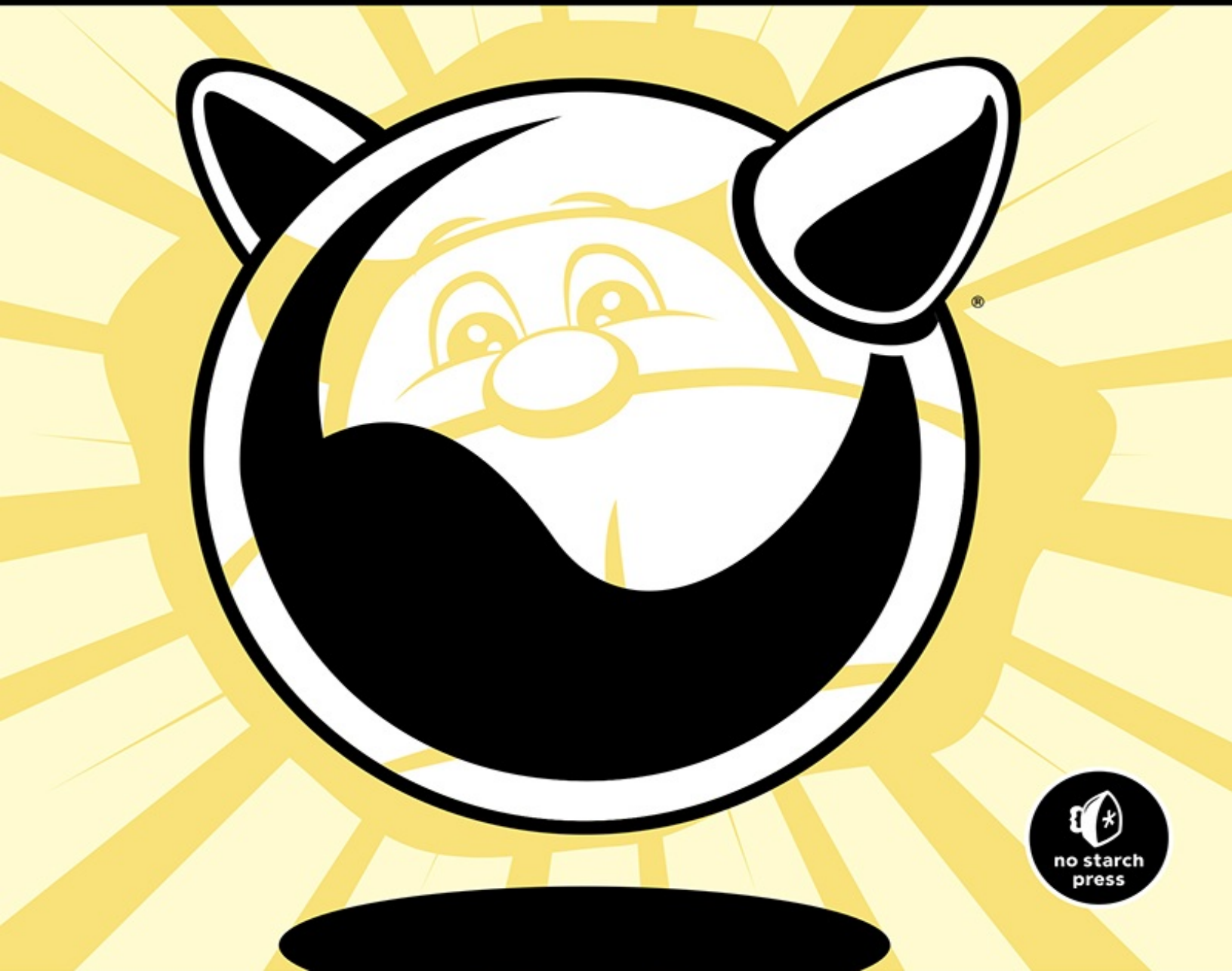
3RD EDITION

ABSOLUTE FREEBSD®

THE *COMPLETE* GUIDE TO *FREEBSD*

MICHAEL W. LUCAS



no starch press

PRAISE FOR *ABSOLUTE FREEBSD*

"Even longtime users of FreeBSD may be surprised at the power and features it can bring to bear as a server platform, and *Absolute BSD* is an excellent guide to harnessing that power."
—UNIXREVIEW.COM

" . . . provides beautifully written tutorials and reference material to help you make the most of the strengths of this OS."
—LINUXUSER & DEVELOPER MAGAZINE

" . . . packed with a lot of information."
—DAEMON NEWS

"When was the last time you could physically feel yourself getting smarter while reading a book? If you are a beginning to average FreeBSD user, *Absolute FreeBSD* . . . will deliver that sensation in spades."
—RICHARD BEJTLICH, TAO SECURITY

"By far the best FreeBSD book I have ever owned is *Absolute FreeBSD*, 2nd Edition by No Starch Press."
—BSD ZEALOT

"Master practitioner Lucas organizes features and functions to make sense in the development environment, and so provides aid and comfort to new users, novices, and those with significant experience alike."
—SCITECH BOOK NEWS

ABSOLUTE FREEBSD®

3RD EDITION

The Complete Guide to FreeBSD

by Michael W. Lucas

San Francisco

ABSOLUTE FREEBSD®, 3RD EDITION. Copyright © 2019 by Michael W. Lucas.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-10: 1-59327-892-6
ISBN-13: 978-1-59327-892-2

Publisher: William Pollock
Production Editor: Janelle Ludowise
Cover and Interior Design: Octopod Studios
Developmental Editor: William Pollock
Technical Reviewers: John Baldwin, Benno Rice, and George V. Neville-Neil
Copyeditor: Julianne Jigour
Compositor: Susan Glinert Stevens
Proofreader: James Fraleigh
Indexer: Nancy Guenther

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:
No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1.415.863.9900; info@nostarch.com
www.nostarch.com

Library of Congress Cataloging-in-Publication Data

```
Lucas, Michael, 1967-
  Absolute FreeBSD : the complete guide to FreeBSD / Michael W. Lucas. -- 2nd ed.
       p. cm.
  Includes index.
  ISBN-13: 978-1-59327-151-0
  ISBN-10: 1-59327-151-4
 1. FreeBSD.  2. UNIX (Computer file)  3. Internet service providers--Computer
programs.  4. Web servers--Computer programs.  5. Client/server computing.  I. Title.
QA76.76.O63L83 2007
004'.36--dc22
                                                                    2007036190
```

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

About the Author

After using Unix since the late '80s and spending twenty-odd years as a network and sytem administrator specializing in building and maintaining high-availability systems, Michael W. Lucas now writes about them for a living. He's written more than 30 books, which have been translated into nine languages. His critically acclaimed titles include *Absolute OpenBSD*, *Cisco Routers for the Desperate*, and *PGP & GPG*, all from No Starch Press. Learn more at *https://mwl.io/*.

About the Technical Reviewers

John Baldwin joined the FreeBSD Project as a committer in 1999. He has worked in several areas of the system, including SMP infrastructure, the network stack, virtual memory, and device driver support. John has served on the Core and Release Engineering teams and organized several FreeBSD developer summits.

Benno Rice has been using FreeBSD since 1995 and has been a committer since 2000 when he started the PowerPC port. Since then he has worked in a variety of areas and for a number of FreeBSD-using companies. He has also served on the Core Team and presented on FreeBSD-related topics at several conferences.

George V. Neville-Neil works on networking and operating system code for fun and profit. His areas of interest are code spelunking, operating systems, networking, and time protocols. He is the co-author with Marshall Kirk McKusick and Robert N. M. Watson of *The Design and Implementation of the FreeBSD Operating System* (Addison-Wesley Professional, 2004).

BRIEF CONTENTS

Foreword by Marshall Kirk McKusick

Acknowledgments

Introduction

Chapter 1: Getting More Help

Chapter 2: Before You Install

Chapter 3: Installing

Chapter 4: Start Me Up! The Boot Process

Chapter 5: Read This Before You Break Something Else! (Backup and Recovery)

Chapter 6: Kernel Games

Chapter 7: The Network

Chapter 8: Configuring Networking

Chapter 9: Securing Your System

Chapter 10: Disks, Partitioning, and GEOM

Chapter 11: The Unix File System

Chapter 12: The Z File System

Chapter 13: Foreign Filesystems

Chapter 14: Exploring /etc

Chapter 15: Making Your System Useful

Chapter 16: Customizing Software with Ports

Chapter 17: Advanced Software Management

Chapter 18: Upgrading FreeBSD

Chapter 19: Advanced Security Features

Chapter 20: Small System Services

Chapter 21: System Performance and Monitoring

Chapter 22: Jails

Chapter 23: The Fringe of FreeBSD

Chapter 24: Problem Reports and Panics

Afterword

Bibliography

Index

CONTENTS IN DETAIL

FOREWORD by Marshall Kirk McKusick

ACKNOWLEDGMENTS

INTRODUCTION

What Is FreeBSD?
- BSD: FreeBSD's Granddaddy
- The BSD License
- The AT&T/CSRG/BSDi Iron Cage Match
- The Birth of FreeBSD

FreeBSD Development
- Committers
- Contributors
- Users

Other BSDs
- NetBSD
- OpenBSD
- DragonFly BSD
- macOS
- FreeBSD's Children

Other Unixes
- Solaris
- illumos
- AIX
- Linux
- Other Unixes

FreeBSD's Strengths
- Portability
- Power
- Simplified Software Management
- Customizable Builds

Advanced Filesystems
Who Should Use FreeBSD?
Who Should Run Another BSD?
Who Should Run a Proprietary Operating System?
How to Read This Book
What Must You Know?
For the New System Administrator
Desktop FreeBSD
How to Think About Unix
Notes on the Third Edition
Contents of This Book

1
GETTING MORE HELP
Why Not Beg for Help?
The FreeBSD Attitude
Support Options
Man Pages
Manual Sections
Navigating Man Pages
Finding Man Pages
Section Numbers and Man
Man Page Contents
FreeBSD.org
Web Documents
The Mailing List Archives
The Forums
Other Websites
Using FreeBSD Problem-Solving Resources
Checking the Handbook and FAQ
Checking the Man Pages
Mailing Lists Archives and Forums
Using Your Answer
Asking for Help

Composing Your Message
Responding to Email
The Internet Is Forever

2
BEFORE YOU INSTALL
Default Files
Configuration with UCL
FreeBSD Hardware
Proprietary Hardware
Hardware Requirements
BIOS versus EFI
Disks and Filesystems
FreeBSD Filesystems
Filesystem Encryption
Disk Partitioning Methods
Partitioning with UFS
Multiple Operating Systems
Multiple Hard Drives
Swap Space
Getting FreeBSD
FreeBSD Versions
Choosing Installation Images
Network Installs

3
INSTALLING
Core Settings
Distribution Selection
Disk Partitioning
UFS Installs
ZFS Installs
Network and Service Configuration
Finishing the Install

4
START ME UP! THE BOOT PROCESS

Power-On
- Unified Extensible Firmware Interface
- Basic Input/Output System

The Loader
- Boot Multi User [Enter]
- Boot FreeBSD in Single-User Mode
- Escape to Loader Prompt
- Reboot

Single-User Mode
- Disks in Single-User Mode
- Programs Available in Single-User Mode
- The Network in Single-User Mode
- Uses for Single-User Mode

The Loader Prompt
- Viewing Disks
- Loader Variables
- Reboot
- Booting from the Loader

Loader Configuration

Boot Options

Startup Messages

Multiuser Startup
- /etc/rc.conf, /etc/rc.conf.d, and /etc/defaults/rc.conf

The rc.d Startup System
- The service(8) Command
- System Shutdown

Serial Consoles
- Serial Protocol
- Physical Serial Console Setup
- IPMI Serial Console Setup
- Configuring FreeBSD's Serial Console
- Using Serial Consoles

Working at the Console

5
READ THIS BEFORE YOU BREAK SOMETHING ELSE! (BACKUP AND RECOVERY)

System Backups
Backup Tapes
Tape Drive Device Nodes, Rewinding, and Ejecting
The $TAPE Variable
Tape Status with mt(1)
Other Tape Drive Commands
BSD tar(1)
tar Modes
Other tar Features
Compression
Permissions Restore
And More, More, More . . .
Recording What Happened
Repairing a Broken System

6
KERNEL GAMES

What Is the Kernel?
Kernel State: sysctl
sysctl MIBs
sysctl Values and Definitions
Viewing sysctls
Changing sysctls
Setting sysctls Automatically
The Kernel Environment
Viewing the Kernel Environment
Dropping Hints to Device Drivers
Kernel Modules
Viewing Loaded Modules

Loading and Unloading Modules
Loading Modules at Boot
Build Your Own Kernel
Preparations
Buses and Attachments
Back Up Your Working Kernel
Configuration File Format
Configuration Files
Building a Kernel
Booting an Alternate Kernel
Custom Kernel Configuration
Trimming a Kernel
Troubleshooting Kernel Builds
Inclusions, Exclusions, and Expanding the Kernel
NOTES
Inclusions and Exclusions
Skipping Modules

7
THE NETWORK
Network Layers
The Physical Layer
Datalink: The Physical Protocol
The Network Layer
Heavy Lifting: The Transport Layer
Applications
The Network in Practice
Getting Bits and Hexes
Network Stacks
IPv4 Addresses and Netmasks
Computing Netmasks in Decimal
Unusable IP Addresses
Assigning IPv4 Addresses
IPv6 Addresses and Subnets

IPv6 Basics
Understanding IPv6 Addresses
IPv6 Subnets
Link-Local Addresses
Assigning IPv6 Addresses
TCP/IP Basics
ICMP
UDP
TCP
How Protocols Fit Together
Transport Protocol Ports
Understanding Ethernet
Protocol and Hardware
MAC Addresses

8
CONFIGURING NETWORKING
Network Prerequisites
Configuring Changes with ifconfig(8)
Adding an IP to an Interface
Testing Your Interface
Set Default Route
Multiple IP Addresses on One Interface
Renaming Interfaces
DHCP
Reboot!
The Domain Name Service
Host/IP Information Sources
Local Names with /etc/hosts
Configuring Nameservice
Caching Nameserver
Network Activity
Current Network Activity
What's Listening on Which Port?

Port Listeners in Detail
Network Capacity in the Kernel
Optimizing Network Performance
Optimizing Network Hardware
Memory Usage
Maximum Incoming Connections
Polling
Other Optimizations
Network Adapter Teaming
Aggregation Protocols
Configuring lagg(4)
Virtual LANs
Configuring VLAN Devices
Configuring VLANs at Boot

9
SECURING YOUR SYSTEM
Who Is the Enemy?
Script Kiddies
Disaffected Users
Botnets
Motivated Skilled Attackers
FreeBSD Security Announcements
User Security
Creating User Accounts
Configuring Adduser: /etc/adduser.conf
Editing Users
Shells and /etc/shells
root, Groups, and Management
The root Password
Groups of Users
Using Groups to Avoid Root
Tweaking User Security
Restricting Login Ability

Restricting System Usage
File Flags
Setting and Viewing File Flags
Securelevels
Securelevel Definitions
Which Securelevel Do You Need?
What Won't Securelevels and File Flags Accomplish?
Living with Securelevels
Network Targets
Putting It All Together

10
DISKS, PARTITIONING, AND GEOM
Disks Lie
Device Nodes
The Common Access Method
What Disks Do You Have?
Non-CAM Devices
The GEOM Storage Architecture
GEOM Autoconfiguration
GEOM vs. Volume Managers
Providers, Consumers, and Slicers
GEOM Control Programs
GEOM Device Nodes and Stacks
Hard Disks, Partitions, and Schemes
The Filesystem Table: /etc/fstab
What's Mounted Now?
Disk Labeling
Viewing Labels
Sample Labels
GEOM Withering
The gpart(8) Command
Viewing Partitions
Other Views

Removing Partitions
Scheming Disks
Removing the Disk Partitioning Scheme
Assigning the Partitioning Scheme
The GPT Partitioning Scheme
GPT Device Nodes
GPT Partition Types
Creating GPT Partitions
Resizing GPT Partitions
Changing Labels and Types
Booting on Legacy Hardware
Unified Extensible Firmware Interface and GPT
Expanding GPT Disks
The MBR Partitioning Scheme
What Is the Master Boot Record?
BSD Labels
MBR Device Nodes
MBR and Disklabel Alignment
Creating Slices
Removing Slices
Activating Slices
BSD Labels
Creating a BSD Label
Creating BSD Label Partitions
Assigning Specific Partition Letters

11
THE UNIX FILE SYSTEM

UFS Components
The Fast File System
How UFS Uses FFS
Vnodes
Mounting and Unmounting Filesystems
Mounting Standard Filesystems

Special Mounts
Unmounting a Partition
UFS Mount Options
UFS Resiliency
Soft Updates
Soft Updates Journaling
GEOM Journaling
Creating and Tuning UFS Filesystems
UFS Labeling
Block and Fragment Size
Using GEOM Journaling
Tuning UFS
Expanding UFS Filesystems
UFS Snapshots
Taking and Destroying Snapshots
Finding Snapshots
Snapshot Disk Usage
UFS Recovery and Repair
System Shutdown: The Syncer
Dirty Filesystems
File System Checking: fsck(8)
Forcing Read-Write Mounts on Dirty Disks
Background fsck, fsck -y, Foreground fsck, Oy Vey!
UFS Space Reservations
How Full Is a Partition?
Adding New UFS storage
Partitioning the Disk
Configuring /etc/fstab
Installing Existing Files onto New Disks
Stackable Mounts

12
THE Z FILE SYSTEM
Datasets

Dataset Properties
Managing Datasets
ZFS Pools
Pool Details
Pool Properties
Viewing Pool Properties
Virtual Devices
VDEV Types and Redundancy
Managing Pools
ZFS and Disk Block Size
Creating and Viewing Pools
Multi-VDEV Pools
Destroying Pools
Errors and -f
Copy-On-Write
Snapshots
Creating Snapshots
Accessing Snapshots
Destroying Snapshots
Compression
Pool Integrity and Repair
Integrity Verification
Repairing Pools
Pool Status
Boot Environments
Viewing Boot Environments
Creating and Accessing Boot Environments
Activating Boot Environments
Removing Boot Environments
Boot Environments at Boot
Boot Environments and Applications

13
FOREIGN FILESYSTEMS

FreeBSD Mount Commands
- Supported Foreign Filesystems
- Permissions and Foreign Filesystems

Using Removable Media
- Ejecting Removable Media
- Removable Media and /etc/fstab
- Formatting FAT32 Media
- Creating Optical Media
- Writing Images to Thumb Drives

Memory Filesystems
- tmpfs
- Memory Disks
- Mounting Disk Images
- Filesystems in Files

devfs
- /dev at Boot
- Global devfs Rules
- Dynamic Device Management with devd(8)

Miscellaneous Filesystems

The Network File System
- NFS Versions
- Configuring the NFS Server
- Configuring NFS Exports
- Enabling the NFS Client

The Common Internet File System
- Prerequisites
- Kernel Support
- Configuring CIFS
- nsmb.conf Keywords
- CIFS Name Resolution
- Other smbutil(1) Functions
- Mounting a Share
- Other mount_smbfs Options
- nsmb.conf Options

CIFS File Ownership
Serving CIFS Shares

14
EXPLORING /ETC
/etc Across Unix Species
/etc/adduser.conf
/etc/aliases
/etc/amd.map
/etc/auto_master
/etc/blacklistd.conf
/etc/bluetooth, /etc/bluetooth.device.conf, and /etc/defaults/bluetooth.device.conf
/etc/casper
/etc/crontab and /etc/cron.d
/etc/csh.*
/etc/ddb.conf
/etc/devd.conf
/etc/devfs.conf, /etc/devfs.rules, and /etc/defaults/devfs.rules
/etc/dhclient.conf
/etc/disktab
/etc/dma/
/etc/freebsd-update.conf
/etc/fstab
/etc/ftp.*
/etc/group
/etc/hostid
/etc/hosts
/etc/hosts.allow
/etc/hosts.equiv
/etc/hosts.lpd
/etc/inetd.conf
/etc/libmap.conf
/etc/localtime

/etc/locate.rc
/etc/login.*
/etc/mail
/etc/mail.rc
/etc/mail/mailer.conf
/etc/make.conf
    CFLAGS
    COPTFLAGS
    CXXFLAGS
/etc/master.passwd
/etc/motd
/etc/mtree
/etc/netconfig
/etc/netstart
/etc/network.subr
/etc/newsyslog.conf
/etc/nscd.conf
/etc/nsmb.conf
/etc/nsswitch.conf
/etc/ntp/, /etc/ntp.conf
/etc/opie*
/etc/pam.d/*
/etc/passwd
/etc/pccard_ether
/etc/periodic.conf and /etc/defaults/periodic.conf
    daily_output="root"
    daily_show_success="YES"
    daily_show_info="YES"
    daily_show_badconfig="NO"
    daily_local="/etc/daily.local"
/etc/pf.conf, /etc/pf.os
/etc/phones
/etc/portsnap.conf
/etc/ppp/

/etc/printcap
/etc/profile
/etc/protocols
/etc/pwd.db
/etc/rc*
/et/regdomain.xml
/etc/remote
/etc/resolv.conf
/etc/rpc
/etc/security/
/etc/services
/etc/shells
/etc/skel/
/etc/snmpd.config
/etc/spwd.db
/etc/src.conf
/etc/ssh/
/etc/ssl/
/etc/sysctl.conf
/etc/syslog.conf, /etc/syslog.conf.d/
/etc/termcap, /etc/termcap.small
/etc/ttys
/etc/unbound/
/etc/wall_cmos_clock
/etc/zfs/

15
MAKING YOUR SYSTEM USEFUL

Ports and Packages
Packages
- Package Files
- Introducing pkg(8)
- Installing pkg(8)
- Common pkg Options

Configuring pkg(8)
Finding Packages
Installing Software
The Package Cache
Package Information and Automatic Installs
Uninstalling Packages
Changing the Package Database
Locking Packages
Package Files
Package Integrity
Package Maintenance
Package Networking and Environment
Package Repositories
Repository Configuration
Repository Customization
Repository Inheritance
Package Branches
Upgrading Packages

16
CUSTOMIZING SOFTWARE WITH PORTS
Making Software
Source Code and Software
The Ports Collection
Ports
The Ports Index
Searching the Index
Legal Restrictions
What's In a Port?
Installing a Port
Port Customization Options
Building Packages
Uninstalling and Reinstalling Ports
Tracking Port Build Status

Cleaning Up Ports
Read-Only Ports Tree
Changing the Install Path
Private Package Repositories
Poudriere Resources
Installing and Configuring Poudriere
Poudriere Jail Creation
Install a Poudriere Ports Tree
Configuring Poudriere Ports
Running Poudriere
Using the Private Repository
All Poudrieres, Large and Small
Small Systems
Large Systems
Updating Poudriere
More Poudriere

17
ADVANCED SOFTWARE MANAGEMENT
Using Multiple Processors: SMP
Kernel Assumptions
SMP: The First Try
Today's SMP
Processors and SMP
Threads, Threads, and More Threads
Startup and Shutdown Scripts
rc Script Ordering
A Typical rc Script
Special rc Script Providers
Vendor Startup/Shutdown Scripts
Debugging Custom rc Scripts
Managing Shared Libraries
Shared Library Versions and Files
Attaching Shared Libraries to Programs

LD_LIBRARY_PATH and LD_PRELOAD
What a Program Wants
Remapping Shared Libraries
Running Software from the Wrong OS
Recompilation
Emulation
ABI Reimplementation
Binary Branding
Supported ABIs
Installing and Configuring the Linuxulator
Using Linux Mode
Debugging Linux Mode
Running Software from the Wrong Architecture or Release

18
UPGRADING FREEBSD

FreeBSD Versions
Releases
FreeBSD-current
FreeBSD-stable
Snapshots
FreeBSD Support Model
Testing FreeBSD
Which Version Should You Use?
Upgrade Methods
Binary Updates
/etc/freebsd-update.conf
Running freebsd-update(8)
Reverting Updates
Scheduling Binary Updates
Optimizing and Customizing FreeBSD Update
Upgrading via Source
Which Source Code?
Updating Source Code

Building FreeBSD from Source
Build the World
Build, Install, and Test a Kernel
Prepare to Install the New World
Installing the World
Customizing Mergemaster
Upgrades and Single-User Mode
Shrinking FreeBSD
Packages and System Upgrades
Updating Installed Ports

19
ADVANCED SECURITY FEATURES

Unprivileged Users
The nobody Account
A Sample Unprivileged User
Network Traffic Control
Default Accept vs. Default Deny
TCP Wrappers
Configuring Wrappers
Wrapping Up Wrappers
Packet Filtering
Enabling PF
Default Accept and Default Deny in Packet Filtering
Basic Packet Filtering and Stateful Inspection
Configuring PF
Small-Server PF Rule Sample
Managing PF
Blacklistd(8)
PF and Blacklistd
Configuring Blacklistd
Configuring Blacklistd Clients
Managing Blacklistd
De-Blacklisting

Public-Key Encryption
OpenSSL
Certificates
TLS Trick: Connecting to TLS-Protected Ports
Global Security Settings
Install-Time Options
Secure Console
Nonexecutable Stack and Stack Guard
Other Security Settings
Preparing for Intrusions with mtree(1)
Running mtree(1)
mtree(1) Output: The Spec File
The Exclusion File
Saving the Spec File
Finding System Differences
Monitoring System Security
Package Security
If You're Hacked

20
SMALL SYSTEM SERVICES
Secure Shell
The SSH Server: sshd(8)
SSH Keys and Fingerprints
Configuring the SSH Daemon
Managing SSH User Access
SSH Clients
Email
mailwrapper(8)
The Dragonfly Mail Agent
The Aliases File and DMA
Network Time
Setting the Time Zone
Network Time Protocol

Name Service Switching
inetd
    /etc/inetd.conf
    Configuring inetd Servers
    Starting inetd(8)
    Changing inetd's Behavior
DHCP
    How DHCP Works
    Configuring dhcpd(8)
    Managing dhcpd(8)
Printing and Print Servers
    /etc/printcap
    Enabling LPD
TFTP
    Root Directory
    tftpd and Files
    File Ownership
    tftpd(8) Configuration
Scheduling Tasks
    cron(8)
    periodic(8)

21
SYSTEM PERFORMANCE AND MONITORING
Computer Resources
Checking the Network
General Bottleneck Analysis with vmstat(8)
    Processes
    Memory
    Paging
    Disks
    Faults
    CPU
    Using vmstat

- Continuous vmstat
- Disk I/O
- CPU, Memory, and I/O with top(1)
  - UFS and top(1)
  - ZFS and top(1)
  - Process List
  - top(1) and I/O
- Following Processes
- Paging and Swapping
  - Paging
  - Swapping
- Performance Tuning
  - Memory Usage
  - Swap Space Usage
  - CPU Usage
  - Rescheduling
  - Reprioritizing with Niceness
- Status Mail
- Logging with syslogd
  - Facilities
  - Levels
  - Processing Messages with syslogd(8)
  - syslogd Customization
- Log File Management
  - Log File Path
  - Owner and Group
  - Permissions
  - Count
  - Size
  - Time
  - Flags
  - Pidfile
  - Signal
  - Sample newsyslog.conf Entry

FreeBSD and SNMP
SNMP 101
Configuring bsnmpd

22
JAILS
Jail Basics
Jail Host Server Setup
Jail Host Storage
Jail Networking
Jails at Boot
Jail Setup
Jail Userland
/etc/jail.conf
Testing and Configuring a Jail
Jail Startup and Shutdown
Jail Dependencies
Managing Jails
Viewing Jails and Jail IDs
Jailed Processes
Running Commands in Jails
Installing Jail Packages
Updating Jails
More Jail Options
Jailing Ancient FreeBSD
Last Jail Notes

23
THE FRINGE OF FREEBSD
Terminals
/etc/ttys Format
Insecure Console
Managing Cloudy FreeBSD
LibXo

Universal Configuration Language
Diskless FreeBSD
Diskless Clients
DHCP Server Setup
tftpd and the Boot Loader
Diskless Security
The NFS Server and the Diskless Client Userland
Diskless Farm Configuration
Configuration Hierarchy
Diskless Remounting /etc
Finalizing Setup
Installing Packages
SSH Keys
Storage Encryption
Generating and Using a Cryptographic Key
Filesystems on Encrypted Devices

24
PROBLEM REPORTS AND PANICS
Bug Reports
Before Filing a Bug
Bad Bug Reports
The Fix
Filing Bugs
After Submitting
System Panics
Recognizing Panics
Responding to a Panic
Preparations
The Crash Dump in Action
Testing Crash Dumps
Crash Dump Types
Textdumps
Dumps and Security

AFTERWORD
The FreeBSD Community
Why Do We Do It?
What Can You Do?
If Nothing Else
Getting Things Done

BIBLIOGRAPHY
References
Books I've Written

INDEX

FOREWORD

I am happy to write the foreword to Michael Lucas's third edition of *Absolute FreeBSD*. For 15 years, Michael's *Absolute* series has provided the definitive guide to BSD software, filling in the whats and whys left unexplained by the detailed but largely factual documentation. And, as its name implies, it distills to its essence the enormous volume of FreeBSD documentation so that those new to the system can get up to speed quickly.

Michael is an important contributor to the FreeBSD community. He has filled many of the roles that contributors can take: answering questions, filling in pieces of missing documentation, helping to make connections in the community, and generally identifying and facilitating the things that need to be done. Michael has interacted with thousands of people: hobbyists, professional software developers, system administrators, and university professors. Much of his real-world experience and understanding of what people are trying to get done has been distilled into this book.

I have been involved with the BSD software since its beginning in 1977 as a student project of my office mate, Bill Joy, at the University of California at Berkeley. By 1980, the BSD distributions had grown from a few programs that could be added to an AT&T UNIX system to a complete system coordinated by four people who called themselves the Computer Systems Research Group (CRSG). By 1983, the socket interface had been designed and TCP/IP had been implemented underneath it, allowing a small set of trusted external contributors to log into the CSRG development machines over the ARPAnet (which later became the internet) and directly update the sources using SCCS, a very early source code control system. The CSRG staff could then use SCCS to track changes and verify them before doing distributions. This structure formed the basis for the current BSD-based projects once BSD was spun off from the university as open source in 1992.

Starting with the open-source distribution, FreeBSD initially ran on only the early PC computers. Over the past quarter century, thousands of developers have contributed to FreeBSD to make it into a powerful network

operating system with state-of-the-art features that runs on all the modern computing platforms. FreeBSD powers core internet companies worldwide. From Netflix movie distribution to WhatsApp messaging, from Network Appliance and Dell/Isilon storage products to Juniper routers, from the foundation of Apple's iOS to the base libraries and services of Google's Android, it is hard to throw a rock at the internet without hitting FreeBSD. However, FreeBSD is not the product of any one company, but of a large open source community: the FreeBSD Project, made up of developers, users, and countless supporters and advocates. While you can, as many people do, use FreeBSD simply as a piece of software without ever interacting with that community, you can significantly enrich your FreeBSD experience by becoming a part of that community.

Whether you are a first-time user or a kernel hacker, the resources available via the *http://www.freebsd.org/* website, countless mailing lists, regional user groups, and conferences can be invaluable. Have a question? Just email *questions@FreeBSD.org*, and one or more of the hundreds of volunteers will undoubtedly answer it. Want to learn more about the exciting new features coming in future FreeBSD versions? Read the Project's quarterly status reports or development mailing lists, or attend one of the many regional BSD conferences taking place around the world.

These resources are a product of the FreeBSD Project and its community, a large number of collaborating individuals and companies, as well as the FreeBSD Foundation, a nonprofit organization coordinating funding, legal resources, and support for development work and community activities. Michael's easy-to-use book provides a gateway for newbies to benefit from this community's expertise and to become active users of FreeBSD themselves.

FreeBSD is open source software, available for you to use and distribute at no charge. By helping to support, advocate, or even develop FreeBSD, you can give back to the FreeBSD Project and help this community grow.

Whether you are a new user of FreeBSD or an experienced one, I am confident you will find Absolute FreeBSD a book you want to keep close at hand.

Marshall Kirk McKusick
FreeBSD Committer
Treasurer, FreeBSD Foundation

Berkeley, California
January 2018

ACKNOWLEDGMENTS

This book would not exist without decades of support from the FreeBSD community. Many people have told me that they reach for my books to learn how to accomplish something. What they don't see is how many times I've reached out to mailing lists, forums, and user groups to get that same sort of help—not to mention all the times I've used other people's archived discussions to figure out where I went horribly wrong. In addition to all those folks who've gone before me, though, I need to name those who helped me on this particular book.

Gavin Atkinson, Diane Bruce, Julian Elischer, Lars Engels, Alex Kozlov, Steven Kreuzer, Ganael Laplanche, Greg "Groggy" Lehey, Warner Losh, Remko Lodder, Ruslan Makhmatkhanov, Hiren Panchasara, Colin Percival, Matthew Seaman, Lev Serebryakov, Carlo Strub, Romain Tartière, and Thomas Zander all provided vital feedback on earlier versions of this book. Some of them read individual chapters that they have special expertise in, while others read the whole blasted book whether they knew the topic or not. Both kinds of feedback are invaluable. John Baldwin, Benno Rice, and George Neville-Neil collaborated on performing a final technical review, catching errors that ranged from the subtly horrific to the blatantly appalling. Any errors that remain in this book were introduced by myself, despite all these people's best efforts.

I've also received years of support from Allan Jude and Benedict Reuschling of the BSDNow (*https://www.bsdnow.tv/*) podcast, along with alumnus Kris Moore. They've backed my work even when they had no idea what the heck I was doing. Their show is a great source of BSD-related news, education, and gossip. (It's a community. There's always gossip.) Just this week, they walked me through understanding the scheduler in a way I never have before.

Bert JW Regeer donated $800 to the FreeBSD Foundation for the dubious privilege of being abused in this book. I sincerely thank Bert for being a good sport, and handling all the indignities I heap upon him with

grace and aplomb.

Of all the folks who back me on Patreon, I must especially thank Stefan Johnson and Kate Ebneter. Because that's what their Patreon reward levels say I'll do. So: thank you!

Janelle over at No Starch Press had the unenviable job of shepherding this book through production, which is kind of like herding cats except the cats are angry and have switchblades. Thank you for dragging this tome across the finish line. I also need to thank the rest of the No Starch staff, who suffered through transforming my meandering babblings into a real book.

And as always, my gratitude to my amazing wife Liz.

INTRODUCTION

Welcome to *Absolute FreeBSD*! This book is a one-stop shop for system administrators who want to build, configure, and manage FreeBSD servers. It will also be useful for those folks who want to run FreeBSD on their desktops, embedded devices, server farms, and so on. By the time you finish this book, you should be able to use FreeBSD to provide network services. You should also understand how to manage, patch, and maintain your FreeBSD systems and have a basic understanding of networking, system security, and software management. We'll discuss FreeBSD versions 11 and 12, which are the most recent versions at the time this book is being released; however, most of this book applies to earlier and later versions as well.

What Is FreeBSD?

FreeBSD is a freely available Unix-like operating system popular with internet service providers, in appliances and embedded systems, and anywhere that reliability on commodity hardware is paramount. One day last week, FreeBSD miraculously appeared on the internet, fully formed, extruded directly from the mutant brain of its heroic creator's lofty intellect. Just kidding—the truth is far more impressive. FreeBSD is a result of almost four decades of continuous development, research, and refinement. The story of FreeBSD begins in 1979, with BSD.

