter.

Adding ACEs to an ACL gets tricky when a file already has an ACL with some of those same principals. The easiest thing to do is remove the old ACL before applying the new one, using the `-b` flag before the `-M`.

```
# setfacl -b -M acl1 file4
```

If the `-b` appears after `-M`, `setfacl` applies the new ACL and immediately blows it away. Just skip `-M` if that's what you want to do.

But what if you don't want to blow away the old ACL, but rather merge the two? Here's the ACL on a file.

```
    group:vermin:rwxp--aARWcCos:-------:allow
          owner@:rwxp--aARWcCos:-------:allow
          group@:r-----a-R-c--s:-------:allow
       everyone@:r-----a-R-c--s:-------:allow
```

The group **vermin** has complete access to this file. But here's an ACL I want to add.

```
          owner@:rwxp--aARWcCos:-------:allow
    group:vermin:r-x---a-R-c--s:-------:allow
```

```
       group:wheel:rw-p--aARWcCos:-------:allow
    group:operator:r-xp--a-R-c---:-------:allow
            group@:r-----a-R-c--s:-------:allow
         everyone@:r-----a-R-c--s:-------:allow
```

This ACL gives access to new principals, the `wheel` and `operator` groups. But it has conflicting rules for the `vermin` group. The file's original ACL gives `vermin` full access, while in the new ACL `vermin` has only read and execute access. Let's apply the ACL and see what happens.

```
# setfacl -M acl2 file4
# getfacl –q file4
       group:wheel:rw-p--aARWcCos:-------:allow
    group:operator:r-xp--a-R-c---:-------:allow
      group:vermin:r-x---a-R-c--s:-------:allow
            owner@:rwxp--aARWcCos:-------:allow
            group@:r-----a-R-c--s:-------:allow
         everyone@:r-----a-R-c--s:-------:allow
```

The new ACEs granting `wheel` and `operator` access appear at the front of the ACL, even though they appeared in the middle of the ACL file. The ACE for `vermin` has been overwritten with the ACE from the file.

NFSv4 ACLs really want the owner, group owner, and everyone permissions to appear at the end of the ACL. You can put these rules in your ACL-in-a-file anywhere you like, but when you apply the ACL to a file they'll be automatically shifted to the end. Remember, NFSv4 ACLs work on a per-privilege first-match basis. The old-fangled Unix-style permissions are the last stop before the implicit "nope" at the end.

What's more, managing ACLs on a file-by-file basis is also a last resort. You want files to inherit their ACLs.

ACL Inheritance

ACL management is normally performed on a per-directory basis. You set an ACL on a directory, and tell it that all files and directories within that directory inherit the ACL. When you need to change the ACL, change it at the directory and it automatically percolates down through the filesystem.

Strictly speaking, inheritance is a feature standardized in NFS 4.1, while the rest of the ACL implementation comes from the NFS 4 specification. Inheritance is compatible with NFS 4, however, and it's an important feature.

Inheritance Styles

NFSv4 ACLs have two different primary inheritance models, and a few less widely used ones. We'll talk about the popular ones separately, then dive into the lesser-known ones.

The default inheritance model combines the user's umask with the inherited ACL to set permissions and privileges on newly created files and directories. The owner loses the `write_owner` and `write_acl` privileges on any inherited ACL, meaning that they can't change the file permissions with chmod(8). This retains much of the traditional Unix model, where the user adjusts their umask to set permissions. Microsoft clients have no concept of a umask, and CIFS servers like Samba can't make reasonable guesses on what umask it should use for all the different files and directories it creates. Using this model completely confuses users.

In addition to the default model, ZFS supports a *passthrough* mode. This ignores the user's umask and simply propagates a directory's ACL down to files and subdirectories. If you're serving Windows clients, you almost certainly want passthrough mode. To enable passthrough mode, set the ZFS properties **`aclmode`** and **`aclinherit`** to `passthrough`.

```
# zfs set aclmode=passthrough zroot
# zfs set aclinherit=passthrough zroot
```

Change these settings before creating any files or directories, especially for Windows clients.

Inheritance Options

You can adjust a ZFS dataset's **`aclinherit`** property to dictate how inheritance functions. While passthrough mode is the most popular, other options might fit special situations.

The default, `restricted`, strips any inherited `write_owner` and `write_acl` permissions on new files and directories. Only the file owner can change the ACL of their own files. The inherited ACL cannot assign privileges beyond those permitted by the user's umask.

To disable all ACL inheritance on newly created files and directories, set **`aclinherit`** to `discard`.

If we set **`aclinherit`** to `noallow`, new files and directories inherit only `deny` ACEs. An outside process must assign any `allow` ACEs.