BSD: FreeBSD's Granddaddy

Many years ago, AT&T needed a lot of specialized, custom-written

computer software to run its business. It wasn't allowed to compete in the computer industry, however, so it couldn't sell its software. Instead, AT&T licensed various pieces of software and the source code for that software to universities at low, low prices. The universities could save money by using this software instead of commercial equivalents with pricey licenses, and university students with access to this nifty technology could read the source code to see how everything worked. In return, AT&T got exposure, some pocket change, and a generation of computer scientists who had cut their teeth on AT&T technology. Everyone got something out of the deal. The best-known software distributed under this licensing plan was Unix.

Compared with modern operating systems, the original Unix had a lot of problems. Thousands of students had access to its source code, however, and hundreds of teachers needed interesting projects for their students. If a program behaved oddly, or if the operating system itself had a problem, the people who lived with the system on a day-to-day basis had the tools and the motivation to fix it. Their efforts quickly improved Unix and created many features we now take for granted. Students added the ability to control running processes, also known as *job control*. The Unix S51K filesystem made system administrators bawl like exhausted toddlers, so they replaced it with the Fast File System (FFS), whose features have spread into every modern filesystem. Many small, useful programs were written over the years, gradually replacing entire swaths of Unix.

The Computer Systems Research Group (CSRG) at the University of California, Berkeley, participated in these improvements and also acted as a central clearinghouse for Unix code improvements. CSRG collected changes from other universities, evaluated them, packaged them, and distributed the compilation for free to anyone with a valid AT&T UNIX license. The CSRG also contracted with the Defense Advanced Research Projects Agency (DARPA) to implement various features in Unix, such as TCP/IP. The resulting collection of software came to be known as the *Berkeley Software Distribution*, or *BSD*.

BSD users took the software, improved it further, and then fed their enhancements back into BSD. Today, we consider this to be a fairly standard way for an open source project to run, but in 1979 it was revolutionary. BSD was also quite successful; if you check the copyright statement on an old BSD system, you'll see this:

```
Copyright 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994
        The Regents of the University of California. All rights reserved.
```

Yep, 15 years of work—a lifetime in software development. How many other pieces of software are not only still in use, but still in active development, 15 years after work began? In fact, so many enhancements and improvements went into BSD that the CSRG found that over the years, it had replaced almost all of the original Unix with code created by the CSRG and its contributors. You had to look hard to find any original AT&T code.

Eventually, the CSRG's funding ebbed, and it became clear that the BSD project would end. After some political wrangling within the University of California, in 1992 the BSD code was released to the general public under what became known as the *BSD license*.

The BSD License

BSD code is available for anyone to use under what is probably the most liberal license in the history of software development. The license can be summarized as follows:

- Don't claim you wrote this.
- Don't blame us if it breaks.
- Don't use our name to promote your product.

This means that you can do almost anything you want with BSD code. (The original BSD license did require that users be notified if a software product included BSD-licensed code, but that requirement was later dropped.) There's not even a requirement that you share your changes with the original authors! People were free to take BSD and include it in proprietary products, open source products, or free products—they could even print it out on punch cards and cover the lawn with it. You want to run off 10,000 BSD CDs and distribute them to your friends? Enjoy. Instead of *copyright*, the BSD license is sometimes referred to as *copycenter*, as in *Take this down to the copy center and run off a few for yourself*. Not surprisingly, companies such as Sun Microsystems jumped right on it: it was free, it worked, and plenty of new graduates had experience with the technology—including Bill Joy, one of Sun's founders. One company, BSDi, was formed

specifically to take advantage of BSD Unix.

The AT&T/CSRG/BSDi Iron Cage Match

At AT&T, UNIX work continued apace even as the CSRG went on its merry way. AT&T took parts of the BSD Unix distribution, integrated them with its UNIX, and then relicensed the result back to the universities that provided those improvements. This worked well for AT&T until the company was broken up and the resulting companies were permitted to compete in the computer software business. AT&T had one particularly valuable property: a high-end operating system that had been extensively debugged by thousands of people. This operating system had many useful features, such as a variety of small but powerful commands, a modern filesystem, job control, and TCP/IP. AT&T started a subsidiary, Unix Systems Laboratories (USL), which happily started selling Unix to enterprises and charging very high fees for it, all the while maintaining the university relationship that had given it such an advanced operating system in the first place.

Berkeley's public release of the BSD code in 1992 was met with great displeasure from USL. Almost immediately, USL sued the university and the software companies that had taken advantage of the software, particularly BSDi. The University of California claimed that the CSRG had compiled BSD from thousands of third-party contributors unrelated to AT&T, and so it was the CSRG's intellectual property to dispose of as it saw fit.

This lawsuit motivated many people to grab a copy of BSD to see what all the fuss was about, while others started building products on top of it. One of these products was 386BSD, which would eventually be used as the core of FreeBSD 1.0.

In 1994, after two years of legal wrangling, the University of California lawyers proved that the majority of AT&T UNIX was actually taken in its entirety from BSD, rather than the other way around. To add insult to injury, AT&T had actually violated the BSD license by stripping the CSRG copyright from files it had assimilated. (Only a very special company can violate the world's most generous software license!) A half-dozen files were the only sources of contention, and to resolve these outstanding issues, USL donated some of them to BSD while retaining some as proprietary information.

Once the dust settled, a new version of BSD Unix was released to the world as BSD 4.4-Lite. A subsequent update, BSD 4.4-Lite2, is the grandfather of the current FreeBSD, as well as ancestor to every other BSD variant in use today.

The Birth of FreeBSD

One early result of BSD was 386BSD, a version of BSD designed to run on the cheap 386 processor.[1] The 386BSD project successfully ported BSD to Intel's 386 processor, but it stalled. After a period of neglect, a group of 386BSD users decided to branch out on their own and create FreeBSD so they could keep the operating system up to date. (Several other groups started their own branches off of 386BSD around the same time, of which only NetBSD remains.)

386BSD and FreeBSD 1 were derived from 1992's BSD release, the subject of AT&T's wrath. As a result of the lawsuit, all users of the original BSD were requested to base any further work on BSD 4.4-Lite2. BSD 4.4-Lite2 was not a complete operating system—in particular, those few files AT&T had retained as proprietary were vital to the system's function. (After all, if those files hadn't been vital, AT&T wouldn't have bothered!) The FreeBSD development team worked frantically to replace those missing files, and FreeBSD 2.0 was released shortly afterward. Development has continued ever since.

Today, FreeBSD is used across the internet by some of the most vital and visible internet-oriented companies. Netflix's content delivery system runs entirely on FreeBSD. IBM, Dell/EMC, Juniper, NetApp, Sony and many other hardware companies use FreeBSD in embedded systems where you'd never even know it unless someone told you. The fact is, if a company needs to pump serious internet bandwidth, it's probably running FreeBSD or one of its BSD relatives.

FreeBSD also finds its way into all sorts of embedded and dedicated-purpose devices. Do you have a PlayStation 4? Congratulations, you're running FreeBSD. I hear a root shell is hard to get on one of them, though.

Like smog, spiders, and corn syrup, FreeBSD is all around you; you simply don't see it because FreeBSD just works. The key to FreeBSD's reliability is the development team and user community—which are really

the same thing.

FreeBSD Development

There's an old saying that managing programmers is like herding cats. Despite the fact that the FreeBSD development team is scattered across the world and speaks dozens of languages, for the most part, the members work well together as parts of the FreeBSD community. They're more like a pride of lions than a collection of house cats. Unlike some other projects, all FreeBSD development happens in public. Three groups of people are responsible for FreeBSD's progress: committers, contributors, and users.

Committers

FreeBSD has about 500 developers, or committers. *Committers* have read-and-write access to the FreeBSD master source code repository and can develop, debug, or enhance any piece of the system. (The term *committer* comes from their ability to *commit* changes to the source code.) Because these commits can break the operating system in both subtle and obvious ways, committers carry a heavy responsibility. Committers are responsible for keeping FreeBSD working or, at worst, not breaking it as they add new features and evaluate patches from contributors. Most of these developers are volunteers; only a handful are actually paid to do this painstaking work, and most of those people are paid only as it relates to other work. For example, Intel employs committers to ensure that FreeBSD properly supports its network cards. FreeBSD has a high profile in the internet's heavy-lifting crowd, so Intel needs its cards to work on FreeBSD.

To plug yourself into the beehive of FreeBSD development, consider subscribing to the mailing list *FreeBSD-hackers@FreeBSD.org*, which contains most of the technical discussion. Some of the technical talk is broken out into more specific mailing lists—for example, fine details of the networking implementation are discussed in *FreeBSD-net@FreeBSD.org*.

Every few years, the committer team elects a small number of its members to serve as a core team, or *Core*. Core's work is simultaneously vital, underrated, and misunderstood. Core is theoretically responsible for the overall management of FreeBSD, but in practice, it manages little other

than resolving personality disputes and procedural conflicts among committers. Core also approves new committers and delegates responsibility for large parts of FreeBSD to individuals or groups. For example, it delegates authority over the ports and packages system to the ports management team. Core does not set architectural direction for FreeBSD, nor does it dictate processes or procedures; that's up to the committers, who must agree en masse. Core does suggest, cajole, mediate, and inspire, however.

Core also experiences the worst part of management. Some of the key functions of management in a company are oversight, motivation, and handling problems between people. Oversight is provided by the millions of users who will complain loudly when anything breaks or behaves unexpectedly, and FreeBSD committers are self-motivated. The ugly part of management is settling squabbles between two people, and that's the part Core gets stuck with. The status one gets from saying "I'm in Core" is an insufficient reward for having to manage the occasional argument between two talented developers who've gotten on each other's nerves. Fortunately such disagreements are rare and usually resolved quickly.

Contributors

In addition to the committer team, FreeBSD has thousands of contributors. *Contributors* don't have to worry about breaking the main operating system source code repository; they submit their patches for consideration by committers. Committers evaluate contributor submissions and decide what to accept and what to reject. A contributor who submits many high-quality patches is often asked to become a committer themselves.

For example, I spent several years contributing to FreeBSD whenever the urge struck me. Any time I feel that I've wasted my life, I can look at the FreeBSD website and see where my work was accepted by the committers and distributed to thousands of people. After I submitted the first edition of this book to the publisher, I spent my spare time submitting patches to the FreeBSD FAQ. Eventually, some members of the FreeBSD Documentation Project approached me and asked me to become a committer. As a reward, I got an email address and the opportunity to humiliate myself before thousands of people, once again demonstrating that no good deed goes unpunished.

If I had never contributed anything, I'd remain a user. Nothing's wrong with that, either.

Users

Users are the people who run FreeBSD systems. It's impossible to realistically estimate the number of FreeBSD users. While organizations such as the BSDstats Project (*http://www.bsdstats.org/*) make an effort, these projects are opt-in. They measure only folks who have installed FreeBSD and then installed the software that adds their system to the count. Most users download the whole of FreeBSD for free and never register, upgrade, or email a mailing list. We have no idea how many FreeBSD users are in the world.

Since FreeBSD is by far the most popular open source BSD, that's not an inconsiderable number of machines. And since one FreeBSD server can handle hundreds of thousands of internet domains, a disproportionate number of sites use FreeBSD as their supporting operating system. This means that there are hundreds of thousands, if not millions, of FreeBSD system administrators out in the world today.

Other BSDs

FreeBSD might be the most popular BSD, but it's not the only one. BSD 4.4-Lite2 spawned several different projects, each with its own focus and purpose. Those projects in turn had their own offspring, several of which thrive today.

NetBSD

NetBSD is similar to FreeBSD in many ways, and NetBSD and FreeBSD share developers and code. NetBSD's main goal is to provide a secure and reliable operating system that can be ported to any hardware platform with minimal effort. As such, NetBSD runs on Vixens, PocketPC devices, and high-end SPARC and Alpha servers. I ran NetBSD on my HP Jornada handheld computer.[2]

OpenBSD

OpenBSD branched off from NetBSD in 1996 with the goal of becoming the most secure BSD. OpenBSD was the first to support hardware-accelerated cryptography, and its developers are rightfully proud of the fact that their default installation was largely immune to remote exploits for several years. The OpenBSD team has contributed several valuable pieces of software to the world, including the LibreSSL TLS library and the OpenSSH suite used by almost everyone from Linux to Microsoft.

DragonFly BSD

DragonFly BSD forked from FreeBSD 4 in 2003. It developed in a different direction than FreeBSD, with a new kernel messaging system. DragonFly BSD has very high performance and its HAMMER filesystem supports snapshots and fine-grained history. Check out *http://www.dragonflybsd.org/* for more information.

macOS

Apple's macOS? That's right. Apple incorporates large chunks of FreeBSD into its macOS on an ongoing basis. If you're looking for a stable operating system with a friendly face and a powerful core, macOS is unquestionably for you. While FreeBSD makes an excellent desktop for a computer professional, I wouldn't put it in front of a random user. I would put macOS in front of that same random user without a second thought, however, and I'd even feel that I was doing the right thing. But macOS includes many things that aren't at all necessary for an internet server, and it runs only on Apple hardware, so I don't recommend it as an inexpensive general-purpose server.

FreeBSD's Children

Several projects have taken FreeBSD and built other projects or products on top of it. The award-winning FreeNAS transforms a commodity system into a network fileserver. The pfSense project transforms your system into a firewall with a nice web management interface. TrueOS gives FreeBSD a friendly face while supporting resource-intensive advanced features, like

ZFS, while GhostBSD puts a friendly face on equipment with less computing oomph. Other projects like this appear from time to time; while not all are successful, I'm sure by the time this book comes out, we'll have one or two more solid members of this group.

Other Unixes

Several other operating systems derive from or emulate primordial Unix in one way or another. This list is by no means exhaustive, but I'll touch on the high points.

Solaris

The best-known Unix might be Oracle Solaris. Solaris runs on high-end hardware that supports dozens of processors and gobs of disk. (Yes, *gobs* is a technical term, meaning *more than you could possibly ever need, and I know very well that you need more disk than I think you need*.) Solaris, especially early versions of Solaris, had strong BSD roots. Many enterprise-level applications run on Solaris. Solaris runs mainly on the SPARC hardware platform manufactured by Sun, which allows Sun to support interesting features, such as hot-swappable memory and mainboards.

The Oracle Corporation acquired Solaris when they bought Sun Microsystems in 2009. Oracle ceased Solaris development in 2016. While there's still an extensive installed base of Solaris systems and you can still get Solaris from Oracle, as of today, Oracle Solaris has no future.

illumos

Several years before Oracle purchased Sun Microsystems, Sun open sourced the majority of Solaris and sponsored the OpenSolaris project to improve that codebase. OpenSolaris ran successfully until Oracle shut down source access and reclaimed all of the OpenSolaris resources.

The OpenSolaris code was still available, though. The OpenSolaris community forked OpenSolaris into illumos (*http://illumos.org/*). If you miss Solaris, you can still use a free, modern, Solaris-like operating system. FreeBSD includes two important features from OpenSolaris, the Zetabyte

Filesystem (ZFS) and DTrace, a full-system tracing system.

AIX

Another Unix contender is IBM's entry, AIX. AIX's main claim to fame is its journaling filesystem, which records all disk transactions as they happen and allows for fast recovery from a crash. It was also IBM's standard Unix for many years, and anything backed by Big Blue shows up all over the place. AIX started life based on BSD, but AT&T has twiddled just about everything so that you won't find much BSD today.

Linux

Linux is a close cousin of Unix, written from the ground up. Linux is similar to FreeBSD in many ways, though FreeBSD has a much longer heritage and is friendlier to commercial use than Linux. Linux includes a requirement that any user who distributes Linux must make his or her changes available to the end user, while BSD has no such restriction. Of course, a Linux fan would say, "FreeBSD is more vulnerable to commercial exploitation than Linux." Linux developers believe in share-and-share-alike, while BSD developers offer a no-strings-attached gift to everyone. It all depends on what's important to you.

Many new Unix users have a perception of conflict between the BSD and Linux camps. If you dig a little deeper, however, you'll find that most of the developers of these operating systems communicate and cooperate in a friendly and open manner. It's just a hard fringe of users and developers that generate friction, much like different soccer teams' hooligans or different *Star Trek* series' fans.[3]

Other Unixes

Many Unixes have come and gone, while others stagger on. Past contenders include Silicon Graphics' IRIX, Hewlett-Packard's HP/UX, Tru64 Unix, and the suicidal SCO Group's UnixWare. Dig further and you'll find older castoffs, including Apple's A/UX and Microsoft's Xenix. (Yes, Microsoft was a licensed Unix vendor, back in that age when dinosaurs watched the skies nervously and my dad hunted mammoth for all the tribal rituals.) Many

high-end applications are designed to run best on one particular flavor of Unix. All modern Unixes have learned lessons from these older operating systems, and today's Unixes and Unix-like operating systems are remarkably similar.

WHY UNIX-LIKE?

One thing to note is that FreeBSD, Linux, and so on are called *Unix-like* instead of *Unix*. The term *Unix* is a trademark of The Open Group. For an operating system to receive the right to call itself Unix, the vendor must prove that the OS complies with the current version of the Single Unix Specification. While FreeBSD generally meets the standard, continuous testing and recertification cost money, which the FreeBSD Project doesn't have to spare. Certification as Unix also requires that someone sign a paper stating not only that he or she is responsible for FreeBSD's conformance to the Single Unix Specification but that he or she will fix any deviations from the standard that are found in the future. FreeBSD's development model makes this even more difficult—bugs are found and deviations are fixed, but there's nobody who can sign a piece of paper that guarantees 100 percent standards compliance.

FreeBSD's Strengths

After all this, what makes FreeBSD unique?

Portability

The FreeBSD Project's goal is to provide a freely redistributable, stable, and secure operating system that runs on the computer hardware that people are most likely to have access to. People have ported FreeBSD to a variety of less

popular platforms as well.

The best supported FreeBSD platform is the common 64-bit hardware developed by AMD, used by almost everyone, and even copied by Intel. FreeBSD also fully supports the older 32-bit computers, such as 486s and all the flavors of Pentiums. This book uses 64-bit commodity hardware, or *amd64*, as a reference platform.

FreeBSD runs well on several other hardware architectures but is not completely supported yet. These include 32-bit ARM processors and PowerPC. While these other platforms are not afterthoughts, they don't receive the same level of attention that x86 and amd64 do. The 64-bit ARM platform is expected to become Tier 1 shortly after this book comes out, however.

You can also load FreeBSD on certain older architectures, such as 64-bit SPARC. These platforms were once well supported but are on their way out.

Power

Since FreeBSD runs adequately on 486 processors, it runs extremely well on modern computers. It's rather nice to have an operating system that doesn't demand 8 cores and 12 gigs of RAM just to run the user interface. As a result, you can actually dedicate your hardware to accomplishing real work rather than tasks you don't care about. If you choose to run a pretty graphical interface with all sorts of spinning gewgaws and fancy whistles, FreeBSD will support you, and it won't penalize you if you choose otherwise. FreeBSD will also support you on the latest *n* -CPU hardware.

Simplified Software Management

FreeBSD also simplifies software management through the packaging system and the Ports Collection. Traditionally, running software on a Unix-like system required a great deal of expertise. Packages and ports simplify this considerably by automating and documenting the install, uninstall, and configuration processes for thousands of software packages.

We discuss packages in Chapter 15 and ports in Chapter 16.

Customizable Builds

FreeBSD provides a painless upgrade procedure, but it also lets you precisely customize the operating system for your hardware. Companies like Apple do exactly this, but they control both the hardware and the software; FreeBSD pulls off the same trick on commodity hardware.

Advanced Filesystems

A *filesystem* is how information is stored on the physical disk—it's what maps the file *My Resume* to a series of zeros and ones on a hard drive. FreeBSD includes two well-supported filesystems, UFS (Chapter 11) and ZFS (Chapter 12). UFS has been around for multiple decades and is highly damage-resistant. ZFS is younger but includes features such as network replication and self-healing.

Who Should Use FreeBSD?

While FreeBSD can be used as a powerful desktop or development machine, its history shows a strong bias toward network services: web, mail, file, and ancillary applications. FreeBSD is most famous for its strengths as an internet server, and it's an excellent choice as an underlying platform for any network service. If major firms such as Netflix count on FreeBSD to provide reliable service, it will work as well for you.

If you're thinking of running FreeBSD (or any Unix) on your desktop, you'll need to understand how your computer works. FreeBSD is not your best choice if you need point-and-click simplicity. If that's your goal, get a Mac so you can use the power of Unix when you need it and not worry about it the rest of the time. If you want to learn FreeBSD, though, running it on your desktop is the best way—as we'll discuss later.

Who Should Run Another BSD?

NetBSD and OpenBSD are FreeBSD's closest competitors. Unlike competitors in the commercial world, this competition is mostly friendly. FreeBSD, NetBSD, and OpenBSD freely share code and developers; some people even maintain the same subsystems in multiple operating systems.

If you want to use old or oddball hardware, NetBSD is a good choice for

you. For several years, I ran NetBSD on an ancient SGI workstation that I used as a Domain Name System (DNS) and fileserver. It did the job well until the hardware finally released a cloud of smoke and stopped working.

OpenBSD has implemented an impressive variety of security features. Some of the tools are eventually integrated into FreeBSD, but that takes months or years. Some of the tools can never be duplicated in FreeBSD, however. If you have real security concerns and can use a Unix-like system without the feature set FreeBSD provides, consider OpenBSD. Take a look at my book *Absolute OpenBSD* (No Starch Press, 2013) for an introduction.

If you're just experimenting to see what's out there, any BSD is good!

Who Should Run a Proprietary Operating System?

Operating systems such as macOS, Windows, AIX, and their ilk are still quite popular, despite the open source operating systems gnawing at their market share. High-end enterprises are pretty tightly shackled to commercial operating systems. While this is slowly changing, you're probably stuck with commercial operating systems in such environments. But slipping in an occasional FreeBSD machine to handle basic services, such as monitoring and department file serving, can make your life much easier at much lower cost. Companies like Dell/EMC/Isilon have built entire businesses using FreeBSD instead of commercial operating systems.

Of course, if the software you need runs only on a proprietary operating system, your choice is pretty clear. Still, always ask a vendor whether a FreeBSD version is available; you might be pleasantly surprised.

How to Read This Book

Many computer books are thick and heavy enough to stun an ox, if you have the strength to lift them high enough. Plus, they're either encyclopedic in scope or so painfully detailed that they're difficult to actually read. Do you really need to reference a screenshot when you're told to click OK or accept the license agreement? And when was the last time you actually sat down to read the encyclopedia?

Absolute FreeBSD is a little different. It's designed to be read once, from

front to back. You can skip around if you want to, but each chapter builds on what comes before it. While this isn't a small book, it's smaller than many popular computer books. After you've read it once, it makes a decent reference.

If you're a frequent buyer of computer books, please feel free to insert all that usual crud about "read a chapter at a time for best learning" and so on. I'm not going to coddle you—if you picked up this book, you either have two brain cells to rub together or you're visiting someone who does. (If it's the latter, hopefully your host is smart enough to take this book away from you before you learn enough to become dangerous.)

What Must You Know?

This book is aimed at the new Unix administrator. Three decades ago, the average Unix administrator had kernel programming experience and was working on their master's degree in computer science. Even a decade ago, they were already a skilled Unix user with real programming skills and most of a bachelor's degree in comp sci. Today, Unix-like operating systems are freely available, computers are cheaper than food, and even 12-year-old children can run Unix, read the source code, and learn enough to intimidate older folks. As such, I don't expect you to know a huge amount about Unix before firing it up.

To use this book to its full potential, you need to have familiarity with some basic tasks, such as how to change directories, list files in a directory, and log in with a username and password. If you're not familiar with basic commands and the Unix shell, I recommend you begin with a book like *UNIX System Administration Handbook* by Evi Nemeth and friends (Prentice Hall PTR, 2017). To make things easier on newer system administrators, I include the exact commands needed to produce the desired results. If you learn best by example, you should have everything you need right here.

You'll also need to know something about computer hardware—not a huge amount, mind you, but something. It helps to know how to recognize a SATA cable. Your need for this knowledge depends on the hardware you're using, but if you're interested enough to pick up this book and read this far, you probably know enough.

For the New System Administrator

If you're new to Unix, the best way to learn is to eat your own dog food. No, I'm not suggesting that you dine with Rover. If you ran a dog food company, you'd want to make a product that your own dog eats happily. If your dog turns his nose up at your latest recipe, you have a problem. The point here is that if you work with a tool or create something, you should actually use it. The same thing applies to any Unix-like operating system, including FreeBSD.

Desktop FreeBSD

If you're serious about learning FreeBSD, I suggest wiping out the operating system on your main computer and running FreeBSD instead. No, not a desktop-oriented FreeBSD derivative like TrueOS or GhostBSD: run raw FreeBSD. Yes, I know, now that dog food doesn't sound so bad. But learning an operating system is like learning a language; total immersion is the quickest and most powerful way to learn. That's what I did, and today I can make a Unix-like system do anything I want. I've written entire books on a FreeBSD laptop, using the open source text editor XEmacs and the LibreOffice.org business suite. I've also used FreeBSD to watch movies, rip and listen to MP3s, balance my bank accounts, process my email, and surf the web. The desktop in my lab has a dozen animated BSD daemons running around the window manager, and I occasionally take a break to zap them with my mouse. If this doesn't count as a Stupid Desktop Trick, I don't know what does.[4]

Many Unix system administrators these days come from a Windows background. They're beavering away in their little world when their manager swoops by and says, "You can handle one more system, can't you? Glad to hear it! It's a Unix box, by the way," and then vanishes into the managerial ether. Once the new Unix administrator decides not to quit her job and start a fresh and exciting career as a whale necropsy technician, she tentatively pokes at the system. She learns that `ls` is like `dir` and that `cd` is the same on both platforms. She can learn the commands by rote, reading, and experience. What she can't learn, coming from this background, is how a Unix machine *thinks*. Unix will not adjust to you; you must adjust to it. Windows and macOS require similar adjustments but hide them behind a

glittering facade. With that in mind, let's spend a little time learning how to think about Unix.

How to Think About Unix

These days, most Unix systems come with pretty GUIs out of the box, but they're just eye candy. No matter how graphically delicious the desktop looks, the real work happens on the command line. The Unix command line is actually one of Unix's strengths, and it's responsible for its unparalleled flexibility.

Unix's underlying philosophy is *many small tools, each of which does a single job well*. My mail server's local programs directory (*/usr/local/bin*) has 262 programs in it. I installed every one of them, either directly or indirectly. Most are small, simple programs that do only one task. This array of small tools makes Unix extremely flexible and adaptable. Many commercial software packages try to do everything; they wind up with all sorts of capabilities but only mediocre performance in their core functions. Remember, at one time you needed to be a programmer to use a Unix system, let alone run one. Programmers don't mind building their own tools. The Unix concept of pipes encouraged this.

Pipes

People used to GUI environments, such as Windows and macOS, are probably unfamiliar with how Unix handles output and input. They're used to clicking something and seeing either an OK message, an error, nothing, or (all too often) a pretty blue screen with nifty high-tech letters explaining in the language called *Geek* why the system crashed. Unix does things a little differently.

Unix programs have three channels of communication, or *pipes*: standard input, standard output, and standard error. Once you understand how each of these pipes works, you're a good way along to understanding the whole system.

Standard input is the source of information. When you're at the console typing a command, the standard input is the data coming from the keyboard. If a program is listening to the network, the standard input is the network. Many programs can rearrange standard input to accept data from the

network, a file, another program, the keyboard, or any other source.

The *standard output* is where the program's output is displayed. This is frequently the console (screen). Network programs usually return their output to the network. Programs might send their output to a file, to another program, over the network, or anywhere else available to the computer.

Finally, *standard error* is where the program sends its error messages. Frequently, console programs return their errors to the console; others log errors in a file. If you set up a program incorrectly, it just might discard all error information.

These three pipes can be arbitrarily arranged, a concept that's perhaps the biggest hurdle for new Unix users and administrators. For example, if you don't like the error messages appearing on the terminal, you can redirect them to a file. If you don't want to repeatedly type a lot of information into a command, you can put the information into a file (so you can reuse it) and dump the file into the command's standard input. Or, better still, you can run a command to generate that information and put it in a file, or just pipe (send) the output of the first command directly to the second, without even bothering with a file.

Small Programs, Pipes, and the Command Line

Taken to their logical extreme, these input/output pipes and the variety of tools seem overwhelming. When I saw a sysadmin type something like the following during my initial Unix training session, I gave serious consideration to changing careers.

```
$ tail -f /var/log/messages | grep -v popper | grep -v named &
```

Lines of incomprehensible text began spilling across the screen, and they kept coming. And worse still, my mentor kept typing as gibberish poured out! If you're from a point-and-click computing environment, a long string of commands like this is definitely intimidating. What do all those funky words mean? And an ampersand? You want me to learn *what*?

Think of learning to use the command line as learning a language. When learning a language, we start with simple words. As we increase our vocabulary, we also learn how to string the words together. We learn that placing words in a certain order makes sense, and that a different order

makes no sense at all. You didn't speak that well at three years old—give yourself some slack and you'll get there.

Small, simple programs and pipes provide almost unlimited flexibility. Have you ever wished you could use a function from one program in another program? By using a variety of smaller programs and arranging the inputs and outputs as you like, you can make a Unix system behave in any manner that amuses you. Eventually, you'll feel positively hogtied if you can't just run a command's output through `| sort -rnk 6 | less`.[5]

Everything Is a File

You can't be around Unix for very long before hearing that everything is a file. Programs, account information, and system configuration are all stored in files. Unix has no Windows-style registry; if you back up the files, you have the whole system.

What's more, the system identifies system hardware as files! Your CD-ROM drive is a file, */dev/cd0*. Serial ports appear as files like */dev/cuaa0*. Even virtual devices, such as packet sniffers and partitions on hard drives, are files.

When you have a problem, keep this fact in mind. Everything is a file, or is in a file, somewhere on your system. All you have to do is find it!

Notes on the Third Edition

Absolute BSD (No Starch Press, 2002) was my first technology book and was written when the various BSD operating systems had more in common than they wanted to admit. The second edition, *Absolute FreeBSD* (No Starch Press, 2007), came out after the BSDs had diverged, and detailed FreeBSD's advances in the previous five years. With another decade of growth, FreeBSD has evolved to compete with the best commercial operating systems. You'll find multiple top-tier filesystems. Disk management has changed to accommodate new partitioning methods. Virtualization is now a thing, and FreeBSD supports it as either a client or a host.

This growth has driven changes in this book.

We won't discuss configuring mail, DNS, or web servers. You have more software choices for these tasks than ever before. Entire books have been written about those choices and how to use them. I've written some of those

books. Those topics have been dropped to make space for FreeBSD-specific material, like ZFS and jails.

Some of these new features are hugely complex. Complete coverage of ZFS would fill entire books—I know, because I've written those books, too. FreeBSD supports a whole bunch of special-purpose filesystems, each incredibly useful to the folks who need them and totally irrelevant to those who don't. Rather than write a monster tome that nobody would actually read, I've elected to cover the material that every FreeBSD sysadmin *must* know. If you're interested in deeper coverage of a particular topic, it's available.

Some subsystems are undergoing radical revision. I could wait to write this book until every FreeBSD subsystem has a stable interface, but then it would come out about . . . never. As I write this, the bhyve developers are actively rototilling their entire configuration system. Given the choice between glossing over a topic and providing flat-out wrong material, I've chosen to skip detail on bhyve. I hope to be able to delete this paragraph before this book goes to press.

I've ruthlessly excised obsolete information from this edition. For example, modern disk drives don't generally have to worry about write caching. If you discover that a piece of advice you remember using doesn't appear in this book, please check FreeBSD's information resources to see whether that advice is still applicable.

Contents of This Book

Absolute FreeBSD, 3rd Edition contains the following chapters.

Chapter 1: Getting More Help

This chapter discusses the information resources the FreeBSD Project and its devotees provide for users. No one book can cover everything, but knowing how to use the many FreeBSD resources on the internet helps fill any gaps you find here.

Chapter 2: Before You Install

Getting FreeBSD installed isn't that hard. Make poor choices during the install, though, and you'll have a system that isn't suited for your needs.

The best way to avoid reinstalling is to think about your requirements and make all the decisions beforehand so that the actual install doesn't require any thought.

Chapter 3: Installing

This chapter gives you an overview of installing FreeBSD using different partitioning schemes and filesystems.

Chapter 4: Start Me Up! The Boot Process

This chapter teaches you about the FreeBSD boot process and how to make your system start, stop, and reboot in different configurations.

Chapter 5: Read This Before You Break Something Else! (Backup and Recovery)

Here we discuss how to back up your data on both a system-wide and a file-by-file level, and how to make your changes so that they can be easily undone.

Chapter 6: Kernel Games

This chapter describes configuring the FreeBSD kernel. Unlike some other operating systems, you're expected to tune FreeBSD's kernel to best suit your purposes. This gives you tremendous flexibility and lets you optimize your hardware's potential.

Chapter 7: The Network

Here we discuss the TCP/IP protocol that underlies the modern internet, both version 4 and version 6.

Chapter 8: Configuring the Network

FreeBSD doesn't only shuffle packets crazy fast, but it also supports virtual LANs, link aggregation, and more. We'll configure all of that here.

Chapter 9: Securing Your System

This chapter teaches you how to make your computer resist attackers and intruders.

Chapter 10: Disks, Partitioning, and GEOM

This chapter covers some of the details of working with hard drives in FreeBSD. Working with modern hardware means understanding multiple partitioning schemes, disk alignment, and FreeBSD's disk management infrastructure.

Chapter 11: The Unix File System

UFS has been FreeBSD's standard filesystem for decades, and the concepts of UFS pervade the whole operating system. Whether you intend to use UFS or not, you must understand its essentials.

Chapter 12: The Z File System

ZFS is a newer filesystem very popular on larger systems. If you're managing large amounts of data, you'll want ZFS.

Chapter 13: Foreign Filesystems

Every sysadmin needs to mount disks over the network or use ISOs without burning them to CD. This chapter takes you through those duties, as well as introducing FreeBSD-specific filesystems like devfs.

Chapter 14: Exploring /etc

This chapter describes the many configuration files in FreeBSD and how they operate.

Chapter 15: Making Your System Useful

Here I describe the packages system that FreeBSD uses to manage add-on software.

Chapter 16: Customizing Software with Ports

Sometimes the prebuilt packages won't cover everything you need. You can leverage FreeBSD's package-building system to create your own software packages, tuned to meet your exact needs.

Chapter 17: Advanced Software Management

This chapter discusses some of the finer points of running software on FreeBSD systems.

Chapter 18: Upgrading FreeBSD

This chapter teaches you how to use FreeBSD's upgrade process. The upgrade system is among the most remarkable and smooth of any operating system.

Chapter 19: Advanced Security Features

Here we discuss some of the more interesting security features found in FreeBSD.

Chapter 20: Small System Services

Here we discuss some of the small programs you'll need to manage in order to use FreeBSD properly.

Chapter 21: System Performance and Monitoring

This chapter covers some of FreeBSD's performance-testing and troubleshooting tools and shows you how to interpret the results. We also discuss logging and FreeBSD's SNMP implementation.