Setting **`aclinherit`** to `passthrough` ignores umask, creating files with Unix-style permissions based on the ACLs.

Lastly, an **`aclinherit`** value of `passthrough-x` restricts execute permissions on new files. If an inheritable ACE includes execute permission, and the umask permits creating executable files, new files inherit the executable permission.

Mode Options

The **`aclmode`** property dictates how chmod(8) can change the file's permissions.

An **`aclmode`** setting of `groupmask` makes the user's umask the maximum permissions level a file can have. See "Umask and ACL Inheritance" later this chapter for a full discussion.

Having **`aclmode`** set to `discard` lets chmod(8) remove all ACL information.

An **`aclmode`** of `passthrough` sends the ACL straight through to new files.

Inheritance Flags

You can dictate exactly how a directory propagates its ACL by using *inheritance flags*. You remember that next-to-last field in each ACE that's always been dashes? Here's where we fill that in. These flags can only be placed on directories, not on individual files. As with privilege flags, you cannot mix names and letters.

The `file_inherit` (`f`) flag means that all files within the directory inherit the ACL on the directory.

The `dir_inherit` (`d`) flag means that the directory's subdirectories inherit the directory's ACLs.

The `inherit_only` (`i`) flag means that newly created directories and files within the directory get this, but this ACL doesn't apply to this directory. This gets applied if, say, the files in the directory have loose permissions but you don't want anyone to delete the directory itself.

The `no_propagate` (`n`) flag tells the directory to add its ACL to child directories, but not their children.

While only directories can have these inheritance flags, you will see one inheritance flag on files. The `inherited` (`I`) flag means that the ACL on this file or directory was inherited. This flag is only supported in FreeBSD 11 and later.

Files and directories inherit their ACLs when they're added to the filesystem. On a file server, this might be when a user uploads a document. For local users, it's at file creation time.

Directories and Inheritance

Let's apply an ACL to a directory and add some files. We'll start with the common case, a ZFS dataset with ACLs in passthrough mode. Note that while I use the full name of a privilege set, I use the letters for the inheritance options. I can't mix full names of privileges with letter privileges, or the full names of inheritance options with letters, but I can use different methods for privilege and inheritance.

```
# mkdir support
# setfacl -a0 g:vermin:full_set:fd:allow support/
# getfacl –q support
      group:vermin:rwxpDdaARWcCos:fd-----:allow
            owner@:rwxp--aARWcCos:-------:allow
            group@:r-x---a-R-c--s:-------:allow
         everyone@:r-x---a-R-c--s:-------:allow
```

My system group **`vermin`** has full access to this directory. The `f` (`file_inherit`) and `d` (`dir_inherit`) inheritance flags tell the system to apply this same ACL to any files and subdirectories.

User **`phk`** uploads a file, *`phones.docx`*, containing instructions on how to set up new phones. He doesn't do anything special to the file. Get its ACL.

```
# getfacl -q phones.docx
      group:vermin:rwxpDdaARWcCos:------I:allow
            owner@:rw-p--aARWcCos:-------:allow
            group@:r-----a-R-c--s:-------:allow
         everyone@:r-----a-R-c--s:-------:allow
```

The ACL on this file differs from that on the directory only by the inheritance flags. The `I` means that this ACL is inherited.

If a user in this group creates a subdirectory, it will get the same ACL, plus the `f` and `d` inheritance flags.

```
# mkdir serverroom
# getfacl -q serverroom/
      group:vermin:rwxpDdaARWcCos:fd----I:allow
            owner@:rwxp--aARWcCos:-------:allow
            group@:r-x---a-R-c--s:-------:allow
         everyone@:r-x---a-R-c--s:-------:allow
```

Files within this new subdirectory will get the same ACL. User **jkh** uploads the document *alarms.docx* to the *serverroom* directory, recording what to do the next time he sets off the alarm. It gets the same ACL as the file above it.

Changing ACLs and Inheritance

ACL inheritance kicks in at file and directory creation time. Let's see how that affects the system in practice. Here I change the inheritance so that ACLs do not propagate to files within subdirectories, by adding the `n` inheritance flag. I also give everyone in the **staff** group read-only access.

```
# setfacl -a0 g:vermin:full_set:fdn:allow support
# setfacl -x 1 support/
# setfacl -a0 g:staff:read_set:fdn:allow support
```

Now check an existing file in the support directory, such as *phones.docx*, and you'll see the ACL is unchanged. The **staff** group can't read this file. Create a new file like *keycards.docx*, though, and it has the new ACL.

```
# getfacl -q keycards.docx
       group:staff:r-----a-R-c---:------I:allow
      group:vermin:rwxpDdaARWcCos:------I:allow
            owner@:rw-p--aARWcCos:-------:allow
            group@:r-----a-R-c--s:-------:allow
         everyone@:r-----a-R-c--s:-------:allow
```

That seems simple enough, but subdirectories are slightly less intuitive. I turned off the "propagate to files in subdirectories" inheritance option, so let's go to the subdirectory *serverroom* and see what happens there. The existing file, *alarms.docx*, has the same ACL as when we created it.