Chapter 22: Jails

FreeBSD has a process-isolation subsystem, much like Linux and Solaris containers, called *jails*. We'll cover the jail system and how you can leverage it for system security.

Chapter 23: The Fringe of FreeBSD

This chapter teaches you some of the more interesting tricks you can do with FreeBSD, such as running systems without disks and with tiny disks, as well as cloud-friendly features, like libxo.

Chapter 24: Problem Reports and Panics

This chapter teaches you how to deal with those rare occasions when a FreeBSD system fails, how to debug problems, and how to create a useful problem report.

You'll also find an annotated bibliography, an afterword, and a really spiffy professionally prepared index.

Okay, enough introductory stuff. Onward!

1
GETTING MORE HELP

As thick as this book is, it still can't possibly cover everything you must know about FreeBSD. After all, Unix has been kicking around for close to 50 years, BSD is pushing 40, and FreeBSD is old enough to have its doctorate. Even if you memorize this book, it won't cover every situation you might encounter. The FreeBSD Project supports a huge variety of information resources, including numerous mailing lists and the FreeBSD website, not to mention the official manual and Handbook. Its users maintain even more documentation on even more sites. The flood of information can be overwhelming in itself, and it can make you want to just email the world and beg for help. But before you send a question to a mailing list or forum, confirm that the information you need isn't already available.

Why Not Beg for Help?

FreeBSD provides two popular resources for assistance: mailing lists and forums. Many participants on both are very knowledgeable and can answer questions very quickly. But when you send a question to these community support resources, you're asking tens of thousands of people all over the world to take a moment to read your message. You're also asking that one or more of them take the time to help you instead of watching a favorite movie, enjoying dinner with their families, or catching up on sleep. Problems arise when these experts answer the same question 10, 50, or even hundreds of times. They become grumpy. Some get downright tetchy.

What makes matters worse is that many of these same people have spent

a great deal of time and effort making the answers to most of these questions available elsewhere. If you make it clear that you've already searched the resources and your answer really doesn't appear therein, you'll probably receive a polite, helpful answer. If you ask a question that's already been asked several hundred times, however, the expert on that subject just might snap and go ballistic on you. Do your homework, and chances are you'll get an answer more quickly than a fresh call for assistance could provide.

The FreeBSD Attitude

"Homework? What do you mean? Am I back in school? What do you want, burnt offerings on bended knee?" Yes, you are in school. The information technology business is nothing but lifelong, self-guided learning. Get used to it or get out. Burnt offerings, on the other hand, are difficult to transmit via email and aren't quite so useful today.

Most commercial software conceals its inner workings. The only access you have to them is through the options presented by the vendor. Even if you want to learn how something works, you probably can't. When something breaks, you have no choice but to call the vendor and grovel for help. Worse, the people paid to help you frequently know little more than you do.

If you've never worked with open source software vendors, FreeBSD's support mechanism might surprise you. There is no toll-free number to call and no vendor to escalate within. No, you may not speak to a manager and for a good reason: you are the manager. Congratulations on your promotion!

Support Options

That being said, you're not entirely on your own. The FreeBSD community includes numerous developers, contributors, and users who care very deeply about FreeBSD's quality, and they're happy to work with users who are willing to do their share of the labor. FreeBSD provides everything you need: complete access to the source code used to create the system, the tools needed to turn that source code into programs, and the same debuggers used by the developers. Nothing is hidden; you can see the innards, warts and all. You can view FreeBSD's development history since the beginning, including

every change ever made and the reason for it. These tools might be beyond your abilities, but that's not the Project's problem. Various community members are even happy to provide guidance as you develop your own skills so you can use those tools yourself. You'll have lots of help fulfilling your responsibilities.

As a grossly overgeneralized rule, people help those like themselves. If you want to use FreeBSD, you must make the jump from eating what the vendor gives you to learning how to cook. Every member of the FreeBSD user community learned how to use it, and they welcome interested new users with open arms. If you just want to know what to type without really understanding what's going on behind the scenes, you'll be better off reading the documentation; the general FreeBSD support community simply isn't motivated to help those who won't help themselves or who can't follow instructions.

If you want to use FreeBSD but have neither the time nor the inclination to learn more, invest in a commercial support contract. It might not be able to put you in touch with FreeBSD's owner, but at least you'll have someone to yell at. You'll find several commercial support providers listed on the FreeBSD website.

It's also important to remember that the FreeBSD Project maintains only FreeBSD. If you're having trouble with some other piece of software, a FreeBSD mailing list is not the place to ask for help. FreeBSD developers are generally proficient in a variety of software, but that doesn't mean they want to help you, say, configure KDE.

The first part of your homework, then, is to learn about the resources available beyond this book. These include the integrated manual, the FreeBSD website, the mailing list archives, and other websites.

Man Pages

Man pages (short for *manual pages*) are the primordial way of presenting Unix documentation. While man pages have a reputation for being obtuse, difficult, or even incomprehensible, they're actually quite friendly—for particular users. When man pages were first created, the average system administrator was a C programmer and, as a result, the pages were written by programmers, for programmers. If you can think like a programmer, man

pages are perfect for you. I've tried thinking like a programmer, but I achieved real success only after remaining awake for two days straight. (Lots of caffeine and a high fever help.)

Over the last several years, the skill level required for system administration has dropped; no longer must you be a programmer. Similarly, man pages have become more and more readable. Man pages are not tutorials, however; they explain the behavior of one particular program, not how to achieve a desired effect. While they're neither friendly nor comforting, they should be your first line of defense. If you send a question to a mailing list without checking the manual, you're likely to get a terse *man whatever* in response.

Manual Sections

The FreeBSD manual is divided into nine sections. Roughly speaking, the sections are:

1. General user commands
2. System calls and error numbers
3. C programming libraries
4. Devices and device drivers
5. File formats
6. Game instructions
7. Miscellaneous information
8. System maintenance commands
9. Kernel interfaces

Each man page starts with the name of the command it documents followed by its section number in parentheses, like this: reboot(8). When you see something in this format in other documents, it's telling you to read that man page in that section of the manual. Almost every topic has a man page. For example, to see the man page for the editor `vi`, type this command:

```
$ man vi
```

In response, you should see the following:

```
VI(1)      FreeBSD General Commands Manual          VI(1)

NAME
     ex, vi, view - text editors

SYNOPSIS
     ex [-FRrSsv] [-c cmd] [-t tag] [-w size] [file ...]
     vi  [-eFRrS] [-c cmd] [-t tag] [-w size] [file ...]
     view [-eFrS] [-c cmd] [-t tag] [-w size] [file ...]

DESCRIPTION
     vi is a screen-oriented text editor.  ex is a line-oriented text editor.
     ex and vi are different interfaces to the same program, and it is
     possible to switch back and forth during an edit session.  view is the
     equivalent of using the -R (read-only) option of vi.
:
```

The page starts with the title of the man page (vi) and the section number (1), and then it gives the name of the page. This particular page has three names: ex, vi, and view. Typing `man ex` or `man view` would take you to this same page.

Navigating Man Pages

Once you're in a man page, pressing the spacebar or the PGDN key takes you forward one full screen. If you don't want to go that far, pressing ENTER or the down arrow scrolls down one line. Typing b or pressing the PGUP key takes you back one screen. To search within a man page, type **/** followed by the word you're searching for. You'll jump down to the first appearance of the word, which will be highlighted. Typing n subsequently takes you to the next occurrence of the word.

This assumes that you're using the default BSD pager, more(1). If you're using a different pager, use that pager's syntax. Of course, if you know so much about Unix that you've already set your preferred default pager, you've probably skipped this part of the book.

Finding Man Pages

New users often say that they'd be happy to read the man pages if they could find the right one. You can perform basic keyword searches on the man pages with apropos(1) and whatis(1). To search any man page name or description that includes the word you specify, use apropos(1). To match

only whole words, use whatis(1). For example, if you're interested in the `vi` command, you might try the following:

```
$ apropos vi
unvis(1) - revert a visual representation of data back to original form
vidcontrol(1) - system console control and configuration utility
vis(1) - display non-printable characters in a visual format
madvise, posix_madvise(2) - give advice about use of memory
posix_fadvise(2) - give advice about use of file data
--snip--
```

This continues for a total of 581 entries, which is probably far more than you want to look at. Most of these have nothing to do with vi(1), however; the letters *vi* just appear in the name or description. *Device driver* is a fairly common term in the manual, so that's not surprising. On the other hand, whatis(1) gives more useful results in this case.

```
$ whatis vi
vi, ex, view, nex, nvi, nview(1) - text editors
$
```

We get only one result, clearly with relevance to vi(1). On other searches, apropos(1) gives better results than whatis(1). Experiment with both and you'll quickly learn how they fit your style.

The `man -k` command emulates apropos(1), while `man -f` emulates whatis(1).

Section Numbers and Man

You might find cases where a single command appears in multiple parts of the manual. For example, every man section has an introductory man page that explains the contents of the section. To specify a section to search for a man page, give the number immediately after the `man` command.

```
$ man 3 intro
```

This pulls up the introduction to section 3 of the manual. I recommend you read the intro pages to each section of the manual, if only to help you understand the breadth and depth of information available.

Man Page Contents

Man pages are divided into sections. While the author can put just about any

heading he or she likes into a man page, several are standard. See mdoc(7) for a partial list of these headings as well as other man page standards:

- NAME gives the name(s) of a program or utility. Some programs have multiple names—for example, the vi(1) text editor is also available as ex(1) and view(1).
- SYNOPSIS lists the possible command line options and their arguments, or how a library call is accessed. If I'm already familiar with a program but just can't remember the option I'm looking for, I find that this header is sufficient to remind me of what I need.
- DESCRIPTION contains a brief description of the program, library, or feature. The contents of this section vary widely depending on the topic, as programs, files, and libraries all have very different documentation requirements.
- OPTIONS gives a program's command line options and their effects.
- BUGS describes known problems with the code and can frequently save a lot of headaches. How many times have you wrestled with a computer problem only to learn that it doesn't work the way you'd expect under those circumstances? The goal of the BUGS section is to save you time by describing known errors and other weirdnesses.[1]
- EXAMPLES gives sample uses of the program. Many programs are very complicated, and a couple samples of how they're used clarify more than any list of options possibly can.
- HISTORY shows when the command or code was added to the system and, if it is not original to FreeBSD, where it was drawn from.
- SEE ALSO is traditionally the last section of a man page. Remember that Unix is like a language and the system is an interrelated whole. Like duct tape, the SEE ALSO links hold everything together.

If you don't have access to the manual pages at the moment, many websites offer them. Among them is the main FreeBSD website.

FreeBSD.org

The FreeBSD website (*http://www.freebsd.org/*) contains a variety of information about general FreeBSD administration, installation, and

management. The most useful portions are the Handbook, the FAQ, and the mailing list archives, but you'll also find a wide number of articles on dozens of topics. In addition to documents about FreeBSD, the website contains a great deal of information about the FreeBSD Project's internal management and the status of various parts of the Project.

Web Documents

The FreeBSD documentation is divided into articles and books. The difference between the two is highly arbitrary: as a rule, books are longer than articles and cover broader topics, while articles are short and focus on a single topic. The two books that should most interest new users are the Handbook and the Frequently Asked Questions (FAQ).

The Handbook is the FreeBSD Project's tutorial-style manual. It is continuously updated, describes how to perform basic system tasks, and is an excellent reference when you're first starting a project. I deliberately chose not to include some topics in this book because they have adequate coverage in the Handbook.

The FAQ is designed to provide quick answers to the questions most frequently asked on the FreeBSD mailing lists. Some of the answers aren't suitable for inclusion in the Handbook, while others just point to the proper Handbook chapter or article.

Several other books cover a variety of topics, such as The FreeBSD Developers' Handbook, The Porter's Handbook, and The FreeBSD Architecture Handbook.

Of the 50 or so articles available, some are kept only for historical reasons (such as the original BSD 4.4 documentation), while others discuss the subtleties of specific parts of the system, such as serial ports or building filtering bridges.

On the other hand, the official documentation is also pruned. The Handbook and FAQ cover the current FreeBSD releases, and the documentation team mercilessly prunes obsolete information. If you want to know exactly what works with current FreeBSD, go to the Handbook.

These documents are very formal, and they require preparation. As such, they always lag a bit behind the real world. When a new feature is first rolled out, the appropriate Handbook entry might not appear for weeks or months.

If the web documentation seems out of date, your best resource for up-to-the-minute answers is the mailing list archive.

The Mailing List Archives

Unless you're really on the bleeding edge, someone has probably struggled with your problem before and posted a question about it to the mailing lists. After all, the archives go back to 1994 and contain millions of messages. The only problem is that there are millions of pieces of email, any one of which might contain the answer you seek. While the FreeBSD.org website has its own search engine, you can also use any other search engine that indexes *https://lists.FreeBSD.org/*.

When reviewing the mailing list archives, be sure to check the date. The mailing list is forever. A discussion of hardware problems from 1995 might help you feel that you're part of a long history of sysadmins that have struggled with cruddy mainboards,[2] but it probably won't help you solve the issue with your brand new server. These ancient messages are basically undead documentation, rising from the grave to give you false hope. They're part of the Project's history, though, and won't be purged.

The Forums

Like many other open source projects, FreeBSD has an online forum, *https://forums.FreeBSD.org/*. A forum is much like a mailing list designed for the web, except that quite a few of us old geezers don't much care for them. You can find many good discussions and instructions on the forums, however, and they're a valuable information source.

Many people have also posted lengthy tutorials on the forums. Forum-based tutorials should properly go in the Handbook or an official article, but nobody's done the work to move them over yet. Read the discussion about such tutorials before following them; people will often point out errors or exceptions, or comment that the whole tutorial is obsolete with a newer version of FreeBSD. If you want to get involved in FreeBSD, converting these tutorials into official documentation would be a great place to start.

The forums have less of a problem with truly old information, but only because they became official in 2009. When the forums reach a quarter-century old, they'll have the same amount of undead documents. By then,

though, an even more whiz-bang discussion system will have come along—or maybe, just maybe, we'll have a better way of indexing and retrieving useful information from online discussions.

Other Websites

FreeBSD's users have built a plethora of websites that you might check for answers, help, education, products, and general hobnobbing. Almost every aggregation site such as *lobste.rs* and Reddit has a FreeBSD section, where you can get links to new posts and articles. Following those links takes you to a whole world of blogs. Also, many hosting companies include extensive FreeBSD tutorials. While these are meant for the company's customers, they're most often perfectly useful for everyone.

One of the most popular FreeBSD sites is FreshPorts, *https://www.FreshPorts.org/*. FreshPorts tracks changes to FreeBSD. Originally, it tracked changes to add-on software available via the Ports system (which I'll discuss in Chapter 16), but it quickly expanded to cover changes to the base system, the documentation, the website, and more. If you're looking to see how FreeBSD has changed, start with FreshPorts.

The FreeBSD Journal (*https://www.freebsdfoundation.org/journal/*) is a project of the FreeBSD Foundation. It's a commercial project, but your subscription fees go directly to the Foundation. Journal articles are reviewed by some of the most experienced FreeBSD developers and users, so the articles can be considered authoritative. Though as an editorial board member, I'm biased.

The FreeBSD Foundation (*https://www.freebsdfoundation.org/*) supports FreeBSD development, and I'd encourage everyone to throw a few bucks their way. I find pages like the project list useful. This lists all of the development projects that the FreeBSD Foundation has financially supported and their current state. Looking through it when writing this paragraph, I learned that FreeBSD has added wireless mesh support and multipath TCP. I don't need either of these right now, but who knows what will happen next week?

Look around, and you'll find your own favorites.

Using FreeBSD Problem-Solving Resources

Okay, let's investigate a common question with FreeBSD resources. People have asked about FreeBSD's cryptographic support for decades, so let's figure out some definitive answers about what cryptographic functions it does and does not support.

Cryptography is a complicated topic, and searching for information on it is complicated by the different ways it's referred to. It might show up as "cryptography," "cryptographic," the informal "crypto," or related words, like "encrypt." We'll try any and all of these.

Checking the Handbook and FAQ

Skimming the Handbook's table of contents brings up entries for "Encrypting Disk Partitions" and "Encrypting Swap," which certainly seem relevant. The FAQ points to these topics as well. That's a start. Those entries will guide you to appropriate man pages, which will lead you to more man pages.

Checking the Man Pages

Let's query the man pages for cryptography, using both `apropos` and `whatis`.

```
$ apropos cryptography
krb5_allow_weak_crypto, krb5_cksumtype_to_enctype...
crypto, cryptodev(4) - user-mode access to hardware-accelerated cryptography
```

This is only two entries, but the first entry is extremely long. Anything that starts with *krb5* is related to Kerberos authentication, which is a critical feature for large networks and does involve cryptography, so that's relevant. The second entry is kind of interesting, though: there's a man page for crypto(4) and cryptodev(4). (Both words point to the same man page in section 4, so the number only appears in one search result.) Let's look at the man page.

```
$ man crypto
crypto(3)                OpenSSL                crypto(3)

NAME
       crypto - OpenSSL cryptographic library
```

```
SYNOPSIS
DESCRIPTION
       The OpenSSL crypto library implements a wide range of cryptographic
       algorithms used in various Internet standards. The services provided by
       this library are used by the OpenSSL implementations of SSL, TLS and
       S/MIME, and they have also been used to implement SSH, OpenPGP, and
       other cryptographic standards.

OVERVIEW
       libcrypto consists of a number of sub-libraries that implement the
       individual algorithms.
--snip--
```

Hang on—OpenSSL is not a cryptographic device. Something obviously isn't right. Look closely at this man page; it's from section 3 of the manual, the C Libraries section. You need to search the manual for other entries containing *crypto*. Let's try the more specific `whatis(1)` search.

```
$ whatis crypto
crypto, cryptodev(4) - user-mode access to hardware-accelerated cryptography
crypto(7) - OpenCrypto algorithms
crypto, crypto_dispatch, crypto_done, crypto_freereq, crypto_freesession,
crypto_get_driverid, crypto_getreq, crypto_kdispatch, crypto_kdone, crypto_kregister,
crypto_newsession, crypto_register, crypto_unblock, crypto_unregister,
crypto_unregister_all, crypto_find_driver(9) - API for cryptographic services in the
kernel
```

Bingo! We have three crypto man pages: one in section 4, section 7, and section 9. This gives us information about the interface for programs accessing hardware cryptographic features, a list of supported algorithms, and a description of the kernel's cryptographic services. Reading these will give you a good grounding in FreeBSD's cryptography support. The SEE ALSO links in each will steer you to more information. You can now fill your brain with crypto.

Mailing Lists Archives and Forums

While the mailing lists and forums are different platforms, you search them both in similar ways. You could use the FreeBSD website search engine to search the mailing list archives, but I prefer either Google or DuckDuckGo. A search for *crypto site:lists.FreeBSD.org* spits out a whole bunch of results, as does *crypto site:forums.FreeBSD.org*.

The problem with using these sorts of discussions for general orientation on a topic is that folks plunge into nitty-gritty details. You won't get an

overview of cryptography, but you'll find details on *this* algorithm used with *that* hardware acceleration on *that* version of FreeBSD. You can get a very detailed answer if you craft a very detailed search.

Using Your Answer

Any answer you get for a question will make certain assumptions. If you're talking about cryptography, the discussion assumes you know why crypto is important, how plaintext differs from ciphertext, and what keys are. This is fairly typical of the level of expertise required for basic problems. If you get an answer that is beyond your comprehension, you need to do the research to understand it. While an experienced developer or system administrator is probably not going to be interested in explaining public key encryption, he or she might be willing to point you to a web page that explains them if you ask nicely. Always remember that people have been asking that question in relation to FreeBSD since 1994 and in relation to Unix for close to half a century.

Asking for Help

When you finally decide to ask for help, do so in a way that allows people to actually provide the assistance you need. No matter whether you prefer email or a forum, you must include all the information you have at your disposal. There's a lot of suggested information to include, and you might think you can skip some or all of it. If you slack off and fail to provide all the necessary information, though, one of the following things will happen:

- Your question will be ignored.
- You'll receive a barrage of email asking you to gather this information.

On the other hand, if you actually want help solving your problem, include the following pieces of information in your message:

- A complete problem description. A message like *How do I make my cable modem work?* only generates a multitude of questions: What do you want your modem to do? What kind of modem is it? What are the symptoms? What happens when you try to use it? How are you trying

to use it?

- The output of `uname -a`. This gives the operating system version and platform.
- Any error output. Be as complete as possible, and include any messages from the console or from your logs, especially */var/log/messages* and any application-specific logs. Messages about hardware problems should include a copy of */var/run/dmesg.boot*.

It's much better to start with a message like "My cable modem won't connect to my ISP. The modem is a BastardCorp v.90 model BOFH667. My OS is version 12.2 on a quad-core Opteron. Here's the contents of */var/log/messages* and */var/run/dmesg.boot* from when I try to connect. When I manually run dhclient, I get these messages." You'll skip a whole round of discussions with a message like this, and you'll get better results more quickly.

Composing Your Message

First, *be polite*. People often say things online that they wouldn't dream of saying to someone's face. These lists are staffed by volunteers who are answering your message out of sheer kindness. Before you click that Send or Submit button, ask yourself, *Would I be late for my dream date to answer this message?* The fierce attitude that is occasionally necessary when working with corporate telephone-based support only makes these knowledgeable people delete your emails unread or flat-out block your account on the forum. Their world doesn't have to include surly jerks. Screaming until someone helps you is a valuable skill when dealing with commercial software support, but it will actively hurt your ability to get support from any open source project.

No matter whether you choose the forums or email, stay on topic. If you're having a problem with *X.org*, check the *X.org* website. If your window manager isn't working, ask the people responsible for the window manager. Asking the FreeBSD folks to help you with your Java Application Server configuration is like complaining to industrial machinery salespeople about your fast-food lunch. They might have an extra ketchup packet, but it's not really their problem. On the other hand, if you want your FreeBSD system to no longer start the mail system at boot time, that's a FreeBSD issue.[3]

Sending Email

FreeBSD developers tend to use mailing lists, not the forums. This means that the mailing lists can get you attention from people who know more about the system, but it also means that you need to follow the etiquette for that environment.

Send plaintext email, not HTML. Many FreeBSD developers read their email with a text-only email program, such as Mutt. Such programs are very powerful tools for handling large amounts of email, but they do not display HTML messages without contortions. To see for yourself what this is like, install */usr/ports/mail/mutt* and read some HTML email with it. If you're using a graphic mail client, such as Microsoft Outlook, either send your email in plaintext or make sure that your messages include both a plaintext and an HTML version. All mail clients *can* do this; it's just a question of discovering where your GUI hides the buttons. What's more, be sure to wrap your text at 72 characters. Sending HTML-only email or email without decent line-wrapping is an invitation to have your email discarded unread.

Harsh? Not at all, once you understand whom you're writing to. Most email clients are poorly suited to handling thousands of messages a day, scattered across dozens of mailing lists, each containing a score of simultaneous conversations. The most popular email clients make reading email easy, but they do not make it efficient; when you get that much email, efficiency is far more important than ease. As most people on those mailing lists are in a similar situation, plaintext mail is very much the standard for them.

Top-posting replies to an email is discouraged. Make any comments inline with the discussion to retain context.

On a similar note, most email attachments are unnecessary. You do not need to use OpenPGP on messages sent to a public mailing list, and those business-card attachments just demonstrate that you aren't a system administrator. Don't use a long email signature. The standard for email signatures is four lines. That's it—four lines, each no longer than 72 characters.[4] Long ASCII art signatures are definitely out.

When you've composed your nicely detailed and polite question, send it to *FreeBSD-questions@FreeBSD.org*. Yes, there are other FreeBSD mailing lists, some of which are probably dedicated to what you're having trouble

with. As a new user, however, your question is almost certainly best suited to the general questions mailing list. I've lurked on many of the other mailing lists for a decade now and have yet to see a new user ask a question on any of them that wouldn't have been better served by *FreeBSD-questions*. Generally, the questioner is referred back to *FreeBSD-questions* anyway. If your question needs to be asked elsewhere, someone will tell you.

This goes back to the first point about politeness. Sending a message to the architectural mailing list asking about what architectures FreeBSD runs on is only going to annoy the people who are trying to work on architectural issues. You might get an answer, but you won't make any friends. Conversely, the people on *FreeBSD-questions* are there because they're volunteering to help people just like you. They want to hear your intelligent, well-researched, well-documented questions. Quite a few are FreeBSD developers, and some are even Core members. Others are slightly more experienced users who have transcended what you're going through now and actively want to give you a hand up.

Forum Posting

The forums have a different population than the mailing lists. Some developers hang out there, but not as many as on the mailing lists. The people on the forums are actively interested in helping you with your problems, though.

Forums are somewhat easier than the mailing lists. You can post only in formats the website supports. There are no concerns about top-posting versus inline posting or unreadable HTML. This ease is part of their popularity. If you get deeper into FreeBSD, though, you'll eventually want to join mailing lists.

But no matter which venue you choose, politeness is vital.

Responding to Email

Your answer might be a brief note with a URL or even just two words: *man such-and-such*. If that's what you get, that's where you need to go. Don't ask for more details until you've actually studied that resource. If you have a question about the contents of the reference you're given, or if you're confused by the reference, treat it as another problem. Narrow down the

source of your confusion, be specific, and ask about that. Man pages and tutorials are not perfect, and some parts appear contradictory or mutually exclusive until you understand them.

Finally, follow through. If someone asks you for more information, provide it. If you don't know how to provide it, learn how. If you develop a bad reputation, nobody will want to help you.

The Internet Is Forever

Those of us who were on the internet back in the '80s remember when we treated it as a private playground. We could say whatever we wanted, to whomever we wanted. After all, it was purely ephemeral. Nobody was keeping this stuff; like CB radio, you could be a total jackass and get away with it.

All those early Usenet discussions? Yeah, Google recovered them and put them online. Our beliefs were the exact opposite of true.

Potential employers, potential dates, even family members might scan the internet for your postings to mailing lists or message boards, trying to learn what sort of person you are. I've rejected hiring more than one person based on their postings to mailing lists and discussion boards. I want to work with a system administrator who sends polite, professional messages to support forums, not childish and incoherent rants without sufficient detail to offer any sort of guidance. And I'd think a lot less of my in-laws if I stumbled across a message from one of them on some message board where they acted like fools. FreeBSD discussions are widely archived; choose your words well, because they will haunt you for decades.

Now that you know how to get more help when things go wrong, let's install FreeBSD.

2
BEFORE YOU INSTALL

Getting FreeBSD running on your computer isn't enough, no matter how much that first install might satisfy you. It's just as important that your install be *successful*. A successful install is one that works for its intended purpose. Servers have very different requirements than desktops, and a server's intended function can completely change installation requirements. Proper planning before installing FreeBSD makes installations much less painful. On the downside, you'll get much less experience in reinstalling FreeBSD because you'll do each install only once. If mastering the installation program through exhaustive repeated practice is your main goal, skip this boring "thinking ahead" stuff and read the next chapter.

I'm assuming that you want to run FreeBSD in the real world, doing real work, in a real environment. This environment might be your laptop—while you might argue that your laptop isn't a production system, I challenge you to erase all the data on it without backing up and then tell me it's not a production system. If you're installing on a system intended for destructive testing, and you're truly indifferent to its fate, I still recommend following best practices so that you develop good habits.

Consider what hardware you need or have. Then decide how best to use that hardware, what filesystem you should use, and how to arrange your disks. Only then should you proceed to downloading and installing FreeBSD.

Before you even start the install, though, let's look at a couple concepts you'll hit throughout your FreeBSD experience: default files and universal configuration language (UCL).

First, however, you must understand FreeBSD's default configuration filesystem.

Default Files

FreeBSD separates configuration files into default files and customization files. The *default files* contain variable assignments and aren't intended to be edited; instead, they're designed to be overridden by another file of the same name.

Default configurations are kept in a directory called *default*. For example, the boot loader configuration file is */boot/loader.conf*, and the default configuration file is */boot/defaults/loader.conf*. If you want to see a comprehensive list of loader variables, check the default configuration file.

During upgrades, the installer replaces the default configuration files but doesn't touch your local configuration files. This separation ensures that your local changes remain intact while still allowing new values to be added to the system. FreeBSD adds features with every release, and its developers go to great lengths to ensure that changes to these files are backward compatible. This means that you won't have to go through the upgraded configuration and manually merge in your changes; at most, you'll have to check out the new defaults file for nifty configuration opportunities and new system features.

The loader configuration file is a good example of these files. The */boot/defaults/loader.conf* file contains dozens of entries much like this:

```
verbose_loading="NO"            # Set to YES for verbose loader output
```

The variable `verbose_loading` defaults to `NO`. To change this setting, do not edit */boot/defaults/loader.conf*—instead, add the line to */boot/loader.conf* and change it there. Your */boot/loader.conf* entries override the default setting, and your local configuration contains only your local changes. A sysadmin can easily see what changes have been made and how this system differs from the out-of-the-box configuration.

I encourage you to keep your configuration files in a version control system. If you have a global configuration management system like Ansible, that's grand. Without such a system, a centralized repository using svn(1) or

the loved-or-loathed git(1) will do. Even local revision control systems like rcs(1) can one day save your hide.

DON'T COPY THE DEFAULT CONFIG!

One common mistake is to copy the default configuration to the override file and then make changes there directly. Such copying will cause major problems in certain parts of the system. You might get away with it in one or two places, but eventually it will bite you. Copying */etc/defaults/rc.conf* to */etc/rc.conf*, for example, will prevent your system from booting. You have been warned.

The default configuration mechanism appears throughout FreeBSD, especially in the core system configuration.

Configuration with UCL

The *universal configuration language*, or *UCL*, is a common library for managing Unix-style configuration files. FreeBSD uses UCL for core functions, such as the packaging system.

Any file that is in UCL can appear in one of several formats, such as the traditional variable = setting format most Unix programs use, YAML, or JSON. If you've configured any Unix software before, UCL won't be a problem.

We'll see examples of UCL-style configuration throughout this book. You don't need to know the details of UCL at this time, merely that UCL is a thing in FreeBSD.

FreeBSD Hardware

FreeBSD supports a whole bunch of hardware, including different architectures and devices designed for each architecture. One of the Project's

goals is to support the most widely available hardware, and that list of hardware includes far more than the "personal computer." Today's fully supported *Tier 1* hardware includes 32-bit and 64-bit versions of the Intel-style processor.

Most modern hardware uses 64-bit extensions to Intel's classic 32-bit architecture. These extensions were created by AMD, and so the platform is called *amd64*. Most hardware built in the last decade uses the amd64 standard. While amd64 hardware will boot both 32-bit and 64-bit versions of FreeBSD, the 32-bit version contains a bunch of workarounds to support the hardware's features and expanded address space. Run 64-bit FreeBSD on 64-bit hardware.

The traditional 32-bit IBM-compatible PC dominated computing for decades. FreeBSD supports that hardware with the *i386* platform.[1] Use the i386 version of FreeBSD only on pure 32-bit hardware. FreeBSD offers limited support for a few other hardware platforms, calling them *Tier 2 architectures*. Some of these are increasingly popular, such as ARM. FreeBSD supports both 32-bit and 64-bit ARM CPUs with the *arm* and *arm64* platforms. Support for 64-bit ARM hardware is improving rapidly, and you can expect ARM64 to become a Tier 1 platform soon. Other hardware platforms are on their way out and have been demoted to Tier 2 before being removed from the source tree. Additionally, you can run FreeBSD on PowerPC (*ppc*) and 64-bit Sparc (*sparc64*) hardware, which never made it up to Tier 1. Temporary breakage of bleeding-edge FreeBSD is acceptable on Tier 2 platforms. Tier 2 platforms might or might not have packages available.

You'll also find *Tier 3* platforms, which are highly experimental. RISCV hardware is at Tier 3.

Tier 4 includes barely supported platforms. Some of them are long obsolete and on their way out. The code still exists and could theoretically be resurrected, but nobody cares enough to do the work. Others might be on their way in but are not yet fully developed. Every platform that reaches a higher tier passes through Tier 4 on its way up.

FreeBSD supports many network cards, hard drive controllers, and other peripherals for each architecture. As many of these architectures use similar interfaces and hardware, this isn't as much of a challenge as you might think: SATA is SATA anywhere, and an Intel Ethernet card doesn't magically

transform when you put it in an arm64 machine.

While FreeBSD runs just fine on ancient hardware, that hardware must be in acceptable condition. If your Pentium IV crashes because it has bad RAM, installing FreeBSD won't stop the crashes.

FreeBSD supports most RAID controllers and includes software to manage most of them. However, I would encourage folks running the UFS filesystem to use FreeBSD's RAID options rather than a hardware RAID controller. RAID controllers were created when managing storage redundancy was so computing intensive that it monopolized the host's processor. Today's computing hardware manages RAID without breaking a sweat. Additionally, RAID controllers use custom formats on hard drives. Often, the only device that can read those disks is another RAID controller of the exact same model. The unexpected demise of a RAID controller can leave you trawling dubious internet auctions in search of old controllers. And if you think those controllers are expensive new, wait until they're five years old and the only folks willing to buy them are those truly desperate for that exact model! FreeBSD has a few different options for software RAID, and those disks can be read with any similar hardware.

If you're using ZFS, the warnings against RAID controllers become "just don't." ZFS expects to have direct access to the disks. Using a RAID controller disables much of ZFS's self-healing and error-correction abilities. If you must use a RAID controller, disable RAID and have it serve as a storage controller. While many RAID cards claim they can act as a RAID controller, most actually serve up a bunch of one-drive RAID containers. Verify that your RAID controller can be shifted to just-a-bunch-of-disks (JBOD) or host-bus-adapter (HBA) mode before deploying ZFS on it.

This book uses amd64 as a reference platform. Everything should work on a 32-bit i386 host, but amd64 is the world's standard these days, so we'll use it. The test systems include a couple of iXsystems storage servers and a variety of virtual machines.[2]

Proprietary Hardware

Some hardware vendors believe that keeping their hardware interfaces secret prevents competitors from copying their designs and breaking into their market. This has repeatedly been demonstrated to be terrible strategy,

especially as the flood of generic parts has largely drowned these secretive hardware manufacturers. A few vendors still cling to their secrecy, however. We call such devices *proprietary hardware*.

Developing device drivers for a piece of hardware without its interface specifications is quite difficult. Some hardware can be well supported without full documentation and is sufficiently common to make struggling through this lack of documentation worthwhile.

If a FreeBSD developer has a piece of hardware, documentation for that hardware, and interest in that hardware, he'll probably implement support for it. If not, that hardware won't work on FreeBSD. In most cases, unsupported proprietary hardware can be easily replaced with less expensive and more open options.

Some vendors provide closed-source binary drivers for their hardware in the form of kernel modules (see Chapter 6). Remember that while FreeBSD refers to the kernel as modular, that means that you can choose which parts to load and which to leave out. Once a kernel module is loaded, that module has complete access to the entire kernel. It's entirely possible for a video driver kernel module to corrupt your filesystem. I strongly encourage you to avoid binary drivers whenever possible, and to avoid hardware that requires such drivers.

IS MY HARDWARE SUPPORTED?