Create a new file in the *serverroom* directory, *racks.docx*. Check its ACL, and you'll find it has the same ACL as the old file *alarms.docx*. Those ACLs aren't supposed to propagate any more—what happened?

The top level directory had an ACL change that said to not propagate ACLs to files in subdirectories, but the subdirectory *serverroom* already existed. Directories inherit their ACLs at creation time. The change to the top-level directory did not propagate to the subdirectory, so the subdirectory retains its own ACL.

If you create a new subdirectory, it will inherit the new ACL from the parent directory. Files in that new subdirectory will not inherit the ACL.

This is no different than you'd experience on a Microsoft system. Changing the ACL in a directory and having it propagate to the entire directory tree rooted there causes a lot of disk churn as the system touches every single file. Recursively changing the ACLs on a whole ZFS directory tree probably requires two find(1) commands: one for the files, and a separate one for the directories. You're best off reading the ACL from a file with `-M`, and removing the existing ACL with `-b` before applying the new one.

Umask and ACL Inheritance

If you're using ACLs on UFS, or you don't want to use ZFS' passthrough mode, the user's umask gives the upper limit on permissions in an ACL. If you're not familiar with umask and permissions at file creation time, go read umask(2) and any number of tutorials on the Internet. Every systems administrator must understand umask before even beginning to ponder thinking about implementing ACLs.

The default FreeBSD umask is 022, meaning that files get created with a mode that the owner can read and write them, while the group owner and everyone else can only read them. If you're using this umask and create a file, those are the *maximum* permissions the file can have. If your inherited ACL says "this other group should have read and write access to the file," and your umask says "everybody else gets read-only access to a file I create," the umask wins.

FreeBSD gives users a default umask of 022 for very good reasons. In some cases it might make sense to increase the umask—on some shared systems I set my umask to 027, so that random system users can't view the contents of my files. It rarely makes sense to *lower* your umask.

If you don't use ZFS's ACL passthrough features, you'll need to reduce your umask to 0 when working within an ACL-protected part of the directory tree, then increase it back to 022 when you're done. I can't be bothered to remember to do that, not when passthrough mode handles it for me.

ACE Ordering and Deny

NFSv4 ACLs are processed on a first-match basis. A match doesn't trigger the whole ACE, however. Rather, each permission gets checked separately. Consider the following file.

```
# touch daft
# chmod 157 daft
# ls -l daft
–xr-xrwx 1 mwl mwl 0 Nov 19 14:55 daft
```

This really is a daft file. It's executable by the owner. It's readable and executable by the group owner. And it's writable, readable, and executable by everyone except the owners. You would almost[3] never do this in the real world.

```
# getfacl –v daft
            owner@:rw-p----------:-------:deny
            group@:-w-p----------:-------:deny
            owner@:--x---aARWcCos:-------:allow
            group@:r-x---a-R-c--s:-------:allow
         everyone@:rwxp--a-R-c--s:-------:allow
```

This is the first time I've shown a `deny` ACE in an ACL, so let's touch on that quickly.

Generally speaking, `deny` ACEs should appear at the beginning of the ACL. Microsoft clients in particular throw a hissy fit when `deny` statements appear intermixed with allow statements. The only `deny` ACE that commonly appears after an `allow` is the implicit one at the very end. Some ACLs don't permit putting all of the deny statements at the beginning of the file, though. Run `chmod 00101` on a file and look at the ACL.

When a principal tries to access a file, each privilege it tries to use gets checked separately. Each check "drops down" through the ACL until it hits a match of principal and privilege.

Suppose the file owner wants to execute the *`daft`* file. The principal on the first ACE is the owner. This ACE specifically forbids reading and writing, but says nothing about execution. We check the next ACE, and the next, until an ACE permits execution, an ACE explicitly denies execution, or we hit the terminal implied `deny`.

This example looks trivial, but the same theory applies to complex ACLs. A user might get access to one privilege as a member of one principal, then a second privilege as a member of another principal. Each individual privilege check falls through the ACL until it is either granted to the user or blocked.

ACLs and Samba

NFSv4 ACLs are most commonly deployed on Samba 4 servers (http://www.samba.org), as storage back ends for CIFS clients. Here are a few things that commonly trip up these sysadmins, courtesy of John Hixson[4], the FreeNAS support guy who answers most ACL questions.