The easiest way to determine whether a piece of hardware is supported is to boot FreeBSD on it. If you don't have physical access to the hardware yet, check *https://www.FreeBSD.org/* for the release notes for your chosen version.

Hardware Requirements

Once upon a time, a host's minimal hardware requirements were a big deal. FreeBSD 1.0 supported very specific hard drive controllers and Ethernet adapters, and needed several megabytes of RAM. Hardware that couldn't run FreeBSD was still in common use back then.

Most hardware requirements are a thing of the past. Any amd64 system ever produced can run FreeBSD. Any server-grade i386 system built this millennium can run FreeBSD. Yes, a Pentium with a meager 18GB SCSI-2 disk and a paltry 128MB of RAM offers mediocre performance, but if you want good performance, try not using that hardware.

Just because a piece of hardware should work doesn't mean it will work. "Inexpensive" is not the same as "cheap." Supported lousy hardware is still lousy. Research your hardware before buying it.

FreeBSD runs fine on hypervisors, such as VMware, VirtualBox, Xen, and KVM. Legitimate cloud providers offer FreeBSD images and ISOs. FreeBSD runs just fine on the integrated bhyve(8) hypervisor and OpenBSD's vmm(8). You can do a base install with 128MB of RAM and 1GB of disk, although you'll probably want more than that for serious experimentation.

BIOS versus EFI

Back in the 1980s, IBM invented the *basic input/output system (BIOS)* to handle low-level hardware tasks, like finding the operating system. Generations of IT people have argued with the BIOS. BIOS had built-in limitations that keep it from working well on modern hardware, though. The modern BIOS-like thing is called the *Extensible Firmware Interface (EFI)*. EFI is far more flexible and powerful than the BIOS. FreeBSD boots just fine from EFI, and using EFI permits FreeBSD to do some interesting things, like full-disk encryption.

If your hardware supports EFI, use it. Only fall back to BIOS mode if FreeBSD exposes a bug in your hardware's EFI implementation, in which case I'd encourage you to file a bug (see Chapter 24). Note that the hardware setup utility might call BIOS mode "legacy boot" or "ancient crap" or some such thing.

Disks and Filesystems

Perhaps the most critical part of installing a system is how you allocate disk space and which filesystem you use. A base install of FreeBSD fits in about half a gigabyte of disk, but the filesystem beneath those files dictates much of

how the system behaves.

FreeBSD Filesystems

FreeBSD supports two major filesystems, UFS and ZFS. Which should you use? That depends entirely on what you want to do with your system. To make a decision before booting your install media, you'll need to understand the basics of each.

FreeBSD's *Unix File System (UFS)* is a direct descendant of the filesystem shipped with 4.4 BSD and has been under continuous development for decades. One of UFS's original authors still hangs around the FreeBSD community actively improving the filesystem, as well as offering support and guidance to newer generations of developers. UFS's place as the primordial FreeBSD filesystem has let it extend fingers throughout the operating system. Many other FreeBSD filesystems attach to the kernel's virtual memory system through infrastructure created for UFS. UFS is designed to handle the most common situations effectively while reliably supporting unusual configurations. FreeBSD ships with UFS configured to be as widely useful as possible on modern hardware, but you can choose to optimize a partition for trillions of tiny files or a handful of 1TB files if you desire.

ZFS (not an acronym) was introduced by Solaris in 2005 and integrated into FreeBSD in 2007. Its youth seems to be a disadvantage, but it combines technologies and concepts that have been used for much longer. ZFS computes a checksum of every block of data or metadata and can use it for error correction. Storage is pooled, meaning that you can dynamically add more disks to an existing ZFS filesystem without recreating the filesystem. ZFS has a whole bunch of cool features, such as highly effective built-in replication and the ability to create and remove datasets (partitions) on the fly.

While ZFS was written over a decade ago, it was written for future hardware. All of those cool features impose a performance cost, and ZFS can use a whole bunch of memory. While 32-bit systems can use ZFS, it's not recommended. I resist running ZFS on hosts with less than 4GB of RAM and refuse to run it on less than 2GB of RAM. UFS serves small and embedded systems better than ZFS can.

ZFS makes a great storage system for a virtualization server, but it isn't

necessarily right for virtual machines that use disk images. Many virtual machines don't get enough memory to effectively run ZFS. Additionally, I've seen more than one KVM-based virtualization system fail to migrate ZFS-based virtual machines. If you want to use ZFS on virtualized clients, be sure your virtualization system supports restoring and migrating ZFS disk images before installing a slew of hosts.

Some people insist that ZFS requires ECC RAM. ECC RAM is good, and you should get it if you can. ZFS without ECC is no worse than UFS *with* ECC, however. ECC provides a layer of integrity checks much like ZFS. If a host's non-ECC memory gets hit by a cosmic ray, ZFS writes corrupt data to disk—just as if you used UFS.

Finally, ZFS assumes you're doing things the ZFS way. ZFS is a combination filesystem and volume manager. It expects access to raw disks. Never, never, *never* use a RAID controller with ZFS; using RAID volumes as disks interferes with ZFS's self-healing features. Many RAID controllers claim to offer raw disks, but what they really offer are one-disk RAID containers.[3]

UFS isn't perfect either. A power failure or system crash can damage a UFS filesystem. Repairing that filesystem takes time and system memory. Roughly speaking, repairing each terabyte in a UFS filesystem requires 700MB of RAM. If you create a 7TB filesystem on a system with 6GB of RAM, FreeBSD can't automatically repair it.

To boil this all down, on a modern amd64 laptop or a server, I recommend ZFS. Test ZFS with your virtualization system. If it works, use ZFS for 64-bit virtual machines with 4GB of RAM or greater. On i386 hardware or 64-bit hosts with less than 4GB of RAM, use UFS.

If you're running a high-load, high-volume application and database, experiment with both UFS and ZFS on your production hardware to see which works better in your application before proceeding. Experiment with different arrangements of disks, ZFS pool types, and GEOM RAID methods. Some applications work better with UFS than ZFS. Netflix, for example, delivers all of its content from FreeBSD hosts with massive amounts of storage formatted with UFS. Before installing your massive storage server, review Chapter 12 for additional ZFS deployment considerations.

All this advice is secondary to an iron rule: choose the filesystem that best

suits your environment.

Filesystem Encryption

Disk encryption has become a vital feature for many environments. A user that loses his laptop doesn't want to lose his data. Certain organizations require that critical data be encrypted on resting, or inactive, disks. You can't retroactively encrypt a disk on an installed system.

FreeBSD supports two disk encryption systems: *GEOM-Based Disk Encryption (GBDE)* and *GELI*. The gbde(8) encryption system is designed for use in situations where the mere existence of encrypted data can threaten the user's life. It's designed to protect a user who has a gun to their head. Thankfully, that use case is rare; this book doesn't cover it.

The geli(8) encryption system protects against more common risks. If your laptop is stolen, GELI prevents the thief from reading the hard drive.[4] If you store your company's financial records on a GELI-encrypted partition, the service tech can't read it during a service call. Chapter 23 covers GELI in more detail.

Many organizations require disks containing financial data or intellectual property to be rendered unreadable when decommissioned. You can send such disks to be shredded, but encrypting the disks at install time is equally effective. The disks become unreadable when you destroy the encryption key.

I recommend encrypting either the entire system or none of the system. Partially encrypted disks leave opportunities for skilled intruders to sabotage your system and subvert the encryption.

Decide whether or not you need encryption before proceeding.

Disk Partitioning Methods

Disk *partitioning* lets you divide a disk or disk array into logical units. Even hosts with average consumer-grade operating systems, such as the Windows laptop you'll find at your local big-box store, ship with multiple partitions on the hard drive. A *partitioning scheme* is the system for organizing partitions on a disk.

Computing is always in transition between technologies, and right now

we're amidst a particularly annoying change in disk partitioning. Older and smaller hardware uses master boot record (MBR) partitioning and is always limited to disks of 2TB or smaller. Newer and larger hardware uses the more flexible and generally better GUID Partition Tables (GPT) scheme. FreeBSD manages both types of partition with gpart(8).

Which should you use in your install? Use GPT on any system that supports GPT, no matter the size of the disk. Use MBR if and *only* if the system can't support GPT. (You can use gptboot(8) and gptzfsboot(8) to bludgeon GPT support onto MBR-only disks, but save that for your second or third install.)

I've encountered more than one system that supports GPT but has a hardware limitation that prevents it from using disks larger than 2TB. While MBR might seem sensible on such a system, remember that GPT is far more flexible. Even if you're a sysadmin with decades of experience with MBR, learn and use GPT.

Partitioning with UFS

If you decide to use UFS for your host, you'll need to consider filesystem partitioning. Thanks to the wide variety of disk sizes FreeBSD supports, the installer doesn't attempt to predict how you'll want to partition your system. Decide how to partition the disk before installing.

At a minimum, separate your operating system from your data. If this host is for user accounts, create a separate */home* partition. If you're running a database, create a partition for the database. Web servers should have a partition for web data and probably a second one for logs.

As an old Unix hand, I usually create separate */usr*, */usr/local*, */var*, */var/log*, and */home* partitions, as well as a partition for root (/) and one for swap space, plus a separate partition for the server's application data. I'm told that I'm a fuddy-duddy, though, and that my concerns about rogue processes and users filling up the hard drive are obsolete these days.[5]

A base install of modern FreeBSD fits in about half a gigabyte. That's trivial next to today's hard drives. On a modern disk running on real hardware, assigning 20GB for the operating system and related programs should be more than sufficient.

If you're running FreeBSD on modern hardware, though, you probably

want to use ZFS rather than UFS.

Multiple Operating Systems

Back in the Stone Age (roughly 2001), being able to install four operating systems on a single 6GB hard drive thrilled me. This was the only way to run multiple operating systems on a desktop without swapping hard drives.

It's still possible to do multiboot installations, but virtualization is far better. You don't have to shut down your main operating system to access one of the other operating systems. The bhyve(8) hypervisor lets you run other operating systems, including Microsoft Windows, on top of FreeBSD. Other operating systems have hypervisors that let you run FreeBSD on top of them.

Multiple Hard Drives

If you have multiple hard drives in your host, you should almost certainly use them to create some sort of storage redundancy. If you're using ZFS, use a mirror or some sort of RAID-Z (see Chapter 12). If you use UFS, FreeBSD supports software RAID. When you have a whole bunch of hard drives, though, life gets a little more complicated.

The rule of thumb is still to separate your operating system from your application data. If you have 30 hard drives, mirror 2 of them for your operating system install and use the others for your data. Like all rules of thumb, this is debatable. But no sysadmin will tell you that this is an actively bad idea.

With many hard drives, consider which data passes through which disk controller. If a disk controller dies, what happens to your system? If both of your operating system disks are attached to a single controller and the controller dies, your host goes down. Putting each drive on a different controller offers redundancy. Ideally, attach your mirrored operating system disks to different drive controllers.

Also, remember that SATA disk controllers split all their data throughput among all the hard drives connected to them. If you have two disks on a SATA controller, each disk works, on average, about half as fast as it would work alone on the same channel. Port multipliers add disks but slash per-disk performance.

Swap Space

When FreeBSD (and any other modern operating system) uses up all the physical RAM, it can move information that's been sitting idle from memory into swap. Now that even laptops ship with 32GB of RAM, it's hard to imagine a host running out of memory, but never underestimate a program's ability to devour RAM. Virtual systems might be allocated very tiny amounts of RAM.

So, how much swap space do you need? This is a matter of long debate between sysadmins. The short answer is, "It depends." What does it depend on? *Everything*. Long-running wisdom claimed that a host should have twice as much swap as it has physical memory, but today that's not only obsolete but dangerous. When a process starts catastrophically allocating memory—say, in a bug caused by an infinite loop—the kernel kills the process once the system runs out of virtual memory. A system with 32GB of RAM and 64GB of swap has 96GB of virtual memory. The i386 platform limits memory usage to 512MB per process, which means that the kernel stops such runaway processes pretty quickly. 64-bit systems, like amd64, have vast virtual memory spaces. A system thrashing gigabytes of memory between disk and RAM will be excruciatingly slow. A modern host should have only enough swap space to perform its task.

Multiple hard drives let you increase the efficiency of swap space by splitting it between disks on different drive controllers. Remember, though, that a crash dump must fit entirely within a single swap partition. FreeBSD compresses crash dumps so that they don't take up as much room, but still, many small swap partitions can be counterproductive. If you have a large number of drives, don't use the application drives for swap; restrain swap space to the operating system drives.

The main use for swap on modern systems is to have a place to store a memory dump should the system panic and crash. FreeBSD uses kernel minidumps, so they dump only the kernel memory. A minidump is much smaller than a full dump: a host with 8GB RAM has an average minidump size of about 250MB. Provisioning a gigabyte of swap per 10GB of RAM should be sufficient for most situations.

If you have a truly intractable problem, though, you might need to dump the entire contents of your RAM to swap. If I'm setting up an important production system, I always create an unused partition larger than the host's

greatest possible virtual memory space and tell the host to dump the kernel to that partition. If my laptop has such a problem, I'll just plug in a flash drive and configure the system to dump on it instead.

Getting FreeBSD

Now that you've made all your decisions, you need a copy of FreeBSD. If this is your first time installing FreeBSD, go to *https://www.FreeBSD.org/* and look for the Get FreeBSD section at the top. Right by that, you'll see a list of supported releases, including (probably) two releases recommended for production. Sometimes there's one. Sometimes there's three, but usually two.

FreeBSD Versions

Two production releases? What madness is this?

FreeBSD development occurs in multiple tracks, as I will discuss in Chapter 18. A few tracks coexist, in various states of support. Each track receives bugfixes and incremental improvements. Newer tracks get new features.

As I write this, FreeBSD.org lists two production releases, numbered 11.0 and 10.4. Version 11.0 is the most recently released version, but it's also a .0 release. It's the first release on this track. It will have the newest features, but it has the greatest likelihood of including unknown bugs. Version 10.4 is slightly older and lacks some features in version 11.0, but it's the fourth release along that track. It's not guaranteed to be bug free, but many people have run it in production for months or years. Any screamingly obvious problems have been fixed.

Every FreeBSD release eventually reaches *End of Life (EoL)* and loses support. The security team stops producing patches and new packages are no longer available. The older release will reach EoL before the newer version. If you install FreeBSD 10.4 today, you'll need to upgrade to 11 at some point—but by then, you'll be upgrading to something that's not a .0 release.

FreeBSD averages two production releases at a time. This isn't an inviolate rule, only observed behavior. Sometime around when the 12.0 release escapes, the 10 branch will reach EoL.

I personally will run FreeBSD .0 releases, but having been burned with other operating systems before, I sympathize with the folks who categorically reject .0 versions. If you've never used FreeBSD before, I recommend installing the most recent production release. It has the latest device drivers and newest features.

Follow the download link and grab your chosen version.

Choosing Installation Images

You can choose between several different formats of FreeBSD installation media. All installation media is available both compressed with xz(1) and uncompressed. If you can conveniently extract *.xz* files, download the compressed versions. This saves the donated bandwidth and reduces download time. Any modern operating system can either handle *.xz* files natively or has add-on software for the task.

FreeBSD offers two styles of installation media. The first contains only enough to boot the FreeBSD installer and bring up the network. The installer then downloads the operating system files from a FreeBSD mirror site. If you're going to do multiple installs of the same FreeBSD version, though, you're better off downloading an installer that includes the operating system files.

The installer comes in both optical disk (*.iso*) and flash (*.img*) formats. Choose the format that fits your system. If you're installing a virtual machine, an ISO is probably simplest.

FREEBSD MIRRORS

Old documents make much of the importance of choosing a good mirror site for installation. Ignore all that. The FreeBSD software distribution site, *ftp.freebsd.org/*, is a worldwide collection of mirror servers. When you grab the installation media, packages, or any other FreeBSD materials, you're automatically directed to the closest mirror site. If you want to use a specific mirror rather than the GeoDNS-selected one, choose it by name from the list in Appendix A of

the FreeBSD Handbook (discussed in Chapter 1).

Each installer image starts with the word *FreeBSD*, the release, and the platform. If you're downloading FreeBSD 12.0 for amd64 hardware, the installer images will all have names that start with *FreeBSD-12.0-RELEASE-amd64*. Right after that, the file identifies the installation type.

File endings are a tool to help you easily find what you need:

- Files ending in *bootonly.iso* are ISO images that boot the FreeBSD installer. Using them means downloading FreeBSD over the network.
- Files ending in *disc1.iso* are ISO images that contain the full FreeBSD installer. This image contains the operating system files.
- Files that end in *mini-memstick.img* are for flash drives. They boot the FreeBSD installer but download the operating system files over the network.
- Files that end in *memstick.img* are flash drive images that contain a complete FreeBSD install.

FreeBSD also provides much larger DVD images. These contain all of FreeBSD and a whole bunch of packages. They're meant for people who want to use FreeBSD without internet access. Please remember that all of the FreeBSD Project's bandwidth is donated; don't download a massive DVD image unless you actually need it.

Once you have an installation image, you need to get it on actual boot media. Use your operating system's built-in tools to burn the image to a physical disk. While Windows now includes CD burning as a built-in feature, it doesn't include flash disk imaging. The FreeBSD Project recommends Image Writer for Windows (*https://sourceforge.net/projects/win32diskimager/*), a perfectly fine option. Bring up the program, select your flash drive and the image, and click **Start**.

Network Installs

If your installation media only boots the installer and you need to grab the FreeBSD distribution files over the network, you'll need to configure the

network while the installer is running. If your network runs DHCP, the installer should just pick up your network configuration. If not, your FreeBSD host will need a valid network configuration. Before starting the installer, gather:

- A valid IP address and netmask
- The default gateway for your network
- The nameserver IP addresses

If you must use a proxy server to reach the internet, you'll need its configuration as well.

Armed with this information, you can install FreeBSD.

3
INSTALLING

You've thought about what you want your FreeBSD install to do. You've chosen hardware. You've downloaded boot media and burned it to a flash drive or optical disk. You've found a working USB keyboard and set up your test machine to boot from that media. Now let's walk through a FreeBSD install. Boot up your install media and follow along.

Throughout this walkthrough, I'll mention the various key mappings, quirks, and shortcuts the installer provides. One annoyance is that the installer offers no back button: if you screw up something basic, like the disk partitioning, start over.

My desktop, of course, has been installed and running for years. I've somehow been coerced to setting up a system for Bert,[1] though. If he doesn't like my installation methods, he can read this chapter and install his own dang machines.

Core Settings

Upon booting the install media, I see the boot loader screen with its 10-second countdown, as shown in Figure 3-1.

Figure 3-1: The boot loader

Hitting ENTER skips the 10-second counter.

I then get the selection menu shown in Figure 3-2.

Figure 3-2: Selecting Install

In Chapter 5, we'll discuss using the live CD option to repair damaged

systems. For right now, choose Install (the default) by pressing ENTER.

You might notice that the first letter of each choice is in red, while most of the text is gray. You can type that letter to make a choice rather than arrowing over. Here, entering S takes you to a shell, while L starts the live CD image.

You're now entering bsdinstall(8), FreeBSD's old-fashioned installer. While other operating systems have pretty graphical installers with mouse-driven menus and multicolor pie charts, FreeBSD's looks like an old DOS program. You'll start your install by choosing a keymap, as shown in Figure 3-3.

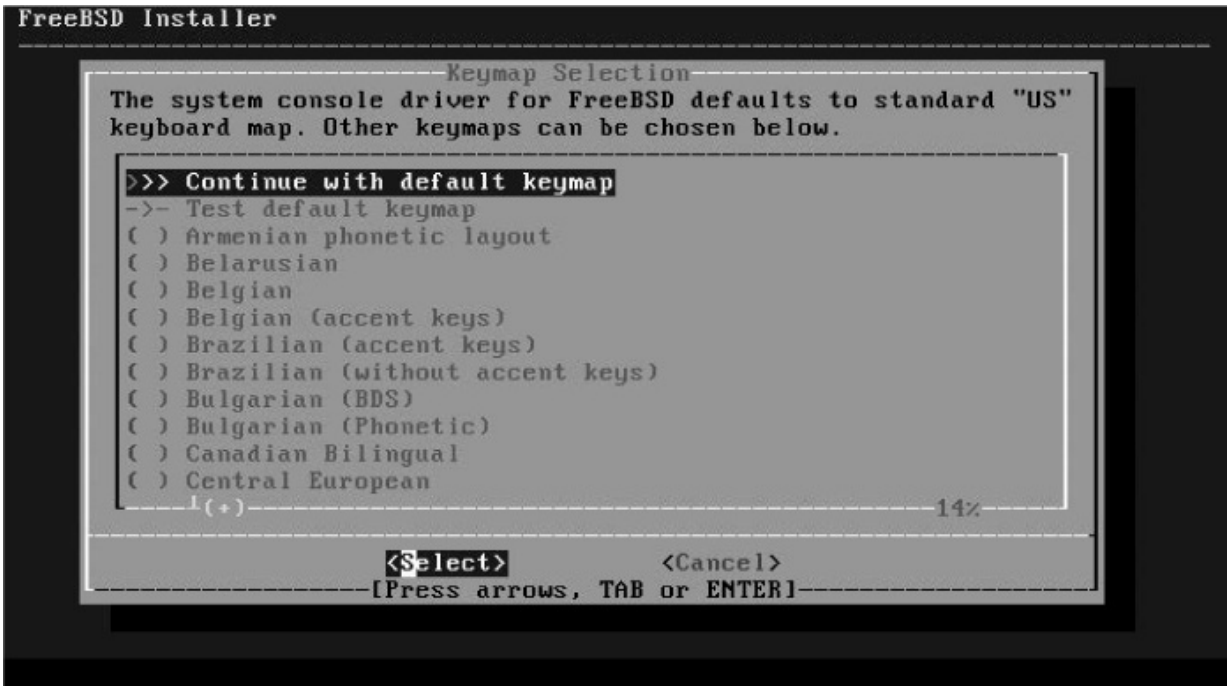


Figure 3-3: Keymap selection

Bert's typing habits are atrocious, and he really needs a better keyboard layout. You can arrow up and down this list, but that's slow. PAGEUP and PAGEDOWN take you up and down a whole screen at a time, while HOME and END take you to the top and bottom, respectively. When I find a keymap I like, I press ENTER. I can then test the keymap, as shown in Figure 3-4.

The keymap looked familiar, but many keymaps have similar names. Hitting ENTER brings up a field where I can hammer on the keyboard to test whether the keymap fits what I think I picked. If it looks good, ENTER brings me back to this screen, where I can hit the up arrow and ENTER to proceed.

The installer then asks me for a hostname, as Figure 3-5 shows.

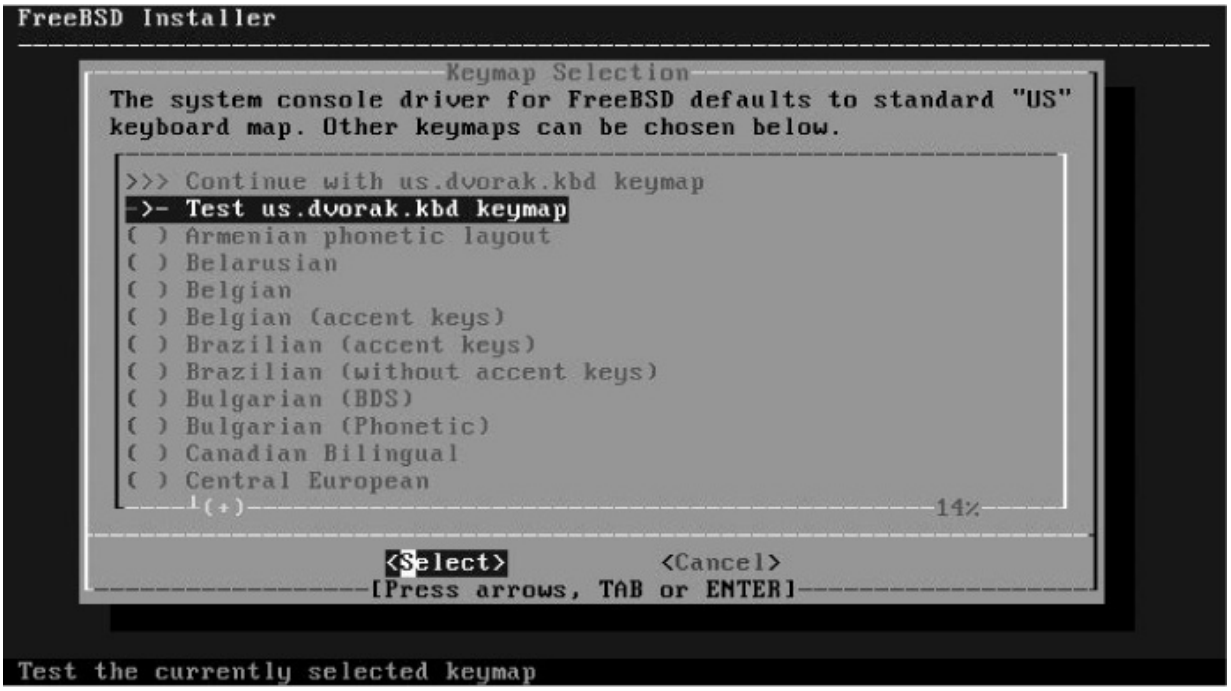


Figure 3-4: To test or not?

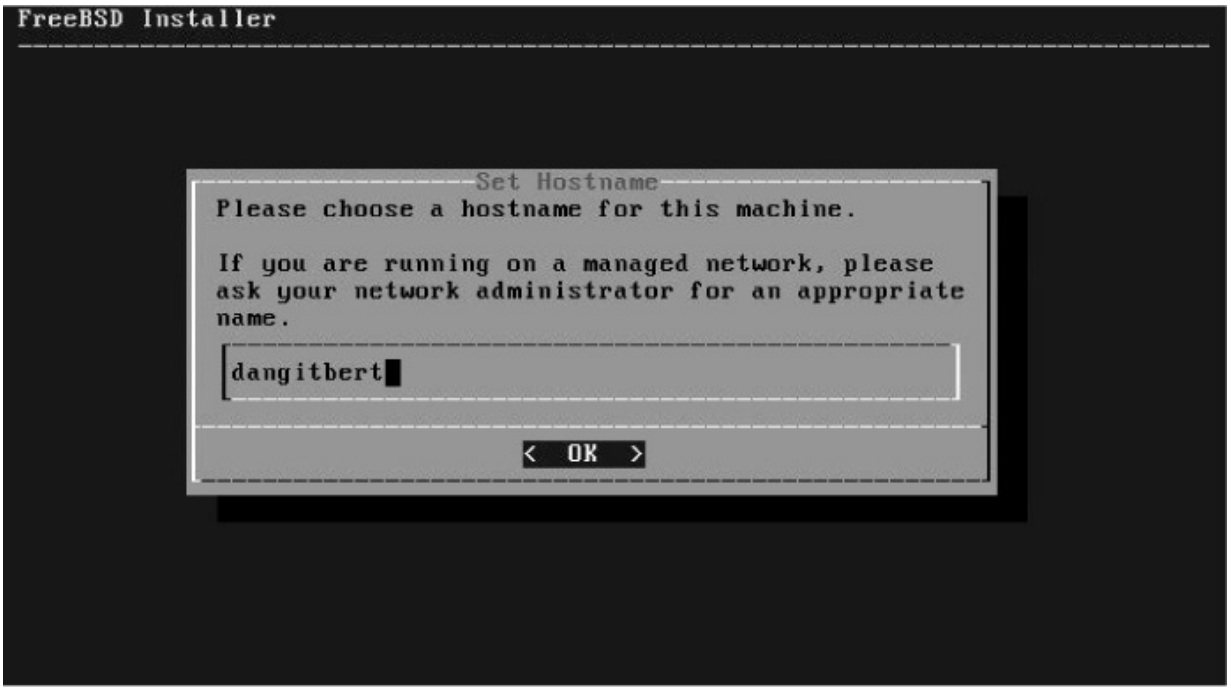


Figure 3-5: Entering a hostname

I'm my own network administrator, so I can use any name I want. Your organization might have different rules. Hit ENTER to proceed.

Distribution Selection

While setting a keymap and a hostname are important, the first truly FreeBSD-specific item comes up when we choose distributions to install. In FreeBSD, a *distribution* is a particular subset of FreeBSD components. When you install FreeBSD, you'll need to pick which distributions you want. The installer doesn't list any mandatory selections: you must have a kernel and the basic userland. Some parts are optional, however (see Figure 3-6).

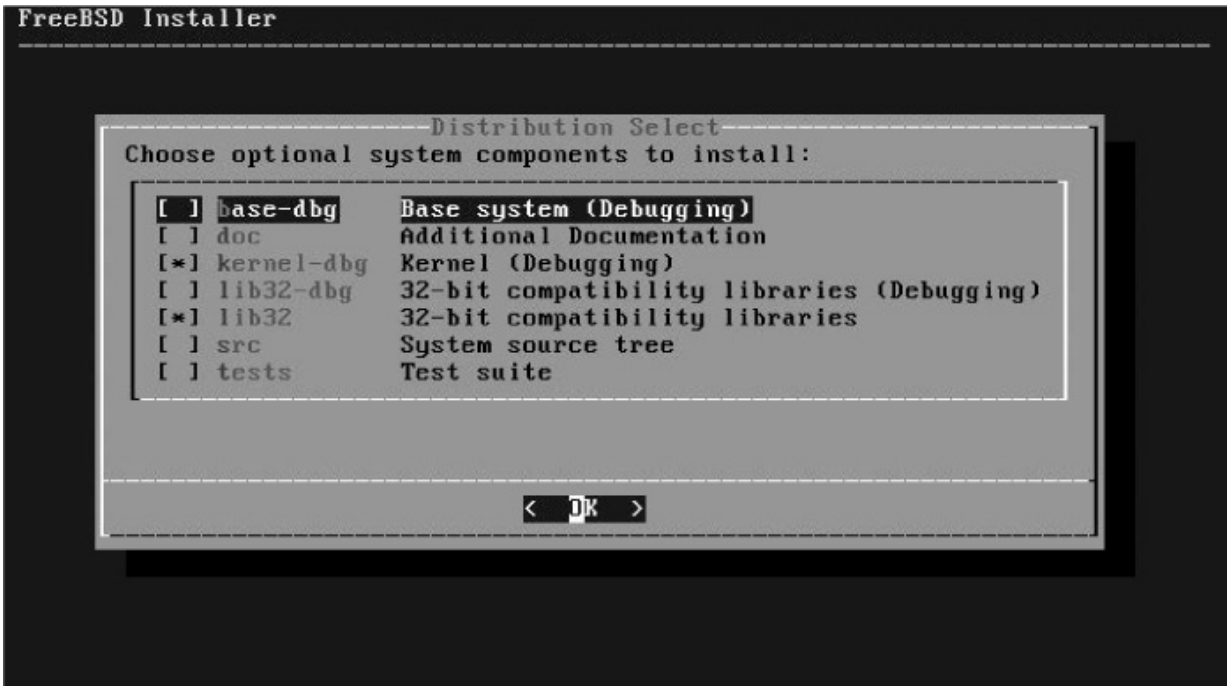


Figure 3-6: Distributions

You don't need any of these, but some will be very useful for certain situations.

base-dbg Debugging symbols for the base system, useful to programmers

doc FreeBSD's official documentation, such as the Handbook

kernel-dbg Debugging symbols for the kernel, useful to programmers

lib32-dbg Debugging symbols for 32-bit libraries (only on 64-bit systems)

lib32 32-bit compatibility libraries (only on 64-bit systems)

src Source code of installed operating system

tests FreeBSD's self-test tools

If you're programming or developing on FreeBSD, or developing FreeBSD itself, arrow up and down to select the debugging libraries. New users might find the documentation helpful. Use the spacebar to select and deselect an option, and ENTER to proceed to disk partitioning.

I recommend always installing the operating system source code. It takes up very little space and can be an invaluable resource.

In my case, I want Bert to bother me as little as possible. I give him all the debugging libraries and the system source code, so if he whinges I can tell him to read */usr/src*.

Disk Partitioning

FreeBSD supports two primary filesystems: UFS and ZFS (see Figure 3-7). Chapter 2 discusses choosing between them, so I won't cover that again. Now I need to stop waffling and make a choice.

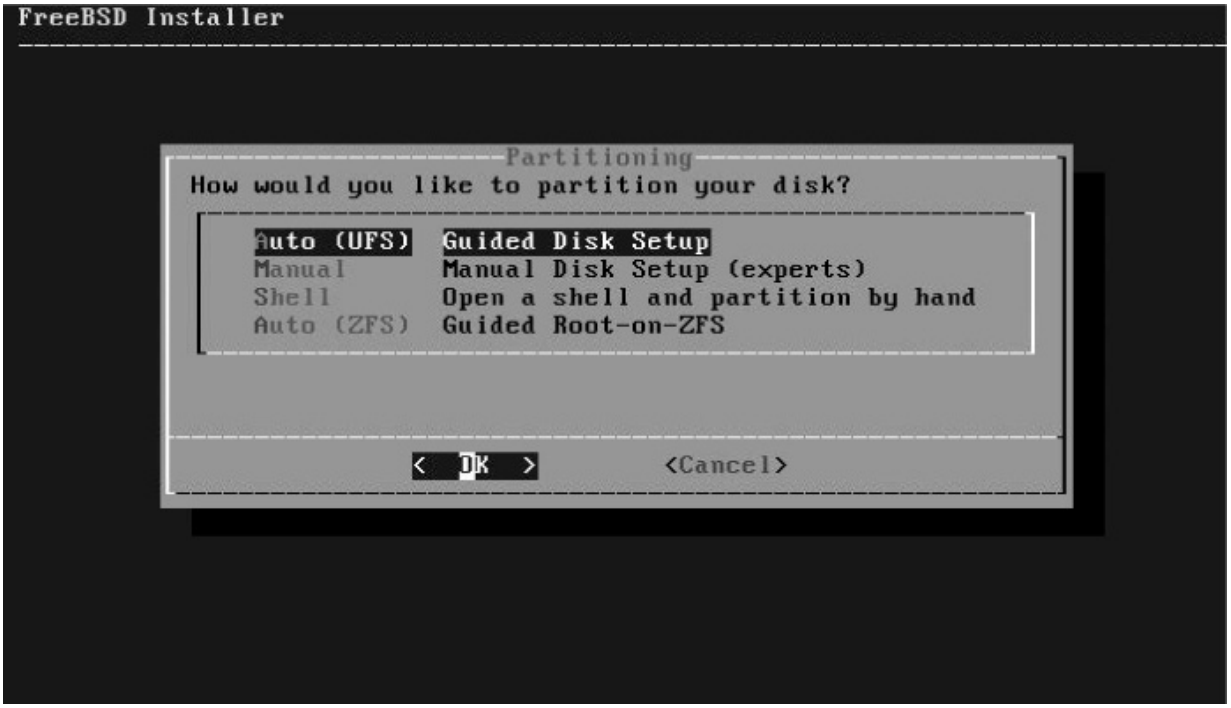


Figure 3-7: Choosing a filesystem

Experienced users can select Manual or, for the hardcore, Shell. I'm letting you follow along, though, so I'll either choose Auto (UFS) or Auto

(ZFS). I'll use UFS to demonstrate disk partitioning and then go on to ZFS.