Always use ZFS. Once you have ZFS, always set **`aclmode`** and **`aclinherit`** to `passthrough` on datasets for Samba.

Do not use world-writable files (mode 777), or use a umask of 000. If you do, you *will* suffer.

Windows Search on directories requires `allow` on at least the `rxaRc` privileges.

ACLs can get far more complicated than the examples in this chapter. Remember, simplicity is the key to successful ACL deployment. Good luck.

After ACLs, FUSE and autofs will seem positively trivial.

---

[1] All usernames are randomly generated. Any resemblance to primordial FreeBSD core team members, living or dead, is purely coincidental.

[2] Still a coincidence.

[3] I would say “absolutely never,” but then some ~~annoying~~ clever reader would tell me exactly why they used this.

[4] Hixson would also like it known that no user named **`jkh`** should ever get access to anything, ever.

Chapter 12: Filesystem Glues

This last chapter covers a couple different tools that aren’t exactly filesystems, but simplify working with filesystems: FUSE and autofs(5).

FUSE, or Filesystem in Userspace, lets users manage filesystems, mount filesystems not supported by the FreeBSD kernel, and transform their own data to be accessible in a filesystem-like manner.

The automounter filesystem autofs(5) can automatically mount filesystems for you, either via NFS or when physically attached to the system.

We’ll start with FUSE, and proceed to autofs.

FUSE

Filesystem in Userspace, or FUSE, presents a way for userland programs to handle and process unusual filesystems. FUSE programs run as user processes, interpreting data and handing the kernel predigested memory structures so the kernel can present the filesystem to the system. This allows a userland program to provide filesystem support to the kernel.

A filesystem can be more than just a place to stick data. Filesystems can also be useful tools for accessing data in novel, unexpected ways. Consider devfs (Chapter 2), which FreeBSD uses to let programs access hardware through a filesystem. Optional tools like fdescfs and mqueuefs (Chapter 3) let you access kernel memory structures as a filesystem. But many users could benefit from exposing their own data in ways you can't even guess at.

FUSE also supports interesting data transformations, such as mounting a filesystem over SSH. The FreeBSD developers would never permit an SSH client in the kernel, but since we're out in userland anyway, you might as well fire up SSH while you're there. Or an MP3 transcoder. Or an HTTP client, letting you mount and examine an ISO on a web server without downloading the whole thing. There's even a Wikipedia FUSE module, letting you view and edit Wikipedia as a filesystem. Some filesystems, such as MooseFS (www.moosefs.org) are implemented entirely for FUSE. FreeBSD has over three dozen FUSE modules as I write this, with more appearing regularly.

FUSE modules are frequently portable between operating systems. The data structures of a Microsoft NTFS disk don't change based on the operating system you're reading it on. FUSE modules do need changes to work between operating systems—FreeBSD expects different filesystem memory structures than Linux or Solaris—but that's trivial compared to the difficulty of implementing a filesystem inside the kernel.

Using FUSE is not risk-free. FUSE modules feed data into the kernel. While the kernel sanitizes the data, a corrupt filesystem might throw garbage into the kernel. A clever intruder might be able to leverage his way into privileged access.

Sometimes, though, FUSE is the best way to solve a problem. You need to investigate files on another server? Use SSH. A program needs to investigate files on another server, but it can only look at local files? Stick FUSE's mount-over-SSH module in the middle and go on.

FUSE Prerequisites

FreeBSD implements its FUSE support as a kernel module, `fuse.ko`. Load it with kldload(8), or at boot-time in */boot/loader.conf*.

Using FUSE requires access to the */dev/fuse* device. By default, only **`root`** and users in the **`operator`** group get access. Assigning users with FUSE privileges to **`operator`** is reasonable on some systems. If that doesn't suit your environment, create a special group for FUSE and use a devd(8) rule to assign */dev/fuse* to that group owner. If you want unprivileged users to have access to FUSE, you'll need to set the vfs.usermount sysctl to `1`.

We'll use the FUSE SSH module as an example. Once you understand how to use sshfs(1), deploying other modules should be a matter of understanding the module's target —that is, to deploy the FUSE NTFS module, you'll need to dive into NTFS. After covering `sshfs` we'll delve into details of FreeBSD's mount_fusefs(8), which lets you fine-tune how FreeBSD treats FUSE filesystems.

SSHFS

The `fusefs-sshfs` FUSE module lets you mount a remote directory over SSH. This lets you treat a remote filesystem as local, and run commands on files stored on the remote host. Once you've installed the package, use sshfs(1). You'll need two arguments: the remote host, and the directory on which you want the SSH filesystem mounted.