UFS Installs

Because the default UFS install is straightforward and many people use the default options just fine, I'm choosing some more obscure options to demonstrate using bsdinstall. I'm asked first how much of the disk I want to use, as shown in Figure 3-8.

If Bert wants to use multiple operating systems, he can fire up a hypervisor. I hit ENTER to use the whole disk. A pop-up appears, warning me that I'm about to erase the disk. Yes, I am. That's the point. Select **Yes**. I'm then asked to choose a partition scheme, as shown in Figure 3-9.

Figure 3-8: Disk use

Figure 3-9: Partition schemes

Bsdinstall conservatively defaults to using MBR partitions. Just about everything supports MBR partitions, much like everything supports BIOS rather than EFI, but GPT will cause me much less pain later. I arrow up one space and select GPT, bringing up the default GPT partitioning (see Figure 3-10).

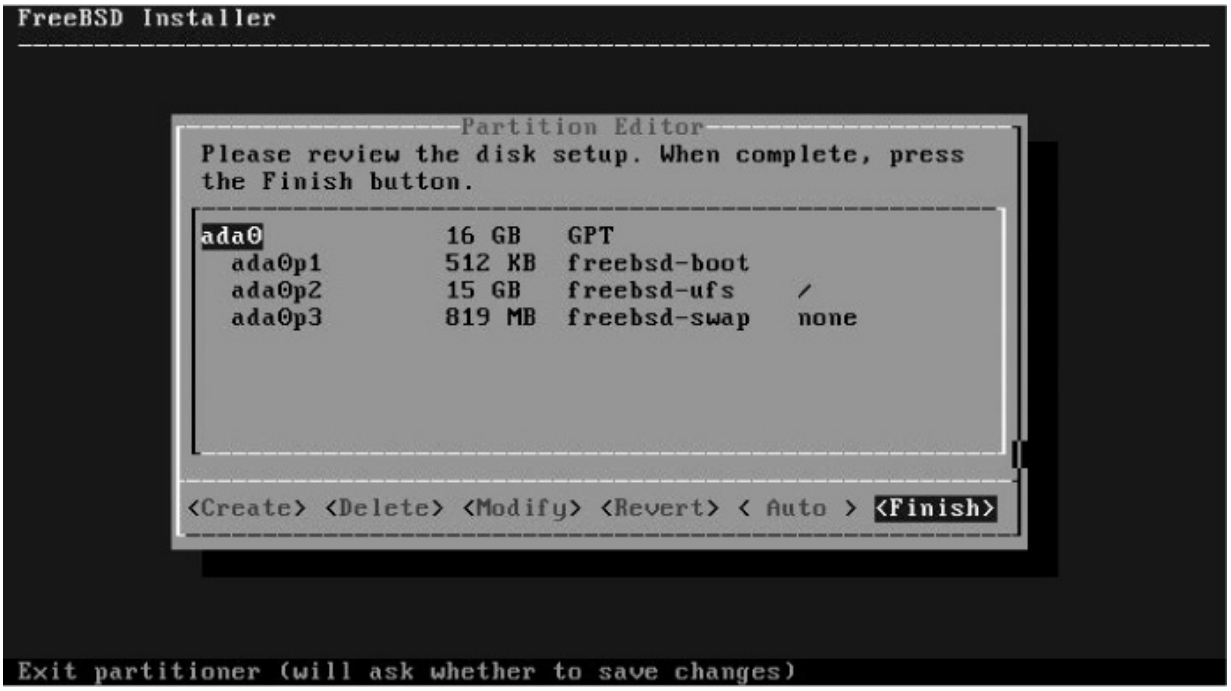


Figure 3-10: Default GPT partitioning

You can hit ENTER right now to finish your UFS partitions, but I'm certain Bert needs special treatment. Let's create special partitioning just for him.

Every GPT system needs a freebsd-boot partition, so leave ada0p1 alone. Arrow down to ada0p2, and either hit D or arrow over to the Delete button to blow it away. Do the same for ada0p3, leaving you with a single partition and a bunch of empty space, as seen in Figure 3-11.

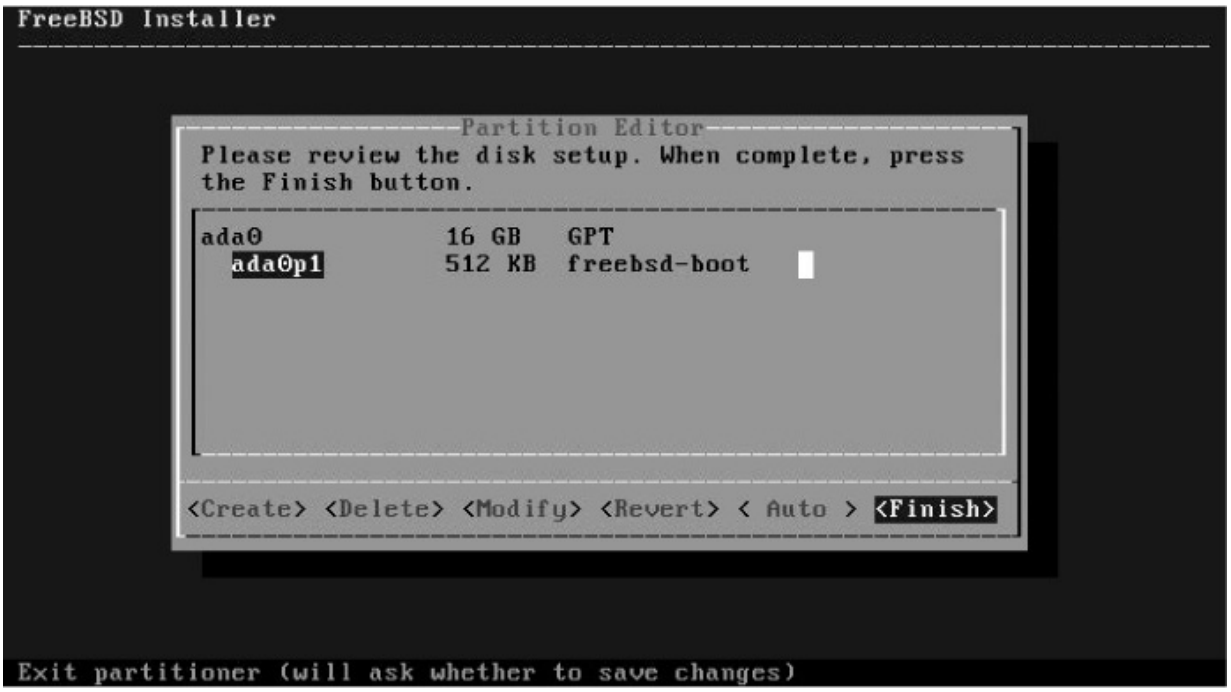


Figure 3-11: Only the boot loader

Now decide how you want this disk partitioned. The disk has 16GB of space, which I'm dividing up as follows:

- 512KB freebsd-boot EFI boot partition
- 1GB swap
- 4GB emergency dump space (see Chapter 24)
- 1GB root (/)
- 512MB /tmp
- 2GB /var
- Everything else in */usr*

The boot partition already exists, so I arrow over to Create or just hit C to add the first partition, bringing up the dialog in Figure 3-12.

Figure 3-12: Adding a new partition

The arrow keys will move you between the options at the bottom of the screen, but you'll need the TAB key to bounce up into the text area at the top. Once you're in the text area, the arrow keys will move you from field to field and back and forth in each line. Our first partition will be swap space, so use the DELETE key to erase the contents of the Type field and enter *freebsd-swap*. Set the size to 1GB. Every partition should have a label, so I label this *swap0*. We discuss labels in Chapter 10.

Now hit TAB to leave the text boxes and select **OK**.

I'm pretty sure that Bert is going to panic this machine and do it in such a terrible way that I'm going to have to dump all of the host's memory to disk. The host has 4GB of RAM, so I create a 4GB dump partition. It'll look exactly like the swap space, including a type of *freebsd-swap*, but I set the size to 4GB and label it *dump0*.

The root partition is a little different, as shown in Figure 3-13. The root partition needs a filesystem, so set the type to *freebsd-ufs*. I've decided to allocate it 1GB. The root partition always has a mountpoint of /, and I label it *root*.

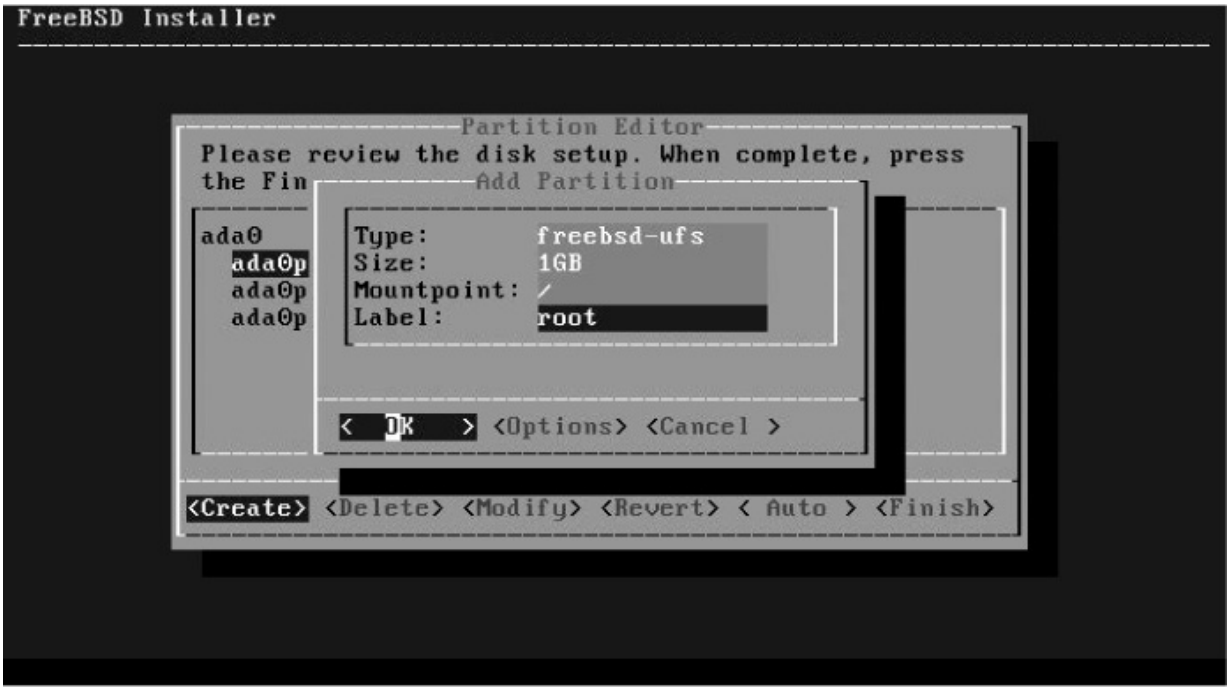


Figure 3-13: Adding the root partition

The remaining partitions for */tmp*, */var*, and */usr* all look similar. When you've used up all the disk space, you'll get a partition table much like that in Figure 3-14.

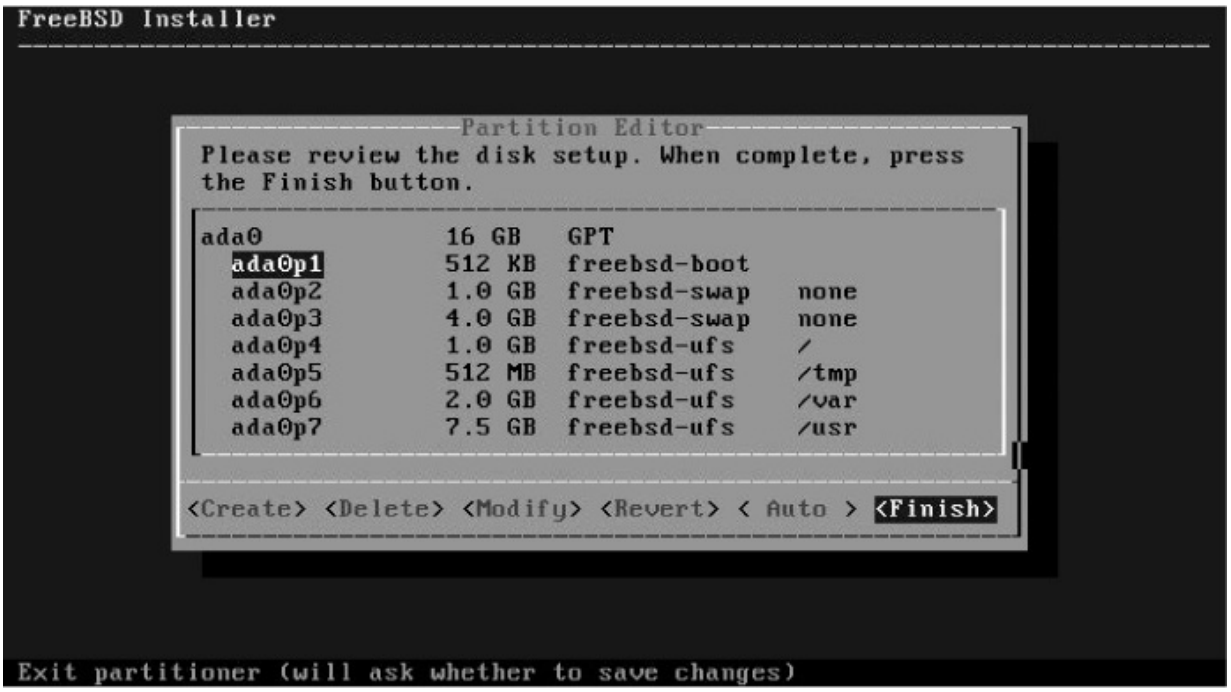


Figure 3-14: Complete custom GPT/UFS partition table

The installer asks me whether I'm sure. This layout should keep Bert from complaining that log files have overflowed his system, so I'm content. Select **Finish** to partition the disk and have the install proceed.

ZFS Installs

If I choose ZFS, I'll get the ZFS configuration screen shown in Figure 3-15.

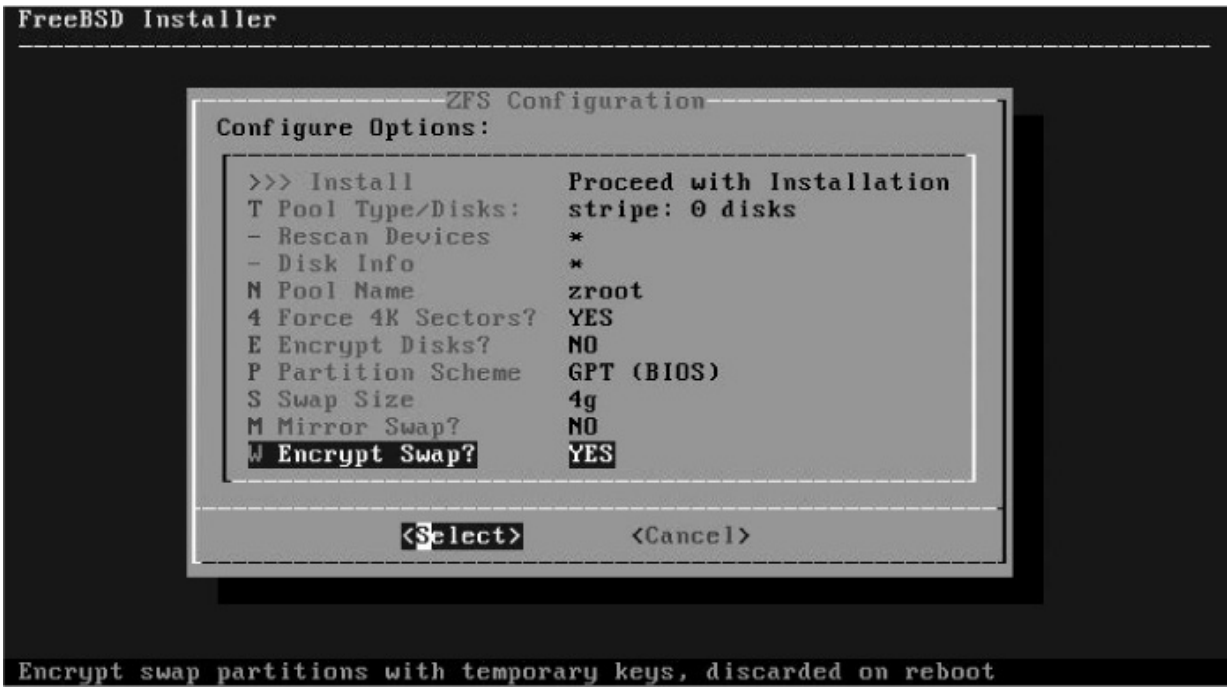


Figure 3-15: ZFS configuration

The default option is Install, which will give you an error because you haven't selected a ZFS virtual device type yet. You'll need to start with Pool Type/Disks. Before we get there, though, let's look at the other choices.

The default name of a FreeBSD root ZFS pool is *zroot*. There's no real reason to change this, unless you want your system to look different than any other ZFS system out there or your organization has standards for naming pools.

The Force 4K Sectors option is important for reasons we'll discuss in Chapter 10. Unless you know for absolutely certain that your disks have 512-byte sectors, leave this option at Yes.

If you choose Encrypt Disks, you'll be prompted for a passphrase for full-disk encryption. FreeBSD uses GELI for ZFS encryption (see Chapter 23),

although when ZFS gets native encryption this might change.

For Partition Scheme, choose GPT. If your host can reasonably run ZFS, it supports GPT.

How much swap space do you need? Adjust Swap Size as necessary. I want this host to have enough space for a full kernel memory dump, because Bert, so I adjust the swap size to 4GB.

Hosts with multiple hard drives can use swap partitions on multiple drives. When a drive containing a swap partition fails, the host loses everything in that swapped-out chunk of memory and crashes. Choosing Mirror Swap gives your swap space redundancy but uses more disk space.

Should you choose Encrypt Swap? There's very little performance cost and, in case your hard drives are stolen, potential advantages.

Now go up and choose Pool Type/Disks to select a ZFS virtual device type, as shown in Figure 3-16.

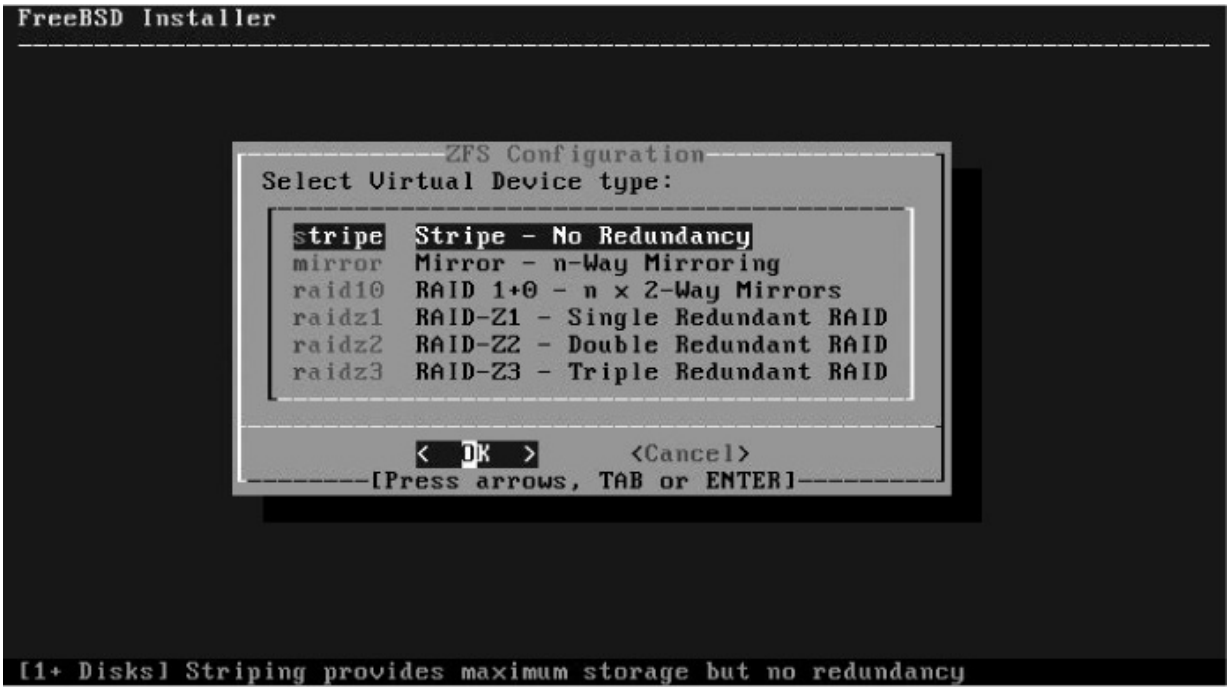


Figure 3-16: Virtual device selection

Chapter 12 discusses ZFS virtual devices at length. Selecting a virtual device type is the most important decision you'll make for a ZFS system. For a single-disk host, however, the only viable option is *stripe*. Select it and you'll get an option to choose the hard drives in your ZFS pool (see Figure 3-17).

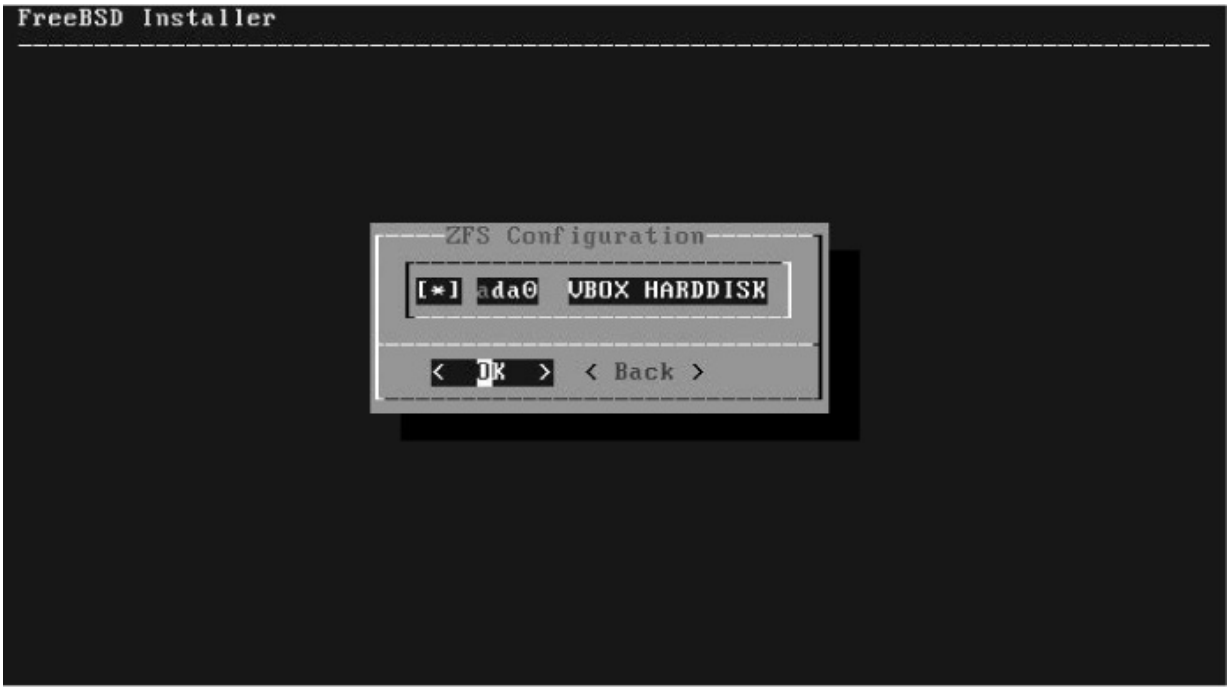


Figure 3-17: ZFS disk selection

Use the spacebar to select the disks you want to include in this pool. As this host has only one disk, I select it and then select OK to continue.

The installer returns me to the main ZFS configuration screen. I double-check my selections (GPT partitioning and 4GB swap) and then arrow up to select Install. The installer gives me a final "Are you really, really sure?" warning. I'm sure.

Network and Service Configuration

Once you approve the disk layout, bsdinstall writes the new partition table to disk, creates filesystems, and extracts the distributions you've chosen without further intervention. The installer moves on to set up the network, services, and users.

First, you are prompted for the system's new root password. The root user can do absolutely anything to the system, so make it a good password. You'll have to enter it twice to have it accepted.

Arrow up and down to choose a network interface. This host has only a single interface, so I hit ENTER to configure it (see Figure 3-18).

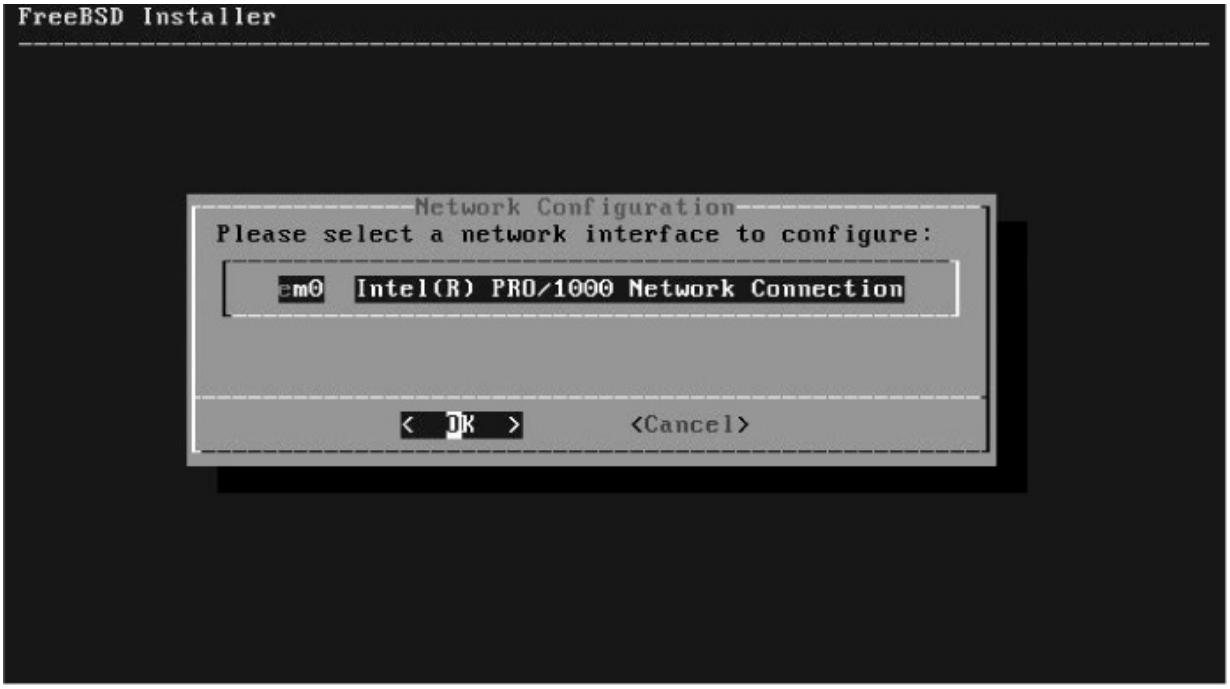


Figure 3-18: Selecting network interface

Next, we're asked whether we want to configure IPv4 for this interface. If you're not sure what IPv4 is, but you want internet, select **Yes**. I certainly do. We're then asked whether we want to use DHCP to automatically configure networking. If this is a disposable system, then probably, but this is going to be Bert's personal server. It needs a special network configuration. I select No and bring up the Network Configuration screen, shown in Figure 3-19.

Your cursor is already up in the text area. Use the arrow keys to move down, not TAB or ENTER. See how OK is highlighted? Once you hit ENTER, the installer proceeds to the next screen whether you've set up the network or not. Fill in the appropriate values for the IP address, subnet mask, and default gateway. If you don't know what these are, you should've used DHCP or read Chapter 7. Don't worry about making a mistake here; if you goof, the last screen of the installer offers a chance to change the network configuration. Hit ENTER when you're done.

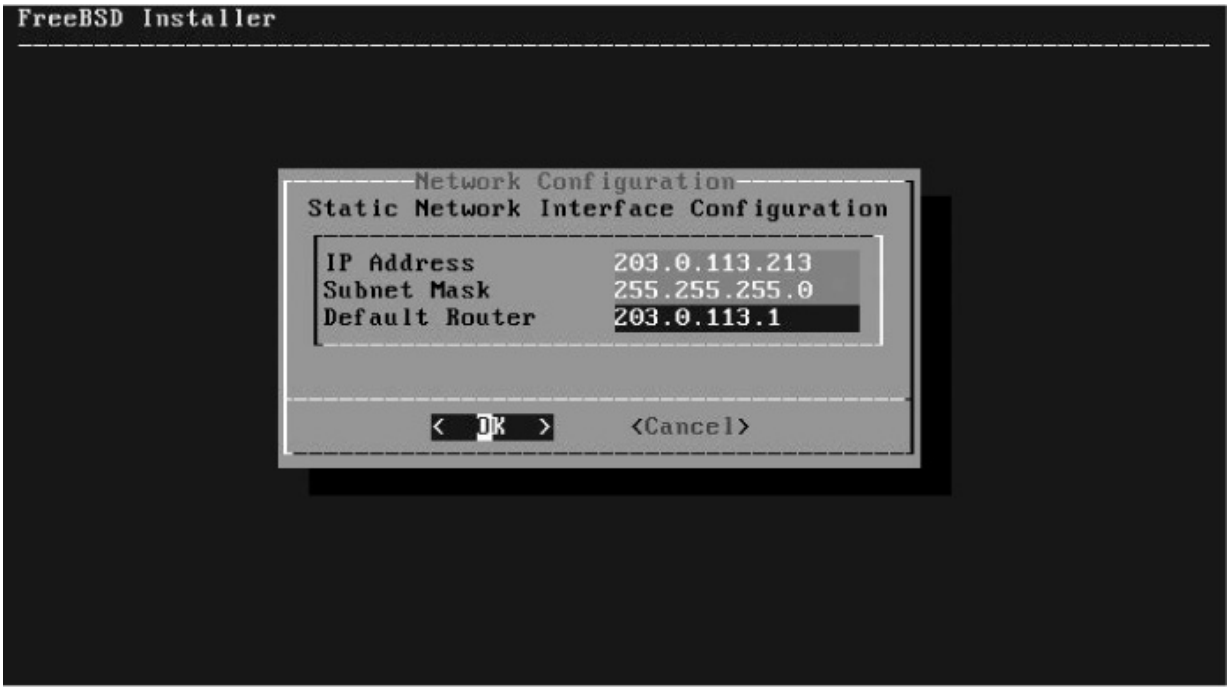


Figure 3-19: Network configuration

Once you've configured IPv4, the installer proceeds to IPv6. You're all on modern networks, so go ahead and configure it. The IP address, netmask, and default router settings are much like IPv4. The installer also supports SLAAC, also known as *DHCP for IPv6*. If you're still on a decrepit IPv4-only network, though, skip IPv6.

You're then given the option to configure DNS. Here, I enter the search domains and nameservers for my network (see Figure 3-20).

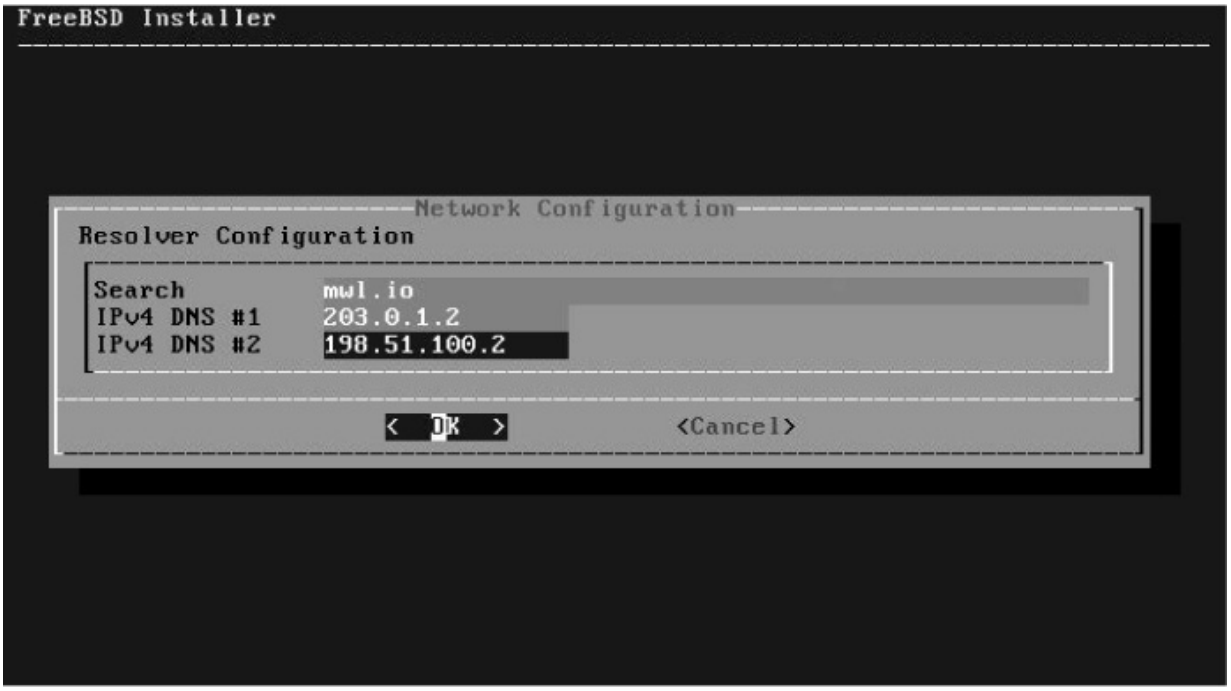


Figure 3-20: Resolver configuration

If you have IP address information for your network but don't know the search domains and the name server IP addresses, copy those values from another machine.

The installer now requests the host's time zone. Rather than dumping all the time zones on you in a giant list, you get a series of hierarchical menus, as shown in Figure 3-21.

Figure 3-21: Time zone selector

Choose your continent. You'll then be asked for a region. I choose United States—Bert's in Europe, yes, but I want him to be painfully aware that if he requests help during his mornings, he's not going to get it. Remember that the END and HOME keys take you to the top and bottom of these long lists; it's much faster to get to the United States by hitting END and going up a couple spots than to page through every country in the Western Hemisphere, including all those little islands. I then get to choose from any time zone in the United States. US citizens will once again be reminded that many states have really messed-up time zones.[2] Even my home of Michigan isn't innocent. But I choose Michigan and am given a chance to confirm my choice (see Figure 3-22).

I recognize EDT, or Eastern Daylight Time. If I didn't, I'd select No and try again.

The next few screens give you the option to set the system clock. Weirdly, the default is set to Skip. While you can enter the time and datehere, it's much easier to set the time from the network, as we'll do later.

Now we can enable a few services at system startup, as shown in Figure 3-23.