I strongly encourage use of public key authentication with SSH. If you're still using password-based authentication with SSH, check out any number of online tutorials or my own *SSH Mastery* (Tilted Windmill Press, 2012).

Here I use sshfs(1) as a regular user, mounting my home directory on the server **`mail`** on a directory I own, *`$HOME/mnt`*.

```
$ sshfs mail: mnt/
```

If I go into the *`mnt`* directory I'll se the files from my home directory on the server. I can move, copy, and rename these files as I wish.

The sshfs module deliberately shares much syntax with the SSH client. To use a different username, put it before the hostname, separated by an "at" (`@`) sign. If you want to mount a different directory, put that directory after the colon. Change the port with `-p` and the port number.

```
$ sshfs -p 2222 mail:/tmp mnt/
```

To unmount the sshfs, use umount(8) as normal.

```
$ umount /home/mwl/mnt
```

If you have weird behavior, add the `-d` to enable FUSE debugging. The `sshfs` command won't detach from the terminal, instead showing only debugging information as you use the filesystem.

```
$ sshfs –d mail: mnt/
```

See sshfs(1) for many more options, most of which most of us won't need. Most of these exist as arguments to `-o`, exactly as in ssh(1).

Permissions and Execution

FUSE filesystems get treated slightly differently than other filesystems.

Only the user who mounted a FUSE filesystem can access it. While **`root`** can see that a user has mounted a FUSE filesystem, it can't see the contents of that filesystems. Attempts to `cd` into that directory will be met with "Operation not permitted."

Also, file ownership and privileges on the FUSE-mounted filesystem might not directly match to the permissions on the local filesystem. I can mount a remote filesystem with sshfs, but sshfs doesn't coordinate user and group IDs between my workstation and the server. I can manipulate those remote files exactly as if I was logged in at a command line. My workstation might support NFSv4 ACLs, but files mounted via sshfs use whatever permission scheme the SSH server uses.

Finally, just because an sshfs mount has files that are executable, don't assume that those programs will actually function on your host. You'll need the correct shared libraries, and any other resources the program requires. I find that even many of my simple shell scripts don't work the way I expected when run on a sshfs mount, as I wrote them assuming they'd only be executed on the server.

Sharing FUSE Filesystems

FreeBSD's FUSE allows only the user who mounted the filesystem to access it. The **`root`** user can override this, making a FUSE filesystem available to everyone. Most FUSE modules have an option to request this behavior. For `sshfs`, it's the `allow_other` option. (You'll also see an `allow_root` option, but that has no effect on FreeBSD.) Here I specifically mount an SSH FUSE filesystem so that anyone can access it.

```
# sshfs -o allow_other mwl@mail: /mnt
```

This can only be run by **`root`**. Using `allow_other` as a non-root user triggers an error.

Because of the nature of FUSE, this mount point lacks all permissions protections. Any user who can access */mnt* gains complete access to the SSH session underlying the filesystem. Carefully restrict access to this directory!

FUSE opens nearly limitless storage options. One of them might fix your intractable problem. For more details on FreeBSD's FUSE implementation and management, read mount_fusefs(8).

Automounting with autofs

FreeBSD 10 and later include a new automounter service, autofs(5). It replaces the more complicated automounter amd(8) in older versions. Autofs automatically identifies and mounts filesystems for users, even if they don't have permissions to mount anything. If you plug in a flash drive, FreeBSD can examine it, identify the filesystems on it, and mount them, all without human intervention. When a user tries to access an NFS share, autofs mounts it if the server permits. Autofs automatically handles removable media, NFS shares and filesystems configured in */etc/fstab* with the `noauto` flag, and can be configured to handle almost anything FreeBSD can mount. While many sysadmins won't want servers to automount removable media, they might find automounting NFS shares very useful.

In previous versions of FreeBSD, the sysadmin could allow unprivileged users to mount removable filesystems by setting the sysctl vfs.usermount to `1`. Users could then mount media on directories that they owned. While many sysadmins feel lucky if more than half of their users can remember to log out when they walk away from the terminal, mounting required that users remember how to run mount(8). For most users, this wasn't going to happen. Automounting gives the sysadmin better control over what media their users can mount, and where it gets mounted.

FreeBSD's autofs was deliberately designed to be compatible with the Solaris automounter. The underlying code is quite different, but the practice should be the same. Any tasks that feel over-engineered as well as any particularly distressing behaviors can be blamed on Oracle.[1] If you have automounter problems, you might check Solaris documentation as well as that for FreeBSD. Sun Microsystems licensed their automounter to just about every commercial Unix vendor, so your existing automounter experience should apply.

Enabling autofs

Enable autofs in */etc/rc.conf*.

```
# sysrc autofs_enable=YES
```

The default autofs configuration expects to mount removable media in */media* and NFS shares in */net*. (Automounting of removable media is disabled by default, but we'll cover that shortly.) While */media* exists in default FreeBSD installs, */net* does not. Create the */net* directory before enabling autofs. The directory's absence won't prevent autofs from starting, but it will prevent you from automounting NFS shares.

autofs Components

You won’t find an autofs startup script. FreeBSD manages each of the three major components separately, all controlled by the `autofs_enable` setting.

The automount(8) command is used to manage automounted filesystems. The */etc/rc.d/automount* startup script loads the autofs kernel module *autofs.ko*.

The automountd(8) daemon handles mount requests. When you go to a directory used by autofs, autofs checks for filesystems of that type. Visit */media*, and `automountd` checks for the presence of removable media. Once you try to access the media, `automountd` calls mount(8) or a variant thereof to perform the mount.

Automatically mounted filesystems can pile up. Autofs’ autounmountd(8) disconnects unused filesystems after a timeout. Yanking mounted removable media from the system can damage the media’s filesystem, and automatic unmounting reduces the risk of that happening. Unmounting unused NFS shares reduces load on your NFS server.

Most of these programs take their instructions from the autofs map file, */etc/auto_master* and the related map files in */etc/autofs*.

Configuring autofs

Autofs is configured through *maps*. A map correlates a device or partition with a method of mounting that entity.

FreeBSD includes several *special maps*. Special maps are generic maps for handling entire classes of media. FreeBSD includes special maps for removable media, NFS shares, noauto filesystems, and preventing autofs mounts. Special maps can handle nearly any automounting configuration you might have.

If you're one of the very few who can't use the special maps, you can create custom maps to automatically mount directories and media. The auto_master(5) man page gives the whole syntax for */etc/auto_master* and custom maps. In most (but not all) cases, using a custom map is an administrative choice rather than a technical necessity.

We'll focus our attention on the special maps.

auto_master

The file */etc/auto_master* points to the automounter maps. Each line is a single mapping, with two or three parts: the mount point, the name of the map, and if desired any options. The default configuration contains only one uncommented map, the NFS special map.

```
/net -hosts -nobrowse,nosuid,intr
```

This map controls the mount point */net*. It uses the map named `-hosts`. When something checks for directories under this mount point, autofs mounts the requested NFS share with the options `nosuid` and `intr`. The `nobrowse` option is specific to autofs, and prevents autofs from automatically creating subdirectories for all the hosts in */etc/hosts*.

Any time you edit *auto_master*, you must run `automount` to reconfigure the maps for the autofs support daemons.

Autofs attaches to all the mount points referenced in *auto_master*. Run `automount -L` to view all those directories.

```
# automount -L
/net -nobrowse,nosuid,intr -hosts
```

The */net* directory now has an autofs instance mounted on it, using the map `-hosts`. As BSD filesystems are stackable you *can* mount something else on */net*, but you shouldn't unless unnecessary confusion amuses you.

/etc/autofs

You can put maps directly in `/etc`, or use special maps in `/etc/autofs`. Special maps are scripts. Autofs runs the script when the mount point is accessed. Non-executable files are almost certainly custom maps created by the sysadmin. Autofs opens and parses these files like any other map.

All of the maps in the `/etc/autofs` directory shipped with FreeBSD are special maps.

Automounting with Special Maps

FreeBSD comes with four special maps: *-hosts*, *-media*, *-noauto*, and *-null*. (There's also a special map to read map information out of LDAP, but that's highly environment-dependent, so we won't cover it.) You know a map is classed as special because its name in */etc/auto_master* begins with a dash.

The *–hosts* special map, in */etc/autofs/special_hosts*, identifies the NFS shares available on a host. When you trigger an NFS automount request, the special map queries the NFS server to see what's available to you.

The *–media* special map in */etc/autofs/special_media* probes unmounted devices, like removable media, and helps mount them.

The *–noauto* special map, */etc/autofs/special_noauto*, checks */etc/fstab* for devices that are not automatically mounted at boot, and mounts them when you try to use them.

Finally, the *–null* special map */etc/autofs/special_null* ties up a mount point so that automountd(8) can't mount anything there.

Automounting NFS

Let's try this automounting thing with NFS. My test network has two NFS servers, **`mail`** and **`www`**. I know that my workstation can access shares on both because I've previously mounted them manually.