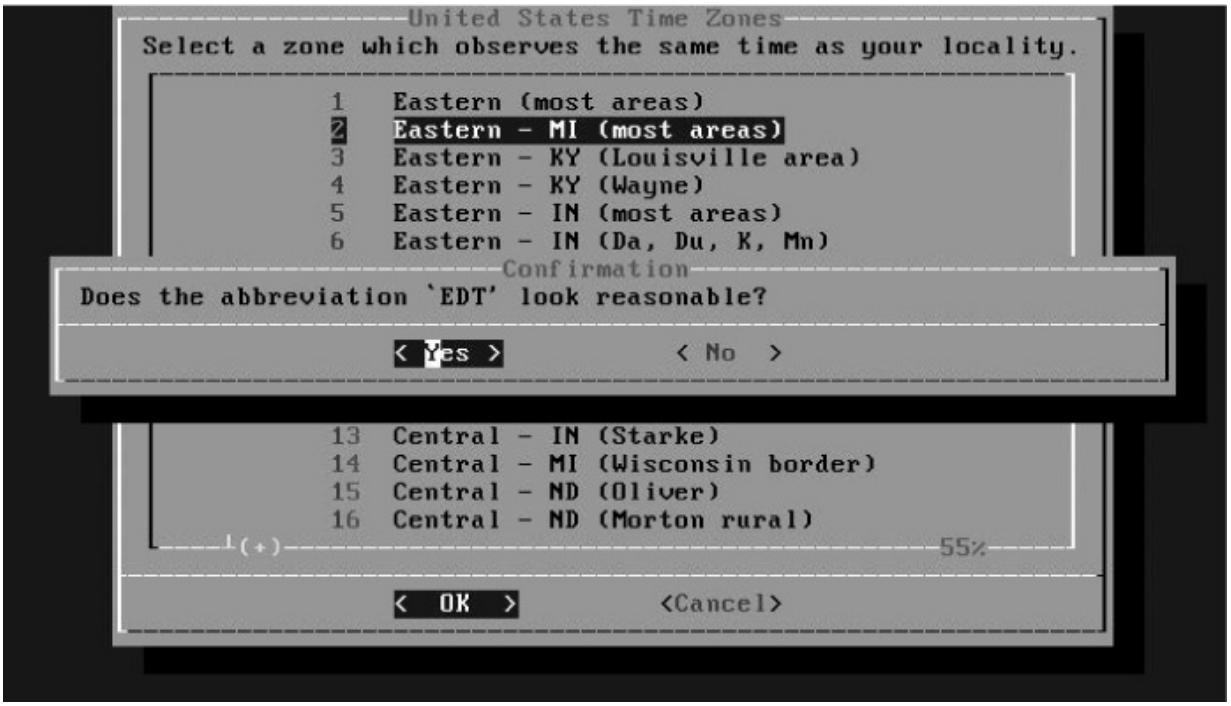


Figure 3-22: US time zones

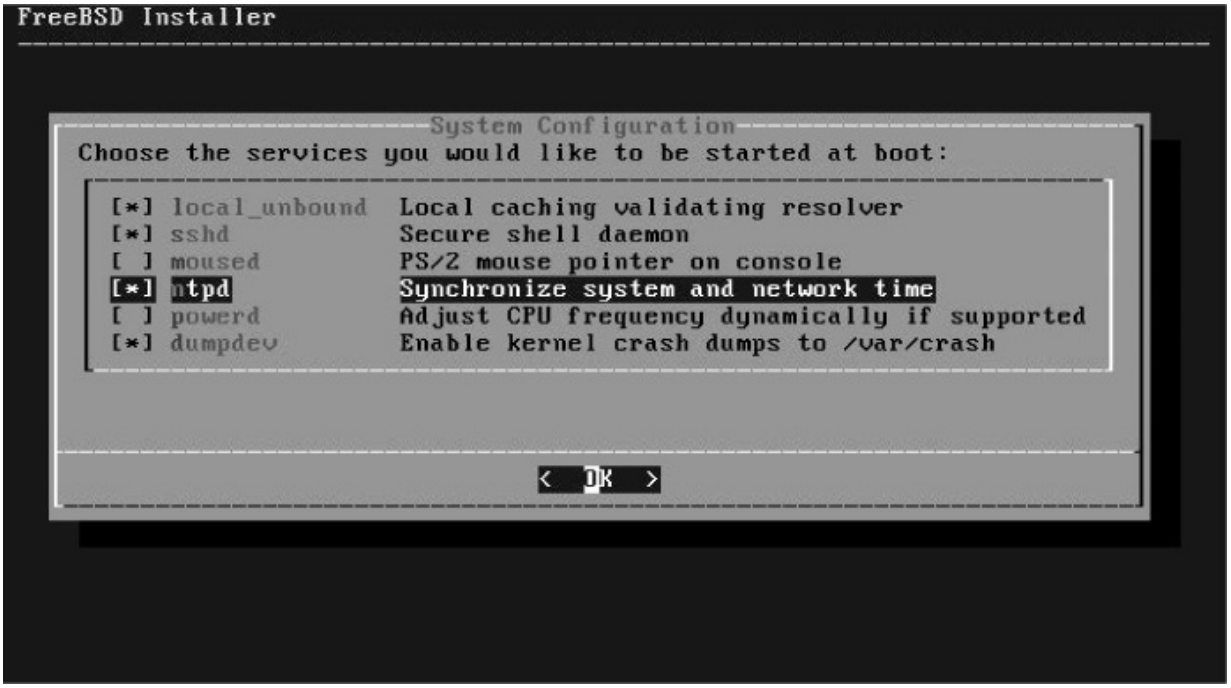


Figure 3-23: Startup services

Most hosts need SSH, and you should *always* enable kernel crash dumps. Other services might not fit your network, though. I always enable `ntpd` (see Chapter 20) and `local_unbound` (see Chapter 8) so that the host's clock

synchronizes itself to the public NTP servers and keeps a local DNS cache, but if your host doesn't have access to the public internet, they aren't as useful. Laptop users might investigate moused(8) and powerd(8).

We then get the system hardening options shown in Figure 3-24.

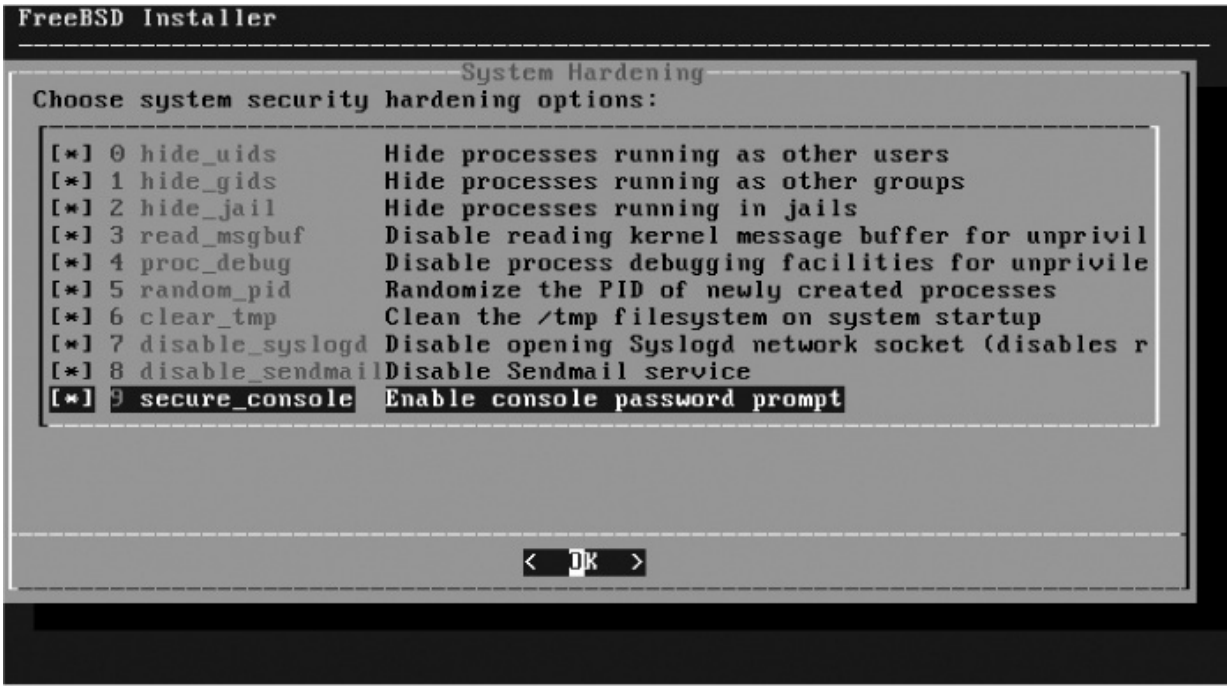


Figure 3-24: Hardening options

We discuss the hardening options at length in Chapter 19. If this is your first install, and you want to have a gentle learning experience, leave them all off. If you want to learn how to work on a more properly secured system, select everything. I enable every hardening option on all of my hosts, and learning to work with improved security will be good for Bert.

Now we can add a user to the system (see Figure 3-25). I recommend adding at least one unprivileged user to each system so that you can log on to the newly installed host without going straight to *root*. If you have a provisioning system such as Ansible that requires a user account, create that account here. This host is for Bert, so I'm giving him an account.

```
FreeBSD Installer
=========================
Add Users

Username: xistence
Full name: Bert Dangit
Uid (Leave empty for default):
Login group [xistence]:
Login group is xistence. Invite xistence into other groups? []: wheel
Login class [default]:
Shell (sh csh tcsh nologin) [sh]: tcsh
Home directory [/home/xistence]:
Home directory permissions (Leave empty for default):
Use password-based authentication? [yes]:
Use an empty password? (yes/no) [no]:
Use a random password? (yes/no) [no]:
Enter password:
Enter password again:
Lock out the account after creation? [no]:
```

Figure 3-25: Adding a user

Chapter 9 discusses creating user accounts in detail, but I'll give some reasonable settings for the first account here. Bert's preferred account name is *xistence*, and I'll indulge him in it. I fill in his first name, and just hit ENTER to take the default *Uid* and *Login group*. He's the primary user on this system, so I add him to the wheel group, allowing him to use the root password. He gets the tcsh shell because it's my favorite.

If you have a policy on where user home directories go, follow it. Otherwise, take the defaults. Similarly, while you can adjust the password settings to fit the default, generally speaking, it's easiest to type the user's password. Many people recommend a password like *ChangeMe*, but I prefer to go with passwords that actively encourage users to change them as soon as possible—maybe something like *BertIsTheWorstIMeanTheWorstHumanBeingEver*.[3] And if I lock out the account after I create it, I'll need to unlock it only when he wants to use the machine.

After adding one user, I'm asked whether I want to add another. If I add an account for myself, I'll bear partial liability for this host. I say No.

Finishing the Install

The core configuration, shown in Figure 3-26, is all done! I then get a chance to go back and tweak some settings.

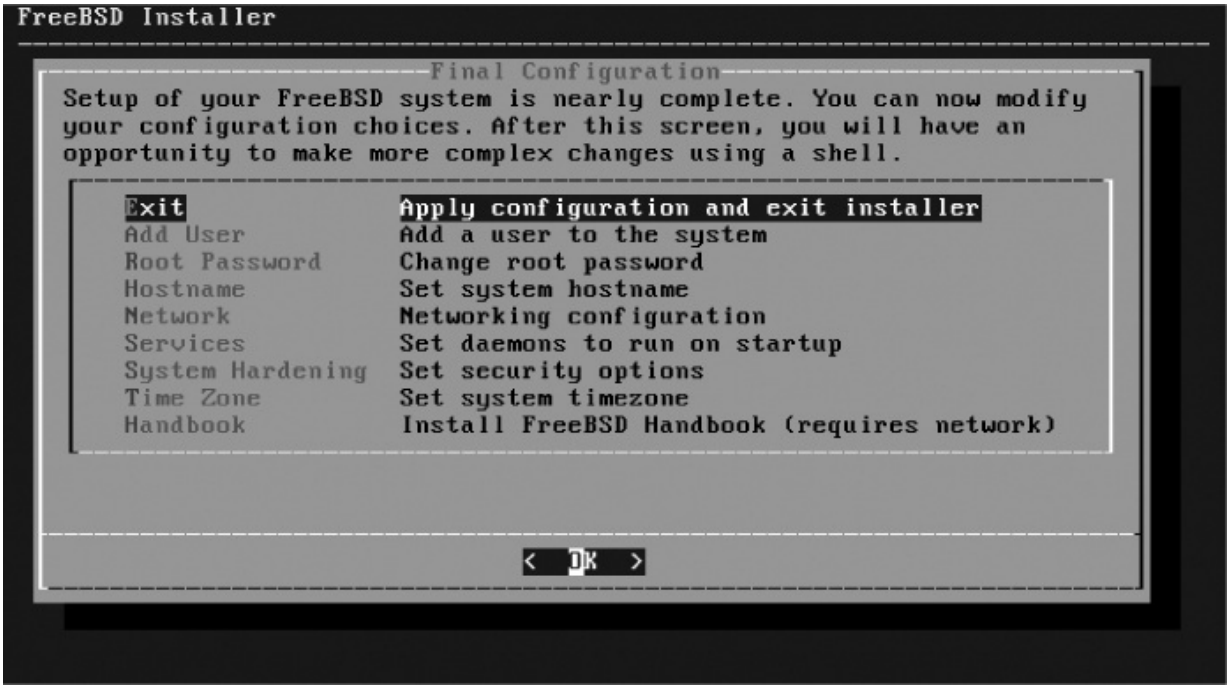


Figure 3-26: Final configuration

Most of these options come straight from earlier in the install process. Do you want to go back to change the network configuration? Choose **Network**. Should you add another user or enable more services? Did you enter the wrong password? This is your chance to right those wrongs.

When you think you're ready, select **Exit** to discover you don't have to be done.

The installer covers the basics, but every environment is unique. Manual configuration offers a command prompt chrooted into the system that gives you the chance to make any final changes (see Figure 3-27). Choose No and you'll be told to remove the boot media and reboot. I often find tweaking a host before its first boot simplifies my life, so I choose Yes.

Figure 3-27: Manual configuration

I'm chrooted into the installed host with a root shell. The exact tasks you perform here depend entirely on your network. Chapter 9 discusses chflags(8) and schg. Now I type `exit`, as shown in Figure 3-28.

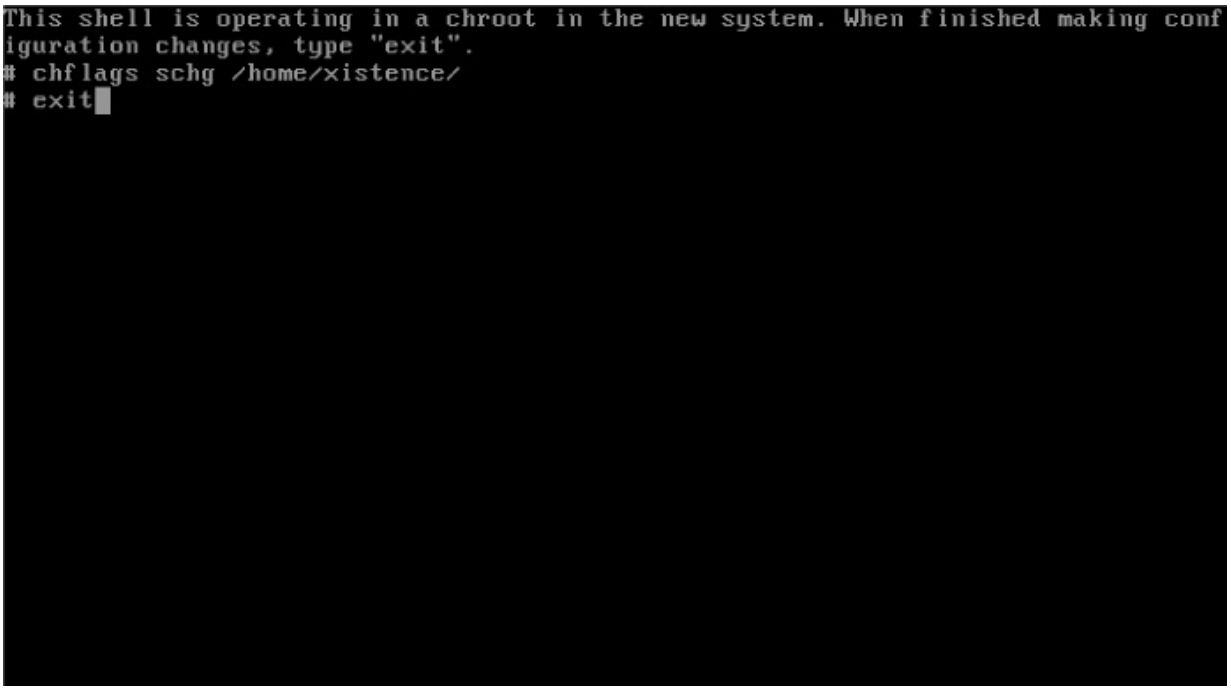


Figure 3-28: Final shell configuration

Then I reboot, pull the installation media, and boot into a complete FreeBSD install!

4
START ME UP! THE BOOT PROCESS

While FreeBSD boots easily and automatically when you turn on the power, understanding exactly what happens at each stage will make you a better system administrator. Intervention during the boot process is rarely necessary, but one day you'll be glad you know how to do it. And once you're comfortable with adjusting the boot process, you'll find you can solve problems you've previously accepted and endured.

We'll start by discussing how the system loader starts and gathering information from the loader. You can use the loader to change the early boot process, including booting alternate kernels and starting in single-user mode. We'll cover serial consoles, a standard system management tool. The FreeBSD multiuser startup process is responsible for starting all the various services that make your computer useful, and we'll give attention to that as well. In addition, we'll cover the information FreeBSD records about the boot process and how FreeBSD turns itself off without corrupting data.

RECURSION WARNING

Some of the topics in this chapter reference material found in later chapters. Those later chapters, in turn, require that you understand this chapter first. There's no good place to begin learning. If you don't quite understand a part of this chapter, just skim over it and continue reading; it really will coalesce in your mind as you proceed.

The boot process itself can be divided into three main parts: the loader, single-user startup, and multiuser startup.

Power-On

A computer needs enough brains to find and load its operating system. For many years, this facility came from the basic input/output system (BIOS). Newer systems use the Unified Extensible Firmware Interface (UEFI) instead of the BIOS. New installs should use UEFI. Other hardware platforms have console firmware or bootroms that perform the same function, but we're focused on commodity hardware, so we'll cover UEFI and BIOS.

Unified Extensible Firmware Interface

UEFI is a replacement for the three-decades-old BIOS. Any new system will come with UEFI enabled and will expect to use it.

UEFI searches the boot drive for a partition marked as a UEFI boot partition. Despite what the special mark might imply, that partition contains only a FAT filesystem with a specific directory and file layout. UEFI executes the file */EFI/BOOT/BOOTX64.EFI*. That file might be a fancy multi-OS boot loader, or it might dump you straight into an operating system. In FreeBSD, the UEFI boot fires up the boot loader, */boot/loader.efi*.

UEFI is comparatively new. If your new system has trouble booting FreeBSD, you might try enabling a BIOS or "legacy" mode. If the system boots FreeBSD in BIOS mode but not with UEFI, please file a bug, as discussed in Chapter 24.[1]

Basic Input/Output System

The primordial Intel PC used a basic input/output system (BIOS) with just enough brains to look for an operating system somewhere on a disk. A BIOS searches for a disk partition marked active and then executes the first section of that partition. For FreeBSD, that chunk of data is called the *loader*. Every FreeBSD system has a reference copy of the loader as */boot/loader*.

A BIOS has all sorts of limitations. The boot loader must reside in a very specific section of the disk. BIOS can't boot from disks larger than 2.2TB. The target boot loader must be smaller than 512KB—huge by 1980 standards, yes, but paltry today. The installed loader is a binary, not a filesystem, so even minor changes require recompiling. UEFI has none of these limitations and offers modern features, like mouse support.

Ultimately, though, a BIOS and UEFI both have the goal of getting your system to the FreeBSD loader.

The Loader

The *loader*, or *boot blocks*, loads the FreeBSD kernel and presents you with a menu before starting that kernel. The loader(8) program offers a menu of seven options on the left. A new FreeBSD system presents these options:

1. Boot Multi User [Enter]
2. Boot Single User
3. Escape to loader prompt
4. Reboot
5. Kernel: default/kernel (1 of 2)
6. Configure Boot Options…
7. Select Boot Environment…

Each option highlights certain words or characters, such as *S* in "Boot Single User" and ESC in "Escape to loader prompt." Select an option by pressing the highlighted character or the number.

The options at the top of the menu control how FreeBSD boots. We'll look at each option in turn. If you wait 10 seconds, the loader automatically boots FreeBSD by default.

The options at the bottom half let you fine-tune the boot process. You can tweak how you want the system to boot, as we'll discuss later, and then choose one of the preceding booting options.

Boot Multi User [Enter]

This is a normal boot. Hit ENTER to boot immediately, skipping the 10-second delay.

Boot FreeBSD in Single-User Mode

Single-user mode is a minimal startup mode that's very useful on damaged systems, especially when the damage was self-inflicted. It's the earliest point where FreeBSD can provide a command prompt, and it's important enough to have its own section later in this chapter.

Escape to Loader Prompt

The loader includes a command line interpreter, where you can issue commands to tweak your system to boot exactly the way you need. We'll cover this in detail in "The Loader Prompt" on page 55.

Reboot

Once more, this time with feeling!

Of these options, the most important are single-user mode and the loader prompt.

Single-User Mode

FreeBSD can perform a minimal boot, called *single-user mode*, that loads the kernel and finds devices but doesn't automatically set up your filesystems, start the network, enable security, or run any standard Unix services. Single-user mode is the earliest the system can possibly give you a command prompt.

Why use single-user mode? If a badly configured daemon hangs the boot, you can enter single-user mode to prevent it from starting. If you've lost your root password, you can boot into single-user mode to change it. If you need to shuffle critical filesystems around, again, single-user mode is the place to do it.

When you choose a single-user mode boot, you'll see the regular system startup messages flow past. Before any programs start, however, the kernel

offers you a chance to choose a shell. You can enter any shell on the root partition; I usually just take the default */bin/sh*, but use */bin/tcsh* if you prefer.

Disks in Single-User Mode

In single-user mode, the root partition is mounted read-only and no other disks are mounted. (We'll discuss mounting filesystems in Chapter 10, but for now just follow along.) Many of the programs that you'll want to use are on partitions other than the root, so you'll want them all mounted read-write and available. The way to do this varies depending on whether you're using UFS or ZFS.

UFS in Single-User Mode

To make all the filesystems listed in the filesystem table */etc/fstab* usable, run the following commands:

```
# fsck -p
# mount -o rw /
# mount -a
```

The fsck(8) program "cleans" the filesystems and confirms that they're internally consistent and that all the files that a disk thinks it has are actually present and accounted for.

The root filesystem is mounted read-only. Whatever drove us to single-user mode probably requires changing the root filesystem. Remount the root filesystem read-write.

Finally, the `-a` flag to mount(8) activates every filesystem listed in */etc/fstab* (see Chapter 10). If one of these filesystems is causing you problems, you can mount the desired filesystems individually by specifying them on the command line (for example, `mount /usr`). If you're an advanced user with NFS filesystems configured (see Chapter 13), you'll see error messages for those filesystems at this point because the network isn't up yet. If the host has network filesystems in */etc/fstab*, mount only the UFS filesystems as shown next.

If you have trouble mounting partitions by name, try using the device name instead. The device name for the root partition is probably */dev/ad0s1a*. You'll also need to specify a mount point for this partition. For example, to mount your first IDE disk partition as root, enter the command:

```
# mount /dev/ad0s1a /
```

If you have network filesystems on your server but your network isn't up yet, you can mount all your local partitions by specifying the filesystem type. Here, we mount all of the local filesystems of type UFS, which is FreeBSD's default filesystem type:

```
# mount -a -t ufs
```

You can now access your UFS filesystems.

ZFS in Single-User Mode

To make all of your ZFS datasets available, use `zfs mount`. You can either mount individual datasets by name or mount everything that's marked as mountable with `-a`.

```
# zfs mount -a
```

ZFS will perform its usual integrity checks before mounting the datasets.

Most of the datasets will be exactly as accessible as in multiuser mode, but the dataset mounted as root will still be read-only. Turn that off. Here, I'm setting the root dataset to read-write on a default FreeBSD install.

```
# zfs set readonly=off zroot/ROOT/default
```

You can now change the filesystem.

Programs Available in Single-User Mode

The commands available for your use depend on which partitions are mounted. Some basic commands are available on the root partition in */bin* and */sbin*, and they're available even if root is mounted read-only. Others live in */usr* and are inaccessible until you mount that partition. (Take a look at */bin* and */sbin* on your system to get an idea of what you'll have to work with when things go bad.)

If you've scrambled your shared library system (see Chapter 17), none of these programs will work. If you're that unlucky, FreeBSD provides statically linked versions of many core utilities in the */rescue* directory.

The Network in Single-User Mode

If you want to have network connectivity in single-user mode, use the shell script */etc/netstart*. This script calls the appropriate scripts to start the network, gives IP addresses to interfaces, and enables packet filtering and routing. If you want some, but not all, of these services, you'll need to read that shell script and execute the appropriate commands manually.

Uses for Single-User Mode

In single-user mode, your access to the system is limited only by your knowledge of FreeBSD and Unix.

For example, if you've forgotten your root password, you can reset it from single-user mode:

```
# passwd
Changing local password for root
New Password:
Retype New Password:
#
```

NOTE

You'll notice that you weren't asked for the old root password. In single-user mode, you're automatically root, and passwd(8) doesn't ask root for any password.

Or, if you find that there's a typo in */etc/fstab* that confuses the system and makes it unbootable, you can mount the root partition with the device name and then edit */etc/fstab* to resolve the issue.

If you have a program that panics the system on boot and you need to stop that program from starting again, you can either edit */etc/rc.conf* to disable the program or set the permissions on the startup script so that it can't execute.

```
# chmod a-x /usr/local/etc/rc.d/program.sh
```

We'll discuss third-party programs (ports and packages) in Chapter 15.

You need to understand single-user mode to be a successful sysadmin,

and we'll refer to it throughout this book. For now, though, let's look at the loader prompt.

SYSTEM FAILURES VERSUS HUMAN FAILINGS

There's a reason all of these examples involve recovering from human errors. Hardware failures aren't common, and FreeBSD failures, even less so. If it weren't for human error, our computers would almost never let us down. As you learn more about FreeBSD, you'll be more and more capable in single-user mode.

The Loader Prompt

The loader prompt allows you to make basic changes to your computer's boot environment and the variables that must be configured early in the boot process. It's not a Unix-like environment; it's cramped and supports only a minimal feature set. When you escape to a loader prompt (the third option in the boot menu), you'll see the following:

```
Type '?' for a list of commands, 'help' for more detailed help.
OK
```

This is the loader prompt. While the word *OK* might be friendly and reassuring, it's one of the few friendly things about the loader environment. This isn't a full-featured operating system; it's a tool for configuring a system boot that's not intended for the ignorant nor the faint of heart. Any changes you make at the loader prompt affect only the current boot. To undo changes, reboot again. (We'll see how to make loader changes permanent in the next section.)

To see all available commands, enter a question mark.

```
OK ?
Available commands:
  heap            show heap usage
  reboot          reboot the system
  lszfs           list child datasets of a zfs dataset
```

```
--snip--
```

Many loader commands aren't useful to anyone except a developer, so we'll focus on the commands useful to a system administrator.

Viewing Disks

To view the disks that the loader knows about, use `lsdev`.

```
   OK lsdev
❶ cd devices:
   disk devices:
❷ disk0:    BIOS drive C (33554432 X 512):
      ❸ disk0p1: FreeBSD boot
         disk0p2: FreeBSD swap
       disk0p3: FreeBSD ZFS
❹ zfs devices:
       zfs:zroot
```

The loader checks for CD drives ❶ and doesn't find any. (The loader finds CD drives only if you boot from a CD, so don't be alarmed by this.) It finds a hard drive, known to the BIOS as drive C ❷. It then describes the partitions on that hard drive. As we'll see in Chapter 10, GPT partitions identify partitions with the letter *p* and a number. The partition disk0p1 ❸ is a FreeBSD boot partition used to bootstrap FreeBSD from the BIOS. You might find this knowledge useful on an unfamiliar system that's having trouble booting. The loader can also identify the ZFS pools ❹ on the host.

Loader Variables

The loader has variables set within the kernel and by a configuration file. View these variables and their settings with the `show` command, and use the spacebar to advance to the next page.

```
OK show
LINES=24
acpi.oem=VBOX
acpi.revision=2
acpi.rsdp=0x000e0000
--snip--
```

These values include low-level kernel tunables and information gleaned from the hardware BIOS or UEFI. We'll see a partial list of loader variables in "Loader Configuration" on page 57, and additional values will be brought up throughout the book in the appropriate sections.

You can show specific variables by name. Sadly, you can't show all of a keyword's sub-variables. A command like `show acpi.oem` works, but `show acpi` or `show acpi.*` doesn't.

Change a value for a single boot with the `set` command. For example, to change the `console` setting to `comconsole`, you'd enter:

```
OK set console=comconsole
```

The loader lets you change variables that really shouldn't change. Setting acpi.revision to 4 won't suddenly upgrade your system to ACPI version 4, and you can't change hard drives with a software setting.

Reboot

You didn't mean to get into the loader? Start over.

Booting from the Loader

Now that you've twiddled your system's low-level settings, you probably want to boot the system. Use the boot(8) command. You can adjust the boot further using the boot flags discussed in the man page.

Once your system boots just the way you need it to, you'll probably want to make those settings permanent. FreeBSD lets you do this through the loader configuration file.

Loader Configuration

Make loader setting changes permanent with the configuration file */boot/loader.conf*. Settings in this file are fed directly into the boot loader at system startup. Of course, if you enjoy being at your console every time the system boots, then don't bother with this!

The loader has a default configuration file, */boot/defaults/loader.conf*. We override many of the values here.

If you look at the default loader configuration, you'll see many options that resemble variables listed in the loader. For example, here we can set the name of the console device:

```
console="vidconsole"
```

Throughout the FreeBSD documentation, you'll see references to *boot-time tunables* and *loader settings*. All of these are set in *loader.conf*, which includes many sysctl values that are read-only once the system is up and kicking. (For more on tunables and sysctls, see Chapter 6.) Here, we force the kernel variable `kern.maxusers` to `32`.

```
kern.maxusers="32"
```

Some of these variables don't have a specific value set in *loader.conf*; instead, they appear as empty quotes. This means that the loader normally lets the kernel set this value, but if you want to override the kernel's setting, you can.

```
kern.nbuf=""
```

The kernel has an idea of what the value of `kern.nbuf` should be, but you can have the loader dictate a different value if you must.

We'll discuss system tuning via the boot loader in the appropriate section—for example, kernel values will be discussed in Chapter 6, where they'll make something resembling sense—but here are some commonly used loader values that affect the appearance and operation of the loader itself and basic boot functionality. As FreeBSD matures, the developers introduce new loader values and alter the functionality of old ones, so be sure to check */boot/defaults/loader.conf* on your installation for the current list.

`boot_verbose="NO"`

This value toggles the verbose boot mode that you can reach through the boot menu. In a standard boot, the kernel prints out a few basic notes about each device as it identifies system hardware. When you boot in verbose mode, the kernel tells each device driver to print out any and all information it can about each device as well as display assorted kernel-related setup details. Verbose mode is useful for debugging and development, but not generally for day-to-day use.

autoboot_delay="10"

This value indicates the number of seconds between the display of the boot menu and the automatic boot. I frequently turn this down to 2 or 3 seconds, as I want my machines to come up as quickly as possible.

beastie_disable="NO"

This value controls the appearance of the boot menu (originally, an ASCII art image of the BSD "Beastie" mascot decorated the boot menu). If set to `YES`, the boot menu will not appear.

loader_logo="fbsdbw"

This value allows you to choose which logo appears to the right of the boot menu. The `fbsdbw` option gives you the default FreeBSD logo in ASCII art. Other options include `beastiebw` (the original logo), `beastie` (the logo in color), and `none` (no logo).

Boot Options

The boot menu also presents three options: choosing a kernel, setting boot options, and selecting a boot environment. We'll discuss each of these in an appropriate section, but here's a bit to orient you.

A host can have multiple kernels in its */boot* directory. Hitting the *Kernel* option tells the loader to cycle between the available options. To have a kernel appear as an option, list it in *loader.conf* in the kernels variable.

```
KERNELS="kernel kernel.old kernel.GENERIC"
```

The menu recognizes kernels only in directories beginning with */boot/kernel*. If you have a kernel in */boot/gerbil*, you'll have to load it from the loader prompt.

FreeBSD supports a number of boot options. Selecting the *Configure Boot Options* item brings up the most popular.

Load System Defaults

You mucked with your settings and want to undo all that? Choose this. You can at least boot the system to single-user mode and fix your *loader.conf*.

ACPI Support

ACPI is the Advanced Configuration and Power Interface, an Intel/Toshiba/Microsoft standard for hardware configuration. It replaces and subsumes a whole bunch of obscure standards. ACPI has been a standard for many years now, but if a particular piece of hardware has trouble running FreeBSD, you can turn it off and see what happens. If you even think of trying this option, also read Chapter 24 and file a bug report.

Safe Mode

FreeBSD's *safe mode* turns on just about every conservative option in the operating system. It turns off DMA and write caching on hard disks, limiting their speed but increasing their reliability. It turns off ACPI. 32-bit systems disable SMP. USB keyboards no longer work in safe mode. This option is useful for debugging older hardware.

Verbose

The FreeBSD kernel probes every piece of hardware as it boots. Most of the information discovered is irrelevant to day-to-day use, so the boot loader doesn't display it. When you boot in verbose mode, FreeBSD prints all the details it can about every system setting and attached device. The information will be available later in */var/run/dmesg.boot*, as discussed in the next section. I encourage you to try verbose mode on new machines, just to glimpse the system's complexity.

Finally, the *Select Boot Environment* option lets you choose between ZFS boot environments, as discussed in Chapter 12.

Startup Messages

A booting FreeBSD system displays messages indicating the hardware attached to the system, the operating system version, and the status of various programs and services as they start. These messages are important when you first install your system and when you do troubleshooting. The boot messages always start off the same way, with a statement listing the copyrights for the FreeBSD Project and the Regents of the University of California:

```
Copyright (c) 1992-2018 The FreeBSD Project.
Copyright (c) 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994
    The Regents of the University of California. All rights reserved.
FreeBSD is a registered trademark of The FreeBSD Foundation.
FreeBSD 12.0-CURRENT #3 r320502: Fri Jun 30 13:48:50 EDT 2017
    root@storm:/usr/obj/usr/src/sys/GENERIC amd64
FreeBSD clang version 4.0.0 (tags/RELEASE_400/final 297347) (based on LLVM 4.0.0)
```

In addition, you get a notice of the version of FreeBSD that's booting, along with the date and time it was compiled and the compiler used. You can also see who compiled this kernel, what machine it was built on, and even where in the filesystem this kernel was built. If you build a lot of kernels, this information can be invaluable when trying to identify exactly what system features are available.

```
WARNING: WITNESS option enabled, expect reduced performance.
```

The kernel will print out diagnostic messages throughout the boot process. The preceding message means that I have debugging and fault-identifying code enabled in this particular kernel, and my performance will suffer as a result. In this case, I don't care about the performance impact, for reasons which will become clear momentarily.

```
Timecounter "i8254" frequency 1193182 Hz quality 100
```

This message identifies a particular piece of hardware. The *timecounter*, or *hardware clock*, is a special piece of hardware, and while your computer needs one, it's such a low-level device that the end user really can't do much with it directly. Now and then, you'll see messages like this for hardware that isn't directly visible to the user but is vital to the system. The boot messages dance between showing too much detail and obscuring details that might be critical. For example, it'll also show all the information it can about the CPU in the system:

```
CPU: Intel(R) Xeon(R) CPU E5-1620 v2 @ ❶3.70GHz (3700.08-MHz K8-class CPU)
  Origin="GenuineIntel"  Id=0x306e4  Family=0x6  Model=0x3e  Stepping=4
             ❷
Features=0xbfebfbff<FPU,VME,DE,PSE,TSC,MSR,PAE,MCE,CX8,APIC,SEP,MTRR,PGE,MCA,
CMOV,PAT,PSE36,CLFLUSH,DTS,ACPI,MMX,FXSR,SSE,SSE2,SS,HTT,TM,PBE>
  Features2=0x7fbee3ff<SSE3,PCLMULQDQ,DTES64,MON,DS_CPL,VMX,SMX,EST,TM2,SSSE3,CX16,
xTPR,PDCM,PCID,DCA,SSE4.1,SSE4.2,x2APIC,POPCNT,TSCDLT,AESNI,XSAVE,OSXSAVE,AVX,F16C,RDRA
  AMD Features=0x2c100800<SYSCALL,NX,Page1GB,RDTSCP,LM>
  AMD Features2=0x1<LAHF>
```

```
  Structured Extended Features=0x281<FSGSBASE,SMEP,ERMS>
  XSAVE Features=0x1<XSAVEOPT>
  VT-x: PAT,HLT,MTF,PAUSE,EPT,UG,VPID,VID,PostIntr
  TSC: P-state invariant, performance statistics
```

You probably didn't know that a simple CPU could have so many details and features, did you? But when you file a trouble report that advanced features don't work, a developer might respond by asking whether your CPU has a particular feature.

Here's why I'm not worried about the performance hit caused by the WITNESS option shown earlier: this box is pretty darn fast ❶ and supports a whole bunch of features important to modern CPUs ❷. While I certainly want all the performance I paid for, I also want to catch any problems when they happen. I want to be able to file good bug reports on those problems, so the developers will listen to my problem report. That's *why* I'm running a development version of FreeBSD that ships with WITNESS enabled, after all!

```
FreeBSD/SMP: Multiprocessor System Detected: 8 CPUs
```

Here, the kernel announces that it's found all eight CPU cores and is ready to manage them. I have CPU power to spare and a fair amount of memory as well.

```
❶ real memory  = 34359738368 (32768 MB)
❷ avail memory = 33207656448 ❸(31669 MB)
```

The *real* memory ❶ is the amount of RAM physically installed in the computer, while the *avail* memory ❷ is the amount of memory left over after the kernel is loaded. I have 31,669MB of RAM ❸ available for real work, which more than suffices for the load on this system.

```
❶ioapic0 <Version 2.0> ❷irqs 0-23 on motherboard
❸ioapic1 <Version 2.0> irqs 24-47 on motherboard
```

Here's a fairly typical device entry. This device is known as ioapic, and the kernel has found that this hardware is version 2.0 and has extra information associated with it ❷. What's more, we've found two devices of

that type, numbered 0 ❶ and 1 ❸. (All devices are numbered starting with zero.) You can find out more about the device by reading the man page for the device driver. Almost all—but not all—device drivers have man pages.

```
usbus0: EHCI version 1.0
usbus0 on ehci0
usbus0: 480Mbps High Speed USB v2.0
```

Not all device drivers print all their information on a single line. Here, we have a single device, usbus, that takes up three lines with just a single instance of the device. The only way to know that this is a single USB bus rather than three separate ones is to check the number of the device. All of these are for device number zero, so it's a single device.

```
❶ pci0: <simple comms> at device 22.0 (no driver attached)
❷ pcib8: <ACPI PCI-PCI bridge> irq 17 at device 28.0 on ❸pci0
❹ pci8: <ACPI PCI bus> on ❺pcib8
```

One interesting thing about the boot messages is that they display how your computer's components are attached to one another. Here, we have pci0 ❶, a PCI interface directly on the mainboard. Then, there's pcib8 ❷, PCI bridge number eight attached to pci0 ❸. We also find PCI bus pci8 ❹ attached to that PCI bridge ❺. As you read on, you'll find individual devices attached to that bus. You might not be equipped to do much with this information now, but you'll find that having it available will be valuable when you have to troubleshoot a problem.

```
❶em0: <Intel(R) PRO/❷1000 Network Connection> port 0xd000-0xd01f mem
0xfba00000-0xfba1ffff,0xfba20000-0xfba23fff irq 18 at device 0.0 on pci9
```

This entry shows em0, a network card of type em(4) ❶, and indicates that the card speaks gigabit Ethernet ❷. We also see all sorts of information about its memory address, IRQ, and PCI bus attachment.

Every device on your computer has one or more entries like these. Taken as a whole, they describe your computer's hardware in reasonable detail. If you boot in verbose mode, you'll see even more detail—probably far more than you want.

THE BOOT MESSAGES FILE

While the boot information is handy, chances are it'll disappear from the screen by the time you need it. For future reference, FreeBSD stores boot messages in the file */var/run/dmesg.boot*. This means that you can inspect your kernel's hardware messages even after your system has been up and running for months.

One key thing that the kernel displays in the boot messages is the device name for each piece of hardware. This is critical information for managing your system. Every piece of hardware has a device node name, and to configure it, you'll need to know that name. For example, earlier we saw an entry for an Ethernet card called em0. The card uses the em(4) driver, and the first device controlled by this driver has number zero. Your second device of this type would be em1, then em2, and so on.

Most devices that can be configured or managed have a device node entry somewhere under */dev*. For example, the first optical drive is represented by the file */dev/cd0*. These files are called *device nodes*, and they're a convenient way to address a particular piece of hardware. Most device nodes can't be directly accessed as a regular file; you can't cat(1) a device node or copy another file to it. However, device nodes are used as arguments to specialized programs. For example, the hard drive that showed up at boot as ada4 is the same as the device node */dev/ada4*. When you want to mount that hard drive, you can use the device node name and be sure you're getting that exact piece of hardware.

Multiuser Startup

Beyond single-user mode, you'll find multiuser mode. This is the standard operating mode for a Unix-like OS. If you're doing real work, your system is in multiuser mode.

When FreeBSD finishes inspecting the hardware and attaching all the device drivers appropriately, it runs the shell script */etc/rc*. This script mounts all filesystems, brings up the network interfaces, configures device

nodes, identifies available shared libraries, and does all the other work necessary to make a system ready for normal work. Most systems have different startup requirements; while almost every server needs to mount a hard drive, a web server's operating requirements are very different from those of a database server, even if it's running on absolutely identical hardware. This means that */etc/rc* must be extremely flexible. It achieves flexibility by delegating everything to other shell scripts responsible for specific aspects of the system.

The */etc/rc* script is controlled by the files */etc/defaults/rc.conf* and */etc/rc.conf*.

/etc/rc.conf, /etc/rc.conf.d, and /etc/defaults/rc.conf

Much like the loader configuration file, the configuration of */etc/rc* is split between two files: the default settings file, */etc/defaults/rc.conf*, and the local settings file, */etc/rc.conf*. Settings in */etc/rc.conf* override any values given in */etc/defaults/rc.conf*, exactly as with the loader.

The */etc/defaults/rc.conf* file is huge and contains quite a few variables, frequently called *knobs*, or *tunables*. We aren't going to discuss all of them, not only because knobs are added continually and such a list would be immediately obsolete but also because quite a few knobs aren't commonly used on servers. Almost everything in a standard FreeBSD system has one or more *rc.conf* knobs, from your keyboard map to TCP/IP behavior. For a complete, up-to-date list, read rc.conf(5). To change *rc.conf* settings, you can either use a text editor or sysrc(8).

sysrc(8)

While editing *rc.conf* by hand works just fine, in this age of cloud computing, it's not sustainable across large numbers of machines. If you must change dozens of servers, you need a reliable way to alter the system without either manually editing each server's config or resorting to sed/awk hackery.[2]

FreeBSD includes sysrc(8), a command line program to consistently and safely alter */etc/rc.conf* and friends from the command line. Additionally, sysrc(8) can display information about your system's non-default settings.

Start by using `-a` to ask sysrc(8) what it knows about your host.

```
# sysrc -a
clear_tmp_enable: YES
defaultrouter: 203.0.113.1
dumpdev: AUTO
keymap: us.dvorak.kbd
--snip--
```

You'll get a list of all non-default */etc/rc.conf* settings.

To have sysrc(8) enable a service, give it the variable name, an equals sign, and the new value.

```
# sysrc rc_startmsgs=NO
rc_startmsgs: YES -> NO
```

The variable `rc_startmsgs` is now set to `no`.

Remember that sysrc(8) is a tool for changing *rc.conf*, not for configuring FreeBSD. It does no validity checking. One of my very junior sysadmins really doesn't want Bert logging in, and he took some bad advice on how to prevent that.

```
# sysrc bert=no
```

While this code sets `bert="no"` in */etc/rc.conf*, this variable doesn't do anything. Remove it with the `-x` flag.

```
# sysrc -x bert
```

Many FreeBSD configuration files closely resemble *rc.conf*. You can use sysrc(8) to manage them by adding the `-f` flag and the file name.

```
# sysrc -af /boot/loader.conf
```

Should you edit *rc.conf* or use sysrc(8)? If you're making manual changes, then use whichever you prefer. Automation should err on the side of sysrc(8). This book mixes examples of both.

/etc/rc.conf.d/

If you use a server configuration system such as Puppet or Ansible, you might trust copying entire files more than editing them. Use */etc/rc.conf.d/* files to enable services through such tools.

To manage a service in */etc/rc.conf.d/*, create a file named after the service.

That is, to manage bsnmpd(8) you'd create */etc/rc.conf.d/bsnmpd*. Enable or disable that service in this file.

```
bsnmpd_enable=YES
```

I normally use Ansible's service enabling features that directly alter */etc/rc.conf* rather than */etc/rc.conf.d*, but use whatever you prefer.

The next few sections illustrate the types of things you can enable and disable in */etc/rc.conf*. Each appears in */etc/defaults/rc.conf* and can be overridden by an */etc/rc.conf* entry. Each variable appears with its default setting.

Startup Options

The following *rc.conf* options control how FreeBSD configures itself and starts other programs. These far-reaching settings affect how all other system programs and services run.

If you're having a problem with the startup scripts themselves, you might enable debugging on */etc/rc* and its subordinate scripts. This can provide additional information about why a script is or isn't starting.

```
rc_debug="NO"
```

If you don't need the full debugging output but would like some additional information about the */etc/rc process*, enable informational messages with `rc_info`:

```
rc_info="NO"
```

When the boot process hits multiuser startup, it prints out a message for each daemon it starts. Remove those messages with the `rc_startmsgs` option.

```
rc_startmsgs="NO"
```

Filesystem Options

FreeBSD can use memory as a filesystem, as we'll discuss in Chapter 13. One common use for this feature is to make */tmp* really fast by using memory rather than a hard drive as its backend. Once you've read Chapter 13, you might consider implementing this. Variables in *rc.conf* let you enable

a memory-backed */tmp* and set its size transparently and painlessly. You can also choose the options FreeBSD will use to complete the filesystem. (The impatient among you are probably wondering what the `-S` flag means. It means *disable soft updates*. If you have no idea what this means, either, wait for Chapter 11.) If you want to use a memory filesystem */tmp*, set `tmpmfs` to `YES` and set `tmpsize` to the desired size of your */tmp*.

```
tmpmfs="AUTO"
tmpsize="20m"
tmpmfs_flags="-S"
```

Another popular FreeBSD filesystem feature is its integrated encrypted partitions. FreeBSD supports two different filesystem encryption systems out of the box: GBDE and GELI. *GEOM-Based Disk Encryption (GBDE)* was FreeBSD's first encrypted filesystem designed for military-grade use. GELI is a little more friendly and complies with different standards than GBDE. (You definitely want to read Chapter 23 before enabling either of these!)

```
geli_devices=""
geli_tries=""
geli_default_flags=""
geli_autodetach="YES"
```

By default, FreeBSD mounts the root partition read-write upon achieving multiuser mode. If you want to run in read-only mode instead, you can set the following variable to `NO`. Many people consider this more secure, but a read-only root can interfere with operation of certain software, and it'll certainly prevent you from editing any files on the root partition!

```
root_rw_mount="YES"
```

When a booting FreeBSD attempts to mount its filesystems, it checks them for internal consistency. If the kernel finds major filesystem problems, it can try to fix them automatically with `fsck -y`. While this is necessary in certain situations, it's not entirely safe. (Be sure to read Chapter 11 very carefully before enabling this!)

```
fsck_y_enable="NO"
```

The kernel might also find minor filesystem problems, which it resolves on the fly using a *background fsck* while the system is running in multiuser

mode, as discussed in Chapter 11. There are legitimate concerns about the safety of using this feature in certain circumstances. You can control the use of background fsck and set how long the system will wait before beginning the background fsck.

```
background_fsck="YES"
background_fsck_delay="60"
```

Miscellaneous Network Daemons

FreeBSD includes many smaller programs, or daemons, that run in the background to provide specific services. We'll cover quite a few of these integrated services throughout the book, but here are a few specific ones that'll be of interest to experienced system administrators. One popular daemon is syslogd(8). Logs are a Good Thing. Logs are so very, *very* good that large parts of Chapter 21 are devoted to the topic of logging with, for, by, and on FreeBSD.

```
syslogd_enable="YES"
```

Once you've decided to run the logging daemon, you can choose exactly how it'll run by setting command line flags for it. FreeBSD will use these flags when starting the daemon. For all the programs included in *rc.conf* that can take command line flags, the flags are given in this format:

```
syslogd_flags="-s"
```

Another popular daemon is inetd(8), the server for small network services. (We cover inetd in Chapter 20.)

```
inetd_enable="NO"
```

Most systems use the Secure Shell (SSH) daemon for remote logins. If you want to connect to your system remotely over the network, you'll almost certainly need SSH services.

```
sshd_enable="NO"
```

While the SSH daemon can be configured via the command line, you're generally better off using the configuration files in */etc/ssh/*. See Chapter 20 for details.

```
sshd_flags=""
```

FreeBSD also incorporates extensive time-keeping software that functions to ensure the system clock remains synchronized with the rest of the world. You'll need to configure this for it to be useful; we'll cover that in Chapter 20.

```
ntpd_enable="NO"
ntpd_flags="-p /var/run/ntpd.pid -f /var/db/ntpd.drift"
```

In addition, FreeBSD includes a small SNMP daemon for use in facilities with SNMP-based management tools. We'll cover configuring SNMP in Chapter 21.

```
bsnmpd_enable="NO"
```

Network Options

These knobs control how FreeBSD configures its network facilities during boot. We'll discuss networking in Chapter 7.

Every machine on the internet needs a hostname. The hostname is the fully qualified domain name of the system, such as *www.absolutefreebsd.org*. Many programs won't run properly without this.

```
hostname=""
```

FreeBSD includes a few different integrated firewall packages. We're going to briefly cover the packet filter (PF) in Chapter 19. Enable and disable PF in *rc.conf*.

```
pf_enable="NO"
```

You might be interested in failed attempts to connect to your system over the network. This will help detect port scans and network intrusion attempts, but it'll also collect a lot of garbage. It's interesting to set this for a short period of time just to see what really happens on your network. (Then again, knowing what's *really* going on tends to cause heartburn.) Set this to `1` to log failed connection attempts.

```
log_in_vain="0"
```

Routers use ICMP redirects to inform client machines of the proper network gateways for particular routes. While this is completely legitimate, on some networks intruders can use this to capture data. If you don't need ICMP redirects on your network, you can set this option for an extremely tiny measure of added security. If you're not sure whether you're using them, ask your network administrator.

```
icmp_drop_redirect="NO"
```

If you *are* the network administrator and you're not sure whether your network uses ICMP redirects, there's an easy way to find out—just log all redirects received by your system to */var/log/messages*.[3] Note that if your server is under attack, this can fill your hard drive with redirect logs fairly quickly.

```
icmp_log_redirect="NO"
```

To get on the network, you'll need to assign each interface an IP address. We'll discuss this in some detail in Chapter 8. You can get a list of your network interfaces with the ifconfig(8) command. List each network interface on its own line, with its network configuration information in quotes. For example, to give your em0 network card an IP address of 172.18.11.3 and a netmask of 255.255.254.0, you would use:

```
ifconfig_em0="inet 172.18.11.3 netmask 255.255.254.0"
```

If your network uses DHCP, use the value `dhcp` as an IP address.

```
ifconfig_em0="dhcp"
```

Similarly, you can assign aliases to a network card. An alias is not the card's actual IP address, but the card answers for that IP address, as discussed in Chapter 8. FreeBSD supports hundreds of aliases on a single card, with *rc.conf* entries in the following form:

```
ifconfig_em0_aliasnumber="address netmask 255.255.255.255"
```

The alias numbers must be continuous, starting with 0. If there's a break in numbering, aliases above the break won't be installed at boot time. (This is a common problem, and when you see it, check your list of aliases.) For

example, an alias of 192.168.3.4 would be listed as:

```
ifconfig_em0_alias0="192.168.3.4 netmask 255.255.255.255"
```

Network Routing Options

FreeBSD's network stack includes many features for routing internet traffic. These start with the very basic, such as configuring an IP for your default gateway. While assigning a valid IP address to a network interface gets you on the local network, a default router will give you access to everything beyond your LAN.

```
defaultrouter=""
```

Network control devices, such as firewalls, must pass traffic between different interfaces. While FreeBSD won't do this by default, it's simple to enable. Just tell the system that it's a gateway and it'll connect multiple networks for you.

```
gateway_enable="NO"
```

Console Options

The console options control how the monitor and keyboard behave. You can change the language of your keyboard, the monitor's font size, or just about anything else you like. For example, the keyboard map defaults to the standard US keyboard, frequently called *QWERTY*. You'll find all sorts of keymaps in the directory */usr/share/syscons/keymaps*. I prefer the Dvorak keyboard layout, which has an entry there as *us.dvorak*. By changing the `keymap` knob to *us.dvorak*, my system will use a Dvorak keyboard once it boots to multiuser mode.

```
keymap="NO"
```

FreeBSD turns the monitor dark when the keyboard has been idle for a time specified in the `blanktime` knob. If you set this to `NO`, FreeBSD won't dim the screen. Mind you, new hardware will dim the monitor after some time as well, to conserve power. If your screen goes blank even if you've set the `blanktime` knob to `NO`, check your BIOS and your monitor manual.

```
blanktime="300"
```

```
blanktime="300"
```

FreeBSD can also use a variety of fonts on the console. While the default font is fine for servers, you might want a different font on your desktop or laptop. My laptop has one of those 17-inch screens proportioned for watching movies, and the default fonts look kind of silly at that size. You can choose a new font from the directory */usr/share/syscons/fonts*. Try a few to see how they look on your systems. The font's name includes the size, so you can set the appropriate variable. For example, the font *swiss-8x8.fnt* is the Swiss font, 8 pixels by 8 pixels. To use it, you would set the `font8x8` knob.

```
font8x16="NO"
font8x14="NO"
font8x8="YES"
```

You can use a mouse on the console, even without a GUI. By default, FreeBSD will try to autodetect your mouse type. If you have a PS/2 or USB mouse, chances are that it'll just work when you enable the mouse daemon, without any special configuration. Some older and more unusual types of mice require manual configuration, as documented in moused(8).

```
moused_enable="NO"
moused_type="AUTO"
```

You can also change the display on your monitor to fit your needs. If you have an odd-sized monitor, you can change the number of lines of text and their length to fit, change text colors, change your cursor and cursor behavior, and do all sorts of other little tweaks. You can get a full list of different options in man vidcontrol(1).

```
allscreens_flags=""
```

Similarly, you can adjust your keyboard behavior almost arbitrarily. Everything from key repeat speed to the effect of function keys can be configured, as documented in kbdcontrol(1).

```
allscreens_kbdflags=""
```

Other Options

This final potpourri of knobs might or might not be useful in any given environment, but they're needed frequently enough to deserve mention. For

example, not all systems have access to a printer, but those that do will want to run the printing daemon lpd(8). We brush up against printer configuration in Chapter 20.

```
lpd_enable="NO"
```

The sendmail(8) daemon manages transmission and receipt of email between systems. While almost all systems need to transmit email, most FreeBSD machines don't need to receive email. The `sendmail_enable` knob specifically handles incoming mail, while `sendmail_outbound_enable` allows the machine to transmit mail. See Chapter 20 for more details.

```
sendmail_enable="NO"
sendmail_submit_enable="YES"
```

One of FreeBSD's more interesting features is its ability to run software built for Linux. We discuss this feature in Chapter 17. Running Linux software isn't quite as easy as throwing this toggle, so don't enable Linux compatibility modes without reading that chapter first!

```
linux_enable="NO"
```

A vital part of any Unix-like operating system is shared libraries. You can control where FreeBSD looks for shared libraries. Although the default setting is usually adequate, if you find yourself regularly setting the `LD_LIBRARY_PATH` environment variable for your users, you should consider adjusting the library path instead. See Chapter 17 for more advice on the library path.

```
ldconfig_paths="/usr/lib /usr/local/lib"
```

FreeBSD has a security profile system that allows the administrator to control basic system features. You can globally disallow mounting hard disks, accessing particular TCP/IP ports, and even changing files. See Chapter 9 for details on how to use these.

```
kern_securelevel_enable="NO"
kern_securelevel="-1"
```

Now that you know a smattering of the configuration knobs FreeBSD supports out of the box, let's see how they're used.

The rc.d Startup System

FreeBSD bridges the gap between single-user mode and multiuser mode via the shell script */etc/rc*. This script reads in the configuration files */etc/defaults/rc.conf* and */etc/rc.conf*, and runs a collection of other scripts based on what it finds there. For example, if you've enabled the network time daemon, */etc/rc* runs a script written specifically for starting that daemon. FreeBSD includes scripts for starting services, mounting disks, configuring the network, and setting security parameters.

These scripts live in */etc/rc.d* and */usr/local/etc/rc.d*. I'd recommend reading a few of them if only to see how the *rc.d* system works.

Control these scripts with service(8).

The service(8) Command

All of the *rc.d* scripts are readable, and the way they fit together is pretty straightforward. When you have a problem, you can read the scripts to see how they work and what they do. But that's a lot like work, and most sysadmins have more interesting work to do. The service(8) command provides a friendly frontend to the *rc.d* scripts. You can use service(8) to see which scripts run automatically; to stop, start, and restart services; to check the status of a service; and more.

Listing and Identifying Enabled Services

Use the `-e` flag to service(8) to see the full path of all scripts that'll be run at system boot, in the order they'll be run.

```
# service -e
/etc/rc.d/hostid
/etc/rc.d/zvol
/etc/rc.d/hostid_save
/etc/rc.d/zfsbe
--snip--
/etc/rc.d/sshd
/etc/rc.d/sendmail
--snip--
```

This tiny host runs 23 scripts at boot.

One important detail here is the script name. You'll use the script name in other commands, like starting, stopping, and restarting services.

Managing Services

While it's entirely possible to restart, say, sshd(8) at the command line, a production host needs everything to run consistently. Best practice calls for using service(8) to manage processes. You'll need the script name as shown earlier, but without the directory path.

```
# service name command
```

For example, suppose I want to restart the sshd(8) service. According to the `service -e` output shown earlier, there's a script */etc/rc.d/sshd*. I strongly suspect this script manages sshd(8), but I want to be certain I don't accidentally restart the Stupidly Similarly named Harassment Daemon. This is where the `describe` command comes in. Let's ask service(8) to describe the *sshd* script.

```
# service sshd describe
Secure Shell Daemon
```

It's the right daemon. Let's restart it.

```
  # service sshd restart
❶ Performing sanity check on sshd configuration.
❷ Stopping sshd.
❸ Performing sanity check on sshd configuration.
❹ Starting sshd.
```

Restarting a service is a combination of "stop the service" and "start the service." This particular service does more than that, though. It starts by verifying the configuration file ❶ and then stopping the daemon ❷. It then reverifies the configuration ❸ and starts the daemon ❹. Why?

SSH handles remote access to this host. If the SSH service breaks, nobody can log into the host to fix the SSH service. Yes, you could use a remote KVM or IPMI or drive to the colocation facility, but any of these prolongs the outage. It's much better to verify that sshd(8) can *be* restarted before shutting it down. Many service scripts include this kind of safety check. If a service complains that it can't stop, read the output carefully to find out why.

The commands each service supports vary. The easiest way to get the full

list of commands a particular service supports is to give the service a bogus argument. Something like "bert" is pretty bogus.

```
# service sshd bert
/etc/rc.d/sshd: unknown directive 'bert'.
Usage: /etc/rc.d/sshd [fast|force|one|quiet](start|stop|restart|rcvar|enabled|
describe|extracommands|configtest|keygen|reload|status|poll)
```

You get a full list of commands this service supports, in two groups.

The first group, in square brackets, contains options for the commands. Here are the standard options. Use them as prefixes for the commands in the second group.

`fast` Do no checking (used during startup).

`force` Try harder.

`one` Start this service despite not being enabled in *rc.conf*.

`quiet` Only print service name (used during startup).

The second group, in parentheses, contains the following commands:

`start` Start the service.

`stop` Stop the service.

`restart` Stop and restart the service.

`rcvar` Print the *rc.conf* variables for this service.

`enabled` Return true in shell if enabled (for script use).

`describe` Print service description.

`extracommands` Show service-specific commands.

The `extracommands` command is very specific to the service and lists only the additional commands this service accepts. By default, the extra commands appear after the default commands. Here are some common extra commands:

`configtest` Parse the service's configuration file and stop if there's an error.

`reload` Perform a soft reload (usually via SIGHUP) rather than a restart.

`status` Determine whether service is running.

To determine exactly what a service's extra commands do, you need to

read the service script.

We'll look at *rc.d* in more detail in Chapter 17, when we discuss customizing and writing your own *rc.d* scripts.

System Shutdown

FreeBSD makes the *rc.d* startup system do double duty; not only must it handle system startup, it must also shut all those programs down when it's time to power down. Something has to unmount all those hard drives, shut down the daemons, and clean up after doing all the work. Some programs don't care whether they're unceremoniously killed when the system closes up for the night—after all, after the system goes down, any clients connected over SSH will be knocked off and any half-delivered web pages remain incomplete. Database software, however, cares very much about how it's turned off, and unceremoniously killing the process will damage your data. Many other programs that manage actual data are just as particular, and if you don't let them clean up after themselves, you'll regret it.

When you shut down FreeBSD with the shutdown(8), halt(8), or reboot(8) commands, the system calls the shell script */etc/rc.shutdown*. This script calls each *rc.d* script in turn with the `stop` option, reversing the order they were called during startup, thereby allowing server programs to terminate gracefully and disks to tidy themselves up before the power dies.

Serial Consoles

All this console stuff is nice, but when your FreeBSD system is in a colocation facility on the other side of the country or on another continent, you can't just walk up to the keyboard and start typing. Many data centers won't have room for a keyboard or monitor. And how do you reset the machine remotely when it won't respond to the network? Using a serial console to redirect the computer's keyboard and video to the serial port instead of the keyboard and monitor helps with all of these problems.

Serial consoles can be physical, such as a serial port on the back of a computer. By hooking up a standard null modem cable to the serial port and attaching the other end to another computer's serial port, you can access the first system's boot messages from the second computer.

They might also be virtual, as provided by IPMI's *Serial-over-LAN (SOL)* protocol. Rather than a null modem cable, you'll need to set up the IPMI interface and use special software to configure and access the virtual serial port.

Before we set up a port, though, let's talk about serial port protocol.

Serial Protocol

Some of the first computer consoles were serial ports connected to teletypes. Serial has been around a long time and has evolved over the decades. Unlike modern protocols, serial lines do not autonegotiate. You must configure both sides of a serial link to the exact same settings. A configuration mismatch will cause either a blank screen or gibberish.

Original serial lines worked at low speeds. Many of the serial cables remain basically the same, but we've developed better software and hardware to stick at each end that allows us to transmit data much faster. Where old serial connections ran at 300 bits per second (baud), a whole bunch of modern hardware can run at 115,200 baud. Across hardware platforms, though, the common standard is 9600 baud, which is FreeBSD's default console speed. A baud rate of 9600 is enough to carry whole screens of text at a comfortable speed.

Stick with 9600 baud for physical connections, unless you can't. Some modern hardware doesn't support 9600 baud. Some claim to support 9600 baud, but don't. I've worked with devices hardcoded to 115,200 baud. Anything that fails or flat-out refuses to do 9600 baud is busted by design, but we often don't control the choice of hardware. Changing the serial console speed for reasons other than hardware limitations makes your connection more fragile, and if you're using the console, you're in no mood for fragility. When I mention changing the port speed, that's for use only when you have to.

SOL connections aren't physical wires, so you don't have to worry about line noise. You can safely run them at higher speeds.

Serial protocols also include a whole bunch of settings beyond their speed. It's possible to muck with them, but the standard settings of 8 data bits, no parity, and 1 stop bit are the most widely used. You can't change these in FreeBSD without recompiling the kernel, so don't muck with them.

With that in mind, let's set up a console.

Physical Serial Console Setup

No matter what sort of serial console you have, you'll need to plug into it correctly to make it work. You'll need a null modem cable, available at any computer store or from online vendors. While the gold-plated serial cables are not worth the money, don't buy the cheapest cable you can find either; if you have an emergency and need the serial console, you're probably not in the mood to endure line noise![4]

Plug one end of the null modem cable into the serial console port on your FreeBSD server—by default the first serial port (COM1 or uart0, depending on what operating system you're used to). You can change this with a server.

Plug the other end of your null modem cable into an open serial port on another system. I recommend either another FreeBSD (or other Unix) system or a terminal server, but you can use a Windows box if that's all you have.

If you have two FreeBSD machines at a remote location, make sure that they each have two serial ports. Get two null modem cables and plug the first serial port on each box into the second serial port of the other machine. That way, you can use each machine as the console client for the other. If you have three machines, daisy-chain them into a loop. By combining twos and threes, you can get serial consoles on any number of systems. I've worked data centers with 30 or 40 FreeBSD machines, where installing monitors was simply not practical, and we used serial consoles to great effect. Once you have a rack or two of servers, however, investing in a terminal server is a really good idea. You can find them cheaply on eBay.

Another option is to use two DB9-to-RJ45 converters, one standard and one crossover. These allow you to run your console connections over a standard CAT5 cable. If you have a lights-out data center where human beings are not allowed, you can have your serial consoles come out near your desk, in your warm room, or anywhere else your standard Ethernet-style patch panels reach. Most modern data facilities are better equipped to handle Ethernet than serial cables.

IPMI Serial Console Setup

The *Intelligent Platform Management Interface (IPMI)* is a standard for managing computer systems at a hardware level. IPMI runs separately from the operating system, using a small device called a *baseboard management controller (BMC)*. Essentially, the BMC acts as your remote hands and eyes to control the server. To use an IPMI console, you'll need to configure both the BMC and the host's BIOS or UEFI.

I'll try to orient you here, but the best resource for configuring BMC or UEFI is your hardware manual.[5]

BMC Setup

A server's BMC has its own IP address and normally gets a dedicated Ethernet port on the mainboard. Each vendor gets to design its own BMC in a way that conforms to its own biases. This means that configuring the BMC is way, way beyond the scope of this book, but here are a few hints.

You configure most BMCs through a web interface. Before you can access the web interface, though, the BMC needs an IP address. Set most BMC IP information in the BIOS or UEFI firmware's setup menu. Once you get in the management interface, configure a username and password. Remember them.

A usual BMC also includes functions such as power cycling the main system, remote console access via some sort of downloaded application (often Java), virtual media, and more.

Never forget that the BMC is a small embedded computer running a web server and that it was written by some overworked corporate employee charged with building the minimum viable product. The BMC wasn't tested for how it performed after several months of uptime. If it gives you even a sneeze of trouble, reboot it. No, you don't have to power cycle the whole computer; there's usually a "BMC Reset" or "Unit Reboot" menu option somewhere in the web interface.

If the BMC supports an applet-based console, why use a serial console? Because the BMC console is applet-based and BMC firmwares are rarely updated. I have quite a few BMC consoles that work only with obsolete, insecure[6] versions of Java. Using them requires overriding security warnings and repeatedly clicking the "Yes, I know I'm an idiot, do it anyway" box. I have to keep a virtual machine with this insecure Java version specifically to access those consoles. The applet-based console doesn't support copy and

paste, and is often very laggy.

IPMI works better than the Java console applet over slower connections. I can copy and paste. Also, I can use the IPMI console from the command line, from any modern operating system.

While you're in the BMC setup, locate the option to launch SOL. That brings up an applet to connect to the host's SOL interface, which will help you test your serial console configuration.

UEFI/BIOS Serial Console Configuration

Once your BMC is ready, you must configure the server hardware to direct a serial port to the BMC. Go to the hardware's Setup menu, where you configure your UEFI or BIOS. Somewhere in that maze of twisty little options, you'll find something like "Serial Port Console Redirection."

A vital question here is, how many serial ports does your host have? Maybe it has none. Maybe it has several. You can choose to redirect one of those ports or add an additional, virtual port. I encourage you to leave your existing serial ports alone and add a virtual port dedicated to SOL. It's probably called something like "SOL Console Redirection." Enable it, and go into the settings for that port.

Here are some settings I find helpful for FreeBSD and SOL:

Terminal type vt100

Data bits 8

Parity none

Stop bits 1

Flow control none

The tricky part is the baud, speed, or bits-per-second setting. Stick with the default speed, but make a note of it. You'll need the speed to connect.

Now that you have a serial console, set up FreeBSD.

Configuring FreeBSD's Serial Console

As FreeBSD boots, the loader decides where to print console messages and where to accept input from. While this defaults to the monitor and keyboard, with a few tweaks, you can redirect the console to a serial port.

The serial console won't grant BIOS access, but you can tweak the FreeBSD boot itself in almost any way. You can configure a serial console in either the first- or second-stage boot loader.

A first-stage boot loader gets you console access at the earliest possible moment but requires you use the first serial port as a console. Changing the port requires recompiling the kernel. The first-stage boot loader allows you to perform tasks like choose which disk you're going to load the second-stage loader from—essentially, to boot from a disk other than the disk the BIOS or UEFI selected. This is undeniably useful, but very few users need this.

The second-stage boot loader can use any serial port as a console, but the first bit of output you'll get is the boot menu discussed in "The Loader Prompt" on page 55. For most of us, that's perfectly acceptable.

Console Options

FreeBSD's default configuration uses the monitor and keyboard as the console. You can choose to switch to only the serial console or to use a dual console. Choose which with the */boot/loader.conf* option `console`.