To automount from a server, look in a directory named after the server in */net*. The directory won't exist before you look for it, but look anyway. Right now, my workstation's */net* directory is empty. To see the shares available on the host mail, I `cd` or `ls` the */net/mail* directory.

```
# ls /net/mail
usr var
```

Nothing is actually mounted at this point: autofs only displays the available shares. On the plus side, directories for the available NFS shares now exist.

This mail server exports */var/log* via NFS. Once I try to examine */net/mail/var/log* on my workstation, autofs mounts the NFS share on that directory. I have whatever access the mail server is configured to permit, exactly as if my sysadmin had mounted this share for me.

If an NFS server disappears, mount requests can still hang. That's why autofs defaults to mounting NFS shares with the `intr` option, so a user can `CTRL-C` and get a terminal back without bugging the sysadmin.

Automounting Removable Media

The default configuration in */etc/auto_master* includes a commented-out line for automounting removable drives on */media*. It's not enabled by default; while */media* is intended for removable drives, many sysadmins use it for manual media mounts. Having the automounter suddenly monopolize */media* would violate the Principle of Least Astonishment.[2]

```
/media -media -nosuid
```

This rule reserves the */media* directory for the automounter. It applies the special map `-media`, and mounts everything with the `nosuid` option. Uncomment the line and run `automount` as **`root`** to activate it.

You'll also need a *devd.conf* entry to flush old automounter cache entries. In FreeBSD version 10.2 and later, this entry appears in */etc/devd.conf* but is commented out. Create a */usr/local/etc/devd/autofs.conf* that contains the following.

```
# Discard autofs caches, useful for the -media special map.
notify 100 {
  match "system" "GEOM";
  match "subsystem" "DEV";
  action "/usr/sbin/automount -c";
};
```

If you have removable media plugged into your machine, you'll suddenly get new directories in */media*.

```
# ls /media
10_2_RELEASE_AMD64_CD
da0
da0p1
```

Apparently this machine had an optical drive in it that I was unaware of. Some gremlin must have stuffed a FreeBSD 10.2-RELEASE CD therein. Autofs found the disk and mounted it by the filesystem label. If I go into */media/10_2_RELEASE_AMD64_CD*, autofs will mount the CD for me.

I don't have privileges to unmount the disk, but I can run `cdcontrol eject cd0` to have the drive spit the disk out. The automounted filesystem remains mounted until the next autounmountd(8) run (see "Auto-Unmounting" later this chapter).

If I later put another CD into the drive, though, it'll show up with its label. The old label remains, however.

```
# ls /media
10_2_RELEASE_AMD64_CD
USENIX06_TECH_SESSIONS
da0
```

While the CD drive now has the Usenix 2006 proceedings disc in it, the directory for the FreeBSD 10.2 disc remains. Not even **`root`** can remove the directory.

Automounting Noauto Filesystems

A *noauto* filesystem is one listed in */etc/fstab* with the `noauto` option. Noauto filesystems are preconfigured on a host. If the sysadmin has preconfigured the mount and enables the –noauto special map, unprivileged users can mount the device.

There's a commented-out entry for -noauto filesystems in */etc/auto_master*. Uncomment it and run `automount` to activate it.

```
/- -noauto
```

The -noauto special map has no mount options. The sysadmin presumably configures any mount options in */etc/fstab*. Sensible options for an NFS share don't really apply to a CD drive. Here are a couple of common entries that might appear in a filesystem table.

```
/dev/cd0 /cdrom cd9660 ro,noauto 0 0
mail:/usr/ports /usr/ports nfs rw,tcp,soft,intr,noauto 0 0
```

The first assigns the first CD drive the mount point of */cdrom*. The second NFS-mounts */usr/ports* from the host **`mail`** onto */usr/ports* on the local host. Neither gets mounted automatically at boot. When you run `automount`, autofs gets mounted on each of these mount points instead.

Autofs doesn't have to worry about what sort of filesystem is on the mount point, the mount options, or anything else. The sysadmin has done all of that. The only thing autofs needs to do is mount the partition when a user accesses it, and unmount it after it times out.

Null Automounts

The -noauto map can overlap with the other two maps. A user could access the same NFS share via */net/mail/usr/ports* or */usr/ports*. He could access a CD in */cdrom* or in a subdirectory of */media*. Also, some maps might allow overlapping access. So what?

So some user or program will make it a problem. Because that's what users and programs *do*. That’s what.

Use the –null special map to block these automount interactions. While blocking all the possible CD drive labels is impractical, blocking NFS mounts is straightforward.

My host uses the –noauto special map to mount */usr/ports* from the NFS server **mail**. I also have configured NFS automounting in */net*. I want to prevent my host from mounting the */usr/ports* share twice. Here’s a –null map to do exactly that.

```
/net/mail/usr/ports -null
```

Run `automount`, and you’ll get a new autofs instance mounted at */net/mail/usr/ports*. A user who goes there will find only an empty directory.[3]

Automounting with Regular Maps

Between the four types of special maps, there’s very little need to hand-craft automounter maps. If you don’t know how to write these maps, there’s little reason to acquire this rarely used skill.

If you are already a user of hand-crafted Solaris automounter maps, though, FreeBSD will accept your maps. FreeBSD can slip straight into your Solaris-based infrastructure, and it accepts the maps distributed by your configuration management system. You’ll have other issues integrating FreeBSD with Solaris, but the automounter isn’t one of them.

Debugging Automounting

Automounting uses two processes: the automount(8) command you run whenever you update */etc/auto_master*, and the automountd(8) daemon that watches the mount points for activity and performs the mounts. Both commands use the `-v` flag to increase verbosity.

The automounter daemon logs all the debugging information generated with `-v` to syslogd with the facility daemon.debug, so it won't appear in */var/log/messages*. You'll need a debugging log or the all-encompassing *all.log* demonstrated in */etc/syslogd.conf* to capture these messages. Alternately, you can stop the main automountd(8) service and run `automountd` in foreground debugging mode with `-d`. This puts all debugging information straight to the terminal.

```
# automountd -dv
```

You'll get some startup information, then helpful lines like this.

```
automountd: executing "mount -t cd9660 -o ro,noauto,automounted /dev/cd0 /cdrom/" as pid 1230
automountd: "mount -t cd9660 -o ro,noauto,automounted /dev/cd0 /cdrom/", pid 1230, terminated
gracefully
```

If automount requests fail, this is where the error messages appear.

Running automount(8) with `-v` shows more detail about its activity, including how it parses */etc/auto_master*. Try it sometime when automounting works, just to see what normal output looks like.

Auto-Unmounting

Automounting is helpful, but what about unmounting filesystems?

Unprivileged users cannot unmount automounted filesystems. The autounmountd(8) daemon watches automounted filesystems, however. It tries to unmount the filesystems every so often—by default, every 10 minutes.

If an automounted filesystem is in use, in any way, autounmountd(8) will not unmount it. Even an idle command prompt in the directory will block unmounting, exactly as if you were running umount(8).

You can adjust how often `autounmountd` tries to unmount the filesystem. The `-t` flag lets you set the number of seconds after automounting that `autounmountd` makes its first attempt to unmount, while `-r` sets the number of seconds to wait before retrying. This *rc.conf* setting tells `autounmountd` to try to unmount a filesystem three minutes after mounting, and retry every 60 seconds thereafter, because my users are whiny and impatient.

```
autounmountd_flags=”-t 150 –r 60”
```

You can increase autounmountd(8)’s verbosity with `-v`, and run it in the foreground for debugging with `-d`. Exactly like `automountd`, `autounmountd` uses the syslog facility daemon.debug, so you’ll need to configure a log file to capture that.

The systems administrator can unmount every automounted filesystem with `automount -u`.

```
# automount -u
```

This disconnects specifically automounted filesystems, but leaves the general automount(8) infrastructure in place and functioning.

You can shut down automounting, while leaving the daemons in place, by unmounting all autofs(5) filesystems with umount(8).

```
# umount -At autofs
```

Suddenly */net* is empty and `ls /media` does… nothing. Run `automount` to restore the autofs mounts.

Automounting can help ease the sysadmin’s job, and FUSE can make some difficult tasks merely annoying.

[1] Yes, automounting was originally created by Sun. But Oracle owns Sun, and blaming Oracle is never wrong.

[2] “would violate the Principle of Least Astonishment” is a polite but long-winded way to say “would really tick off the

sysadmin.”

[3] He’ll then call the helpdesk to complain, but that’s a separate issue best solved with carefully targeted “Authentic Oregon Trail”™ brand dysentery.

Afterword

While I'm a fan of several different BSD-based operating systems, FreeBSD's filesystems really are a killer feature that offers a flexibility unmatched by any other operating system. FreeBSD doesn't pat you on the head and tell you it knows how your storage should work. FreeBSD lets you make your own decisions, and configure storage in ways that the developers never even conceived of. The specialty filesystems herein, combined with tools like GEOM and ZFS, will let you make FreeBSD work almost anywhere, under almost any conditions, and address problems other operating systems can't approach.

Of all the books I've written, this has unexpectedly been among the most difficult. Exploring so many different filesystems, all with their own uses and sharp edges, has demanded that I fill my brain to overflowing and empty it many times.

I hope you find this book worthwhile.

If not, it's easily recyclable.