A serial-only console prevents some random colocation employee from power cycling your box, plugging in a monitor, and dinking with the menu. Yes, they could still work mayhem from the first-stage loader or boot off of USB, but that requires greater skill. Set the *console* variable to `comconsole` to use only the serial port as a console.

```
console="comconsole"
```

For most deployments, I recommend a dual console. Dual consoles show console activity on both the serial port and the monitor. You can use either the standard or the serial console as needed. Specify a dual-console configuration by listing both `comconsole` and `vidconsole`.

```
console="comconsole vidconsole"
```

If you're in a server-room situation, you might want to switch back and forth between a standard console and a serial console. I generally manage large arrays of FreeBSD systems via the serial console but leave the video console in place in case of trouble.

The console won't be effective until after a reboot. You can see whether FreeBSD put its console on a serial port by checking the boot messages.

```
uart0: <16550 or compatible> port 0x3f8-0x3ff irq 4 flags 0x10 on acpi0
uart0: console (9600,n,8,1)
```

The second line shows that the serial port uart0 is configured as a console, using the default settings. We'll look at those settings in "Using Serial Consoles" on page 79.

Advanced Console Options

In addition to enabling the console, you can adjust the console's port and the speed.

Maybe I need to use the second serial port for the console. Perhaps the first serial port has something plugged into it, or maybe the second port is the virtual SOL port. Serial ports use the uart(4) device driver. Remember that FreeBSD devices start numbering at zero, while COM ports start numbering at 1. COM1 is uart0, COM2 is uart1, and so on. You'll need the port's base I/O port, which you can get from the system bootup messages.

```
# grep uart /var/run/dmesg.boot
uart0: <16550 or compatible> port ❶0x3f8-0x3ff irq 4 flags 0x10 on acpi0
uart1: <16550 or compatible> port ❷0x2f8-0x2ff irq 3 on acpi0
```

The first number after the word port is the base I/O port ❶. The base address of COM2, or uart1, is 0x2f8. Set `comconsole_port` to this value ❷.

```
comconsole_port="0x2f8"
```

Your console is now on serial port COM2.

If my serial connection won't do 9600 baud, I can change the port speed with the `comconsole_speed` option.

```
comconsole_speed="115200"
```

On a physical port, don't increase the port speed just because you can.

Using Serial Consoles

Now that you have both physical and software set up, configure your client to access the serial console. The key to using a serial console is to remember the following settings:

- Speed (9600 baud, or whatever your hardware is set to)
- 8 bits
- No parity
- 1 stop bit

The way you access a serial line depends on whether it's a physical line or an IPMI SOL connection.

Physical Serial Lines

Connect your client to the other end of the serial line. You can find terminal emulators for Microsoft platforms (PuTTY being the most famous), macOS, and almost any other operating system. Once upon a time, I used a Palm handheld with a serial cable to access serial consoles. Enter the correct value settings into the terminal emulator, and the serial console will "just work."

FreeBSD accesses serial lines with tip(1), a program that allows you to connect to remote systems in a manner similar to telnet. To run tip, do this as root:

```
# tip portname
```

A port name is shorthand for specifying the serial port number and speed to be used on a serial port. The file */etc/remote* contains a list of port names. Most of the entries in this file are relics of the eon when UUCP was the major data transfer protocol and serial lines were the norm instead of the exception.[7] At the end of this file, you'll see a few entries like:

```
# Finger friendly shortcuts
uart0|com1:dv=/dev/cuau0:br#9600:pa=none:
uart1|com2:dv=/dev/cuau1:br#9600:pa=none:
--snip--
```

The `uart` entries are the standard Unix-type device names, while the `com` names were added for the convenience of people who grew up on x86 hardware.

Assume that you have two FreeBSD boxes wired back-to-back, with each one's serial port 1 null-modemed into serial port 2. Both machines are configured to use a serial console. You'll want to connect to your local serial port 2 to talk to the other system's serial console:

```
# tip uart1
connected
```

You're in!

To disconnect the serial console, press ENTER and then type the disconnect sequence "tilde-dot" at any time.

```
~.
```

You'll be gracefully disconnected. (This also works in the OpenSSH client.)

The tip(1) program uses the tilde (~) as a control character. Read the man page for a full list of things you can do with it.

IPMI SOL Connections

You'll need a SOL client to connect to your IPMI serial port. The quickest way to test your configuration is probably with the SOL client applet included in your BMC. While that client has most of the disadvantages of the console applet, it's a good place to test. If the BMC SOL client doesn't work, check your SOL settings and FreeBSD configuration. Verify that the SOL client is set to use the same speed you set in the hardware and in FreeBSD. If it doesn't work but all your settings appear to match, reboot the BMC. Once it works, you can use SOL from another host.

The standard IPMI SOL client is IPMItool (*https://sourceforge.net/projects/ipmitool/*), available as the ipmitool package. (Chapter 15 discusses packages.) IPMItool can interact with your BMC over the network, granting you all of the BMC functions without logging into a clunky web interface. You can reboot the host, check hardware alarms and sensors, and more, all from the command line. But for the moment, we'll stick with the SOL console. Use the BMC's hostname or IP, the username, and the password to log into SOL.

```
# ipmitool -H bmc -U username -I lanplus sol activate
```

Here, I log into my web server's BMC, with the hostname www-bmc, using the username "bert."

```
# ipmitool -H www-kvm -U bert -I lanplus sol activate
```

Enter the password at the prompt, and the SOL will acknowledge your login.

```
[SOL Session operational.  Use ~? for help]
```

We have a console. Probably. Let's do the final test.

Working at the Console

The real test of a serial console is whether or not you can get data across it. Once you have your console connected, hit ENTER.

```
FreeBSD/amd64 (www) (ttyu2)

login:
```

FreeBSD permits logins on serial consoles by default. Log in to the host and reboot it, and you'll get the usual console messages.

```
Jul 13 11:48:24 Stopping cron.
Stopping sshd.
Stopping devd.
Writing entropy file:.
Writing early boot entropy file:.
Terminated
.
Jul 13 11:48:24 zfs1 syslogd: exiting on signal 15
Waiting (max 60 seconds) for system process `vnlru' to stop... done
Waiting (max 60 seconds) for system process `bufdaemon' to stop... done
Waiting (max 60 seconds) for system process `syncer' to stop...
Syncing disks, vnodes remaining... 0 0 0 done
All buffers synced.
```

There will be a long pause while the system runs its BIOS routines and hands control over to the serial console. Just about the time you decide that the machine is never coming back up, you'll get the loader menu. Congratulations! You're using a serial console. Press the spacebar to interrupt the boot just as if you were at the keyboard.

It doesn't matter how far away the system is; you can change your

booting kernel, get a verbose boot, bring it up in single-user mode, or manually fsck the hard drive—whatever. A software serial console might not show you the BIOS, but chances are that's set up correctly already. Once you've used a serial console for a while, it won't matter whether the machine is on the other side of the world or the other side of the room; getting out of your chair merely to access the console will feel like too much work.

If a system in a remote location entirely locks up, you can connect to your serial console and have the "remote hands" at the colocation facility power-cycle the system. It might not be good for your computer, but it's also not good for it to be locked up. With the serial console, you can boot into single-user mode and fix the problem by digging through the logs and whatever other troubleshooting you feel capable of. We'll discuss troubleshooting this sort of problem in Chapter 24.

Now that you understand how FreeBSD starts up and shuts down, let's look at some basic tools you can use to ensure that your system will continue to run even after you've been experimenting with it.

5
READ THIS BEFORE YOU BREAK SOMETHING ELSE! (BACKUP AND RECOVERY)

The most common cause of system failure is those pesky humans, but hardware and operating systems also fail. Hackers learn new ways to disrupt networks and penetrate applications, and you'll inevitably need to upgrade and patch your system on a regular basis. (Whether or not you *will* upgrade and patch is an entirely separate question.) Any time you touch a system, there's a chance you'll make a mistake, misconfigure a vital service, or otherwise totally ruin your system. Just think of how many times you've patched a computer running any OS and found something behaving oddly afterward! Even small system changes can damage data. You should, therefore, always assume that the worst is about to happen. In our case, this means that if either the hardware or a human being destroys the data on your hard drive, you must be able to restore that data.

We'll start with system backups and managing tape drives using tar(1) and then review recording system behavior with script(1). Finally, should you suffer a partial or near-total disaster, we'll consider recovering and rebuilding with single-user mode and the install media.

System Backups

You need a system backup only if you care about your data. That isn't as inane as it sounds. The real question is, "How much would it cost to replace my data?" A low-end tape backup system can run a few hundred dollars.

How much is your time worth, and how long will it take to restore your system from the install media? If the most important data on your hard disk is your web browser's bookmarks file, a backup system probably isn't worth the investment. But if your server is your company's backbone, you'll want to take this investment very seriously.

Online backups can easily be damaged or destroyed by whatever ruins the live server. Proper backups are stored safely offline. Tools like rsync(1), and even ZFS replication, don't create actual backups; they create convenient online copies.

A complete backup and restore operation requires a tape drive and media. You can also back up to files, across the network, or to removable media, such as CDs or DVDs. Many people use removable multiterabyte hard drives connected via USB 3 for backups. Despite our best efforts, tape is still an important medium for many environments.

Backup Tapes

FreeBSD supports SCSI and USB tape drives. SCSI drives are the fastest and most reliable. USB tape drives are not always standards-compliant and hence not always compatible with FreeBSD. Definitely check the release notes or the FreeBSD mailing list archives to confirm that your tape drive is compatible with FreeBSD.

Once you've physically installed your tape drive, you need to confirm that FreeBSD recognizes it. The simplest way is to check the */var/run/dmesg.boot* file for *sa* devices (see Chapter 4). For example, the following three lines from *dmesg.boot* describe the SCSI tape device in this machine:

```
❶sa0 at mps0 bus 0 ❷scbus0 ❸target 3 lun 0
sa0: ❹<QUANTUM ULTRIUM 5 3210> Removable Sequential Access SPC-4 SCSI device
sa0: Serial Number HU1313V6JA
sa0: ❺600.000MB/s transfers
sa0: Command Queueing enabled
```

Of all the information we have on this tape drive, the most important is that your FreeBSD system knows this device as sa0 ❶. We also see that it's attached to the SCSI card mps0 ❷ at SCSI ID 3 ❸, and we see the drive's model number ❹ as well as the fact that it can run at 600MB per second ❺.

Tape Drive Device Nodes, Rewinding, and Ejecting

Tape is a linear storage medium. Each section of tape holds a particular piece of data. If you back up multiple chunks of data to tape, avoid rewinding after each backup operation. Imagine that you wrote a backup of one system to tape, rewound the tape, and backed up another system. The second backup would overwrite the first because it used the same chunk of tape. When you run multiple backups on a single tape, use the appropriate device node to ensure you don't rewind the tape between tasks.

As with many Unix devices with decades worth of history, the way you access a tape drive controls how it behaves. Tape drives have several different device nodes, and each one makes the tape drive behave differently. The most basic tape-control mechanism is the device node used to access it. Normal tape drives have three nodes: */dev/esa0*, */dev/nsa0*, and */dev/sa0*.

Tapes are sequential access devices, and to access certain data on a particular section of tape, you must roll the tape back to expose that section. To rewind or not to rewind is an important question.

NOTE

The behavior of different tape device nodes varies between operating systems. Different versions of Unix, with different tape management software, handle tapes differently. Do not make assumptions with your backup tapes!

If you use the node name that matches the device name, the tape drive will automatically rewind when your command finishes. Our sample SCSI tape drive has a device name of sa0, so if you run a command using */dev/sa0* as the device node, the tape will rewind when the command finishes.

If you don't want the tape to automatically rewind when the command completes, stop it from rewinding by using the node name that starts with *n*. Perhaps you need to append a second backup from a different machine onto the tape or you want to catalog the tape before rewinding and ejecting. In our example, use */dev/nsa0* to run your command without rewinding.

To automatically eject a tape when a command finishes, use the node that begins with *e*. For example, if you're running a full system backup, you probably want the tape to eject when the command finishes so the operator

can put the tape in a case to ship offsite or place in storage. Our example uses the */dev/esa0* device name to eject the tape when the command finishes. Some tape drives might not support automatic ejection; they'll require you to push the physical button to work the lever that winches the tape out of the drive. The easiest way to identify such a drive is to try to eject it via the device node and see what happens.

The $TAPE Variable

Many programs assume that your tape drive is */dev/sa0*, but that isn't always correct. Even if you have only one tape drive, you might want to eject it automatically (*/dev/esa0*) or not to rewind it upon completion (*/dev/nsa0*).

Many (but not all) backup-related programs use the environment variable `$TAPE` to control which device node they use by default. You can always override `$TAPE` on the command line, but setting it to your most commonly used choice can save you some annoyances later.

Tape Status with mt(1)

Now that you know how to find your tape drive, you can perform basic actions on it—such as rewinding, retensioning, erasing, and so on—with mt(1). One basic thing mt(1) does is check a tape drive's status, as follows:

```
# mt status
Mode      Density              Blocksize      bpi      Compression
Current: ❶0x25:DDS-3          variable       97000   ❷DCLZ
---------available modes---------
0:        0x25:DDS-3           variable       97000    DCLZ
1:        0x25:DDS-3           variable       97000    DCLZ
2:        0x25:DDS-3           variable       97000    DCLZ
3:        0x25:DDS-3           variable       97000    DCLZ
--------------------------------
            ❸ Current Driver State: at rest.
---------------------------------
File Number: 0  Record Number: 0         Residual Count 0
```

You don't have to worry about most of the information here, but if you want to go through it line by line, the mt(1) man page contains a good description of all the features. At the very least, if the command returns anything useful, this means mt(1) can find your tape drive.

One of the first things we see is the drive density ❶. Older drives can have tapes of different densities for different purposes, but modern tape drives pack data as tightly as possible. This particular tape drive is a DDS-3 model; while you could choose to use another density, all the choices it offers are DDS-3. We also see that this tape drive offers hardware compression with the DCLZ algorithm ❷. Near the bottom, we see what the tape drive is doing right now ❸.

The `status` command might give you different sorts of messages. The most problematic is the one that tells you that your tape drive is not configured:

```
#mt status
mt: /dev/nsa0: Device not configured
```

This means that you don't actually have a tape at the device node that your `$TAPE` variable points at. You can experiment with device nodes and mt(1) by using the `-f` flag to specify a device node (for example, `mt -f /dev/nsa1 status`), although you should get correct information from *dmesg.boot*. If you're sure that your device node is correct, perhaps you don't have a tape inserted into the drive or the tape drive needs cleaning.

Another response you might get from `mt status` is `mt: /dev/nsa0: Device busy`. You asked for the status of your tape, and it replied, "I can't talk now. I'm busy." Try again later, or check `ps -ax` to see what commands are using the tape drive. When you're working with actual tape, only one program instance can access it at a time. You can't list the contents of a tape while you're extracting a file from that tape.

Other Tape Drive Commands

You can do more with a tape drive than just check to see whether it's alive. The mt(1) subcommands I use most frequently are `retension`, `erase`, `rewind`, and `offline`.

Tapes tend to stretch, especially after they're used the first time. (I know perfectly well that modern tape vendors all claim that they prestretch their tapes or that their tapes can't be stretched, but that claim and two slices of bread will get you a bologna sandwich.) *Retensioning* a tape is simply running the tape completely through, both forward and back, with the command `mt`

`retension`. Retensioning takes all the slack out of the tape and makes backups more reliable.

Erasing removes all data from a tape. This isn't a solidly reliable erasure, which you'd need to conceal data from a data recovery firm or the IRS; `mt erase` simply rolls through the tape and overwrites everything once. This can take a very long time. If you want to erase the tape quickly, you can use `mt erase 0` to simply mark the tape as blank.

The `mt rewind` command rolls a tape back to the beginning, same as accessing the device through its default device node.

When you *offline* a tape, you rewind and eject it so that you can put a new tape in. The command is, oddly enough, `mt offline`.

Now let's get some data on that tape.

TAPE DRIVE TEMPERAMENT

Not all tape drives support all functions. Older tape drives in particular are quite touchy, even crotchety, requiring very specific settings to work acceptably. If you have a problem with a particular drive, check the *FreeBSD-questions* mailing list archive for messages from others with the same problem. You'll probably find your answer there.

BSD tar(1)

The most popular tool for backing up systems to tape is tar(1). *Tar* is short for "tape archiver"—it's literally written for backups. FreeBSD also includes dump(8), but that's intended only for UFS filesystems that don't use soft updates journaling. You'll certainly encounter other backup tools too, such as pax and cpio, as well as network-based backup tools, like Amanda, Bacula, and Tarsnap. These tools are well suited for certain environments but aren't as universal as tar. Tar is a common standard recognized by almost every operating system vendor; you can find tar for Windows, Linux, Unix, BSD, macOS, AS/400, VMS, Atari, Commodore 64, QNX, and just about everything else you might encounter.

You can use tar(1) to back up to tape or to a file. A backup file containing tarred files is called a *tarball*. It's very fast and easy to restore just one file or a subset of files from a tarball. It's also easy to restore a portion of your backup from tape, but it's not nearly as fast.

FreeBSD uses a version of tar called *bsdtar*. Bsdtar can behave completely consistently with GNU tar and can also behave in strict accordance with POSIX tar. If you're at all concerned about the differences between GNU tar, POSIX tar, and bsdtar, read tar(1) for all the gory details. Bsdtar is built on libarchive(3), a library specifically for creating and extracting backup archives. Thanks to libarchive, bsdtar can extract files from anything from a traditional tape backup to an ISO image, all with the same interface. If you need to open an RPM, a zip file, or almost any other archive, bsdtar is your friend.

Bsdtar, like any other tar(1), can be dumb. If your filesystem is corrupt in any way, bsdtar will back up what it thinks you asked for. It will then happily restore files that were damaged during the original backup, overwriting working-but-incorrect files with not-working-and-still-incorrect versions. These sorts of problems rarely happen, but tend to be unforgettable when they do.

FILESYSTEM COHERENCE

No matter what backup software you use, files can change as you're trying to back them up. Log files constantly add stuff at the end, while databases can change anywhere in the file. Filesystem snapshots are always consistent, and both UFS (Chapter 11) and ZFS (Chapter 12) support them. Never back up live databases; instead, dump the database to an archive file and back up that archive.

tar Modes

Tar can perform several different actions, controlled by the command line flags. These different actions are called *modes*. You'll need to read the man

page for a complete description of all tar modes, but the following list describes the most commonly used ones.

Create an Archive

Use *create mode* (`-c`) to create a new archive. Unless you specify otherwise, this flag backs up everything to your tape drive (`$TAPE`, or */dev/sa0* if you haven't set `$TAPE`). To back up your entire system, you'd tell tar to archive everything from the root directory down:

```
# tar -c /
```

In response, your tape drive should light up and, if your tape is big enough, eventually present you with a complete system backup. Many modern hard drives are bigger than tape drives can hold, however, so it makes sense to back up only the vital portions of your system. For example, if the only files on your computer that you need are in the directories */home* and */var*, you could specify those directories on the command line:

```
# tar -c /home /var
```

List Archive Contents

List mode (`-t`) lists all the files in an archive. Once you've created an archive, you can use this mode to list the tape's contents.

```
# tar -t
.
.snap
dev
tmp
--snip--
```

This list includes all the files in your backup and might take a while to run. Note that the initial slashes are missing from filenames; for example, */tmp* shows up as *tmp*. This becomes important during restores.

Extract Files from Backup

In *extract* mode, tar retrieves files from the archive and copies them to the disk. (This is also called *untarring*.) Tar extracts files in your current location; if you want to overwrite the existing */etc* directory of your system

with files from your backup, go to the root directory first. On the other hand, to restore a copy of */etc* in my home directory, I'd go to my home directory first.

```
# cd /home/mwlucas
# tar -x etc
```

Remember when I said that the missing initial slash would be important? Here's why. If the backup included that initial slash, tar would always extract files relative to the root directory. The restored backup of */etc/rc.conf* would always be written to */etc/rc.conf*. Without the leading /, you can recover the file anywhere you want; the restored */etc/rc.conf* can be resurrected as */home/mwlucas/etc/rc.conf*. If I'm restoring files from a machine that's been decommissioned, I don't want them to overwrite files on the current machine; I want them placed elsewhere so they won't interfere with my system.

Verify Backups

Once you have a backup, you probably want to confirm that it matches your system. *Diff* mode (`-d`) compares the files on tape to the files on disk. If everything on the tape matches the system, `tar -d` runs silently. A perfect match between tape and system is *not* normal, however. Log files usually grow during the backup process, so the log files on tape shouldn't match the files on disk. Similarly, if you have a database server running, the database files might not match. If you truly want a perfect backup (also called a *cold backup*), you'll need to shut down to single-user mode before taking the backup. You must decide which errors you can live with and which require correction.

Other tar Features

Tar has several other features that can make it more friendly or useful. These include verbose behavior, different types of compression, permissions restore, and the most popular option—alternate storage.

Use Non-default Storage

Tar feeds everything to your tape drive by default, but the `-f` flag allows you

to specify another device or file as the destination. In all of the preceding examples, either I'm using the default tape drive, */dev/sa0*, or I've set `$TAPE`. If I have neither of these, I'd need to specify a tape drive with `-f`:

```
# tar -c -f /dev/east0 /
```

You can also back up to a file (or tarball) instead of using a tape. Source code distributed via the internet is frequently distributed as tarballs. Use the same `-f` flag to specify a filename. For example, to back up the chapters of this book as they were written, I ran the following every so often to create the tarball *bookbackup.tar*:

```
#tar -cf bookbackup.tar /home/mwlucas/af3e/
```

This file can easily be backed up on machines elsewhere—so even if my house burns down, the book would be safe. I could then run phone and power lines to the neighbor's house, borrow a laptop, find an open wireless access point, run `tar -xf bookbackup.tar`, and work amidst the charred timbers while waiting for the insurance company. (I couldn't do much else at the time, anyway.)

Verbose

Tar normally runs silently unless it encounters an error. This is good most of the time (who wants to read the complete list of files on the server every time a backup runs?), but sometimes you like to have the warm fuzzy feeling of watching a program do its work. Adding the `-v` flag makes tar print the name of each file it processes. You can use the verbose flag to create a complete list of all the files being backed up or restored. In a routine backup or restore, this verbosity makes errors difficult to see.

Compression

Bsdtar inherits support for every compression algorithm libarchive(3) understands. We'll cover a few you might use to create archives, in order from the most to least desirable. Bsdtar supports many more compression algorithms, but you wouldn't normally use them to create an archive.

XZ Compression

The XZ compression algorithm is the new hotness. Enable it with `-J`. Non-FreeBSD hosts might need to pipe restores through xz(1) to read them. Tarballs compressed with XZ usually end in *.txz*.

bzip Compression

FreeBSD's tar supports bzip compression, which shrinks files even more tightly than gzip, with the `-j` flag. Bzip uses more CPU time than gzip, but these days, CPU time is not nearly as limited as when gzip came out. Not all versions of tar support bzip compression, either. If you'll only be reading your files on a FreeBSD machine or you're comfortable installing bzip on other platforms, use the `-j` flag. Most tarballs compressed with bzip(1) end in *.tbz*.

gzip Compression

The gzip flag (`-z`) runs the files through the gzip(1) compression program on their way to or from the archive. Compressed tarballs usually have the extension *.tar.gz*, *.tgz*, or, on rare occasion, *.taz*. Compression can greatly reduce the size of an archive; many backups shrink by 50 percent or more with compression. While all modern versions of tar support gzip, older versions don't, so if you want absolutely everybody to be able to read your backup, don't use `-z`.

Primordial Unix Compression

In contrast, all Unix versions of tar can use the `-Z` flag to compress files with compress(1). The compress program isn't as efficient as gzip, but it does reduce file size. Every implementation of tar you're likely to encounter supports compress(1). Tarballs compressed with `-Z` have the extension *.tar.Z*.

Permissions Restore

The `-p` flag restores the original permissions on extracted files. By default, tar sets the owner of an extracted file to the username that's extracting the file. This is fine for source code, but for system restores, you really want to restore the file's original permissions. (Try to restore these permissions by hand some time; you'll learn quite a bit about why you should have done it right the first time.)

COMPRESSION AND FREEBSD TAR

FreeBSD's libarchive autodetects compression types used in backups. While you must specify your desired compression when creating an archive, you don't need to give a compression algorithm when extracting. Let tar(1) determine the compression type, and it will Do The Right Thing automatically, even if the archive is compressed with an algorithm you've never seen before.

And More, More, More . . .

Tar has many, many more functions to accommodate decades of changes in backups, files, filesystems, and disks. For a complete list of functions, read man tar(1).

Recording What Happened

You can now back up your entire system as well as track changes in a single file. All that remains is to track what's happening on the screen in front of you. One of those rarely mentioned but quite useful tools every sysadmin should know is script(1). It logs everything you type and everything that appears on the screen. You can record errors and log output for later dissection and analysis. For example, if you're running a program that fails in the same spot every time, you can use script to copy your keystrokes and the program's response. This is notably useful when upgrading your system or building software from source code; the last 30 lines or so of the log file make a nice addition to a help request.

To start script(1), just type **`script`**. You'll get your command prompt back and can continue working normally. When you want the recording to stop, just type `exit` or press CTRL-D. Your activity will appear in a file named *typescript*. If you want the file to have a particular name or be in a particular location, just give that name as an argument to `script`:

```
# script /home/mwlucas/debug.txt
```

This tool is extremely useful for recording exactly what you typed and exactly how the system responded. Any time you need to ask for help, consider script(1).

Repairing a Broken System

The best way to learn an operating system is to play with it, and the harder you play, the more you learn. If you play hard enough, you'll certainly break something, which is a good thing—having to fix a badly broken system is arguably the fastest way to learn. If you've just rendered your system unbootable or plan to learn quickly enough to risk doing that, this section is for you. If your system is deeply hosed, you'll learn a lot and quickly.

Single-user mode (discussed in Chapter 4) gives you access to many different commands and tools. What if you've destroyed those tools, however? Perhaps you've even damaged the statically linked programs in */rescue*. That's where the install media comes in.

The FreeBSD installation images have an option to activate a live system. This live system includes all the programs that come by default with FreeBSD. When you boot off the install media, you can choose to enter the live CD instead of installing.

You must have some familiarity with system administration to use the live CD. Essentially, the live CD gives you a command prompt and a variety of Unix utilities. You get to use the boot-time error messages and that ballast you keep between your ears to fix the problem. It's you against the computer. Of the first half-dozen times I've resorted to a live CD or its predecessors, the computer won three. After that, though, my success rate was much improved. Reading this book, as well as other Unix administration manuals, will improve your odds of success.

It's impossible to outline a step-by-step process for generic problem situations; the exact steps you must follow depend on the exact damage you've inflicted on your poor, innocent operating system. If you're really desperate, however, the live CD gives you a shot at recovery without reinstalling. I've had problems where I've accidentally destroyed my */etc* directory or fried the getty(1) program that displays a login prompt. Careful

use of the live CD can repair these problems in a fraction of the time a reinstall would take. If nothing else, perhaps you can back up any data that survived being exposed to you and then reinstall.

Always use install media that's roughly equivalent to the FreeBSD version you're running. You can probably use a 12.2 install CD to repair a 12.1 system, but a 14-current install disk might cause a whole new set of problems.

Now that you can recover from almost any mistake you might make, let's dive into the heart of FreeBSD: the kernel.

6
KERNEL GAMES

If you're new to Unix administration, the word *kernel* might intimidate you. After all, the kernel is one of those secret parts of a computer that mere mortals are not meant to dabble in. In some versions of Unix, kernel tampering is unthinkable. Microsoft doesn't advertise that its operating systems even have kernels, which is like glossing over the fact that human beings have brains.[1] While high-level users can access the kernel through a variety of methods, this isn't widely acknowledged or encouraged. In many parts of the open source Unix-like world, however, meddling with the kernel is a very viable and expected way to change system behavior. It would probably be an excellent way to adjust other operating systems, if you were allowed to do so.

The FreeBSD kernel can be dynamically tuned or changed on the fly, and most aspects of system performance can be adjusted as needed. We'll discuss the kernel's sysctl interface and how you can use it to alter a running kernel.

At the same time, some parts of the kernel can be altered only while the system is in the early stages of booting. The boot loader lets you adjust the kernel before the host even finds its filesystems.

Some kernel features require extensive reconfiguration. You can custom-build kernels for really tiny systems or build a kernel tuned precisely for the hardware you're running. The best way to do this is to build your own kernel.

FreeBSD has a modular kernel, meaning that entire chunks of the kernel can be loaded or unloaded from the operating system, turning entire subsystems on or off as desired. This is highly useful in this age of removable

hardware, such as PC cards and USB devices. Loadable kernel modules can impact performance, system behavior, and hardware support.

Finally, we'll cover basic debugging of your kernel, including some of the scary-looking messages it gives out as well as when and how to boot alternate kernels.

What Is the Kernel?

You'll hear many different definitions of a kernel. Many are just flat-out confusing, some are technically correct but bewilder the novice, while others are wrong. The following definition isn't complete, but it'll do for most people most of the time and it's comprehensible: *the kernel is the interface between the hardware and the software*.

The kernel lets the software write data to disk drives and to the network. When a program wants memory, the kernel handles all the low-level details of accessing the physical memory chip and allocating resources for the job. Once your MP3 file passes through the codec software, the kernel translates the codec output into a stream of zeros and ones that your particular sound card understands. When a program requests CPU time, the kernel schedules a time slot for it. In short, the kernel provides all the software interfaces that programs need in order to access hardware resources.

While the kernel's job is easy to define (at least in this simplistic manner), it's actually a complicated task. Different programs expect the kernel to provide different interfaces to the hardware, and different types of hardware provide interfaces differently. For example, FreeBSD supports a few dozen families of Ethernet cards, each with its own requirements that the kernel must handle. If the kernel can't talk to the network card, the system isn't on the network. Different programs request memory to be arranged in different ways, and if you have a program that requests memory in a manner the kernel doesn't support, you're out of luck. The way your kernel investigates some hardware during the boot sequence defines how the hardware behaves, so you have to control that. Some devices identify themselves in a friendly manner, while others lock up if you dare to ask them what they're for.

The kernel and any modules included with FreeBSD are files in the directory */boot/kernel*. Third-party kernel modules go in */boot/modules*. Files elsewhere in the system are not part of the kernel. Nonkernel files are

collectively called the *userland*, meaning they're intended for users even if they use kernel facilities.

Since a kernel is just a set of files, you can have alternative kernels on hand for special situations. On systems where you've built your own kernel, you will find */boot/kernel.old*, a directory containing the kernel that was installed before your current kernel. I habitually copy the kernel installed with the system into */boot/kernel.install*. You can also create your own special kernels. The FreeBSD team makes configuring and installing kernels as simple as possible. The simplest and best-supported way to alter a kernel is through the sysctl interface.

Kernel State: sysctl

The sysctl(8) program allows you to peek at the values used by the kernel and, in some cases, to set them. Just to make things more confusing, these values are also sometimes known as *sysctls*. The sysctl interface is a powerful feature because, in many cases, it will let you solve performance issues without rebuilding the kernel or reconfiguring an application. Unfortunately, this power also gives you the ability to sweep the legs out from under a running program and make your users really, really unhappy.

The sysctl(8) program handles all sysctl operations. Throughout this book, I'll point out how particular sysctls change system behavior, but first, you need to understand sysctls in general. Start by grabbing all the human-visible sysctls on your system and saving them to a file so you can study them easily.

```
# sysctl -o -a > sysctl.out
```

The file *sysctl.out* now contains hundreds of sysctl variables and their values, most of which will look utterly meaningless. A few of them, however, you can interpret without knowing much:

```
kern.hostname: storm
```

This particular sysctl, called `kern.hostname`, has the value `storm`. Oddly enough, the system I ran this command on has a hostname of *storm*, and the sysctl hints that this is the kernel's name for the system it's running on. See

these sysctls with the `-a` flag. Most sysctls are meant to be read this way, but a few, called *opaque sysctls*, can only be interpreted by userland programs. Show opaque sysctls with the `-o` flag.

```
net.local.stream.pcblist: Format:S,xunpcb Length:5488 Dump:0x20000000000000001
1000000dec0adde...
```

I could guess that the variable `net.local.stream.pcblist` represents something for the network stack. I can't even guess what the value means. Userland programs like netstat(1) pull information from these opaque sysctls.

sysctl MIBs

The sysctls are organized in a tree format called a *management information base (MIB)* with several broad categories, such as net (network), kern (kernel), and vm (virtual memory). Table 6-1 lists the roots of the sysctl MIB tree on a system running the GENERIC kernel.

Table 6-1: Roots of the sysctl MIB Tree

sysctl	Function
`kern`	Core kernel functions and features
`vm`	Virtual memory system
`vfs`	Filesystem
`net`	Networking
`debug`	Debugging
`hw`	Hardware
`machdep`	Machine-dependent settings
`user`	Userland interface information
`p1003_1b`	POSIX behavior
`kstat`	Kernel statistics
`dev`	Device-specific information
`security`	Security-specific kernel features

Each of these categories is divided further. For example, the `net` category,

covering all networking sysctls, is divided into categories such as IP, ICMP, TCP, and UDP. The concept of a management information base is used in several other parts of system administration, as we'll see in Chapter 21 and you'll see throughout your career. The terms *sysctl MIB* and *sysctl* are frequently used interchangeably. Each category is named by stringing together the parent category and all of its children to create a unique variable name, such as:

```
--snip--
kern.maxfilesperproc: 11095
kern.maxprocperuid: 5547
kern.ipc.maxsockbuf: 262144
kern.ipc.sockbuf_waste_factor: 8
kern.ipc.max_linkhdr: 16
--snip--
```

Here we have five sysctls plucked from the middle of the `kern` category. The first two are directly beneath the `kern` label and have no sensible grouping with other values other than the fact that they're kernel-related. The remaining three all begin with `kern.ipc`; they're part of the IPC (interprocess communication) section of kernel sysctls. If you keep reading the sysctls you saved, you'll see that some sysctl variables are several categories deep.

sysctl Values and Definitions

Each MIB has a value that represents a buffer, setting, or characteristic used by the kernel. Changing the value changes how the kernel operates. For example, the kernel handles transmitting and receiving packets, but by default won't send a packet from one interface to another. You can change a sysctl to permit this forwarding, thereby turning your host into a router.

Each sysctl value is either a string, an integer, a binary value, or an opaque. *Strings* are free-form texts of arbitrary length; *integers* are ordinary whole numbers; *binary* values are either 0 (off) or 1 (on); and *opaques* are pieces of machine code that only specialized programs can interpret.

Many sysctl values are not well documented; there is no single document listing all available sysctl MIBs and their functions. A MIB's documentation generally appears in a man page for the corresponding function, or sometimes only in the source code. For example, the original documentation

for the MIB `kern.securelevel` (discussed in Chapter 9) is in security(7). Although sysctl documentation has expanded in recent years, many MIBs still have no documentation.

Fortunately, some MIBs have obvious meanings. For example, as we discuss later in this chapter, this is an important MIB if you frequently boot different kernels:

```
kern.bootfile: /boot/kernel/kernel
```

If you're debugging a problem and have to reboot with several different kernels in succession, you can easily forget which kernel you've booted (not that this has ever happened to me, really). A reminder can therefore be helpful.

An easy way to get some idea of what a sysctl does is to use the `-d` switch with the full MIB. This prints a brief description of the sysctl:

```
# sysctl -d kern.maxfilesperproc
kern.maxfilesperproc: Maximum files allowed open per process
```

This brief definition tells you that this sysctl controls exactly what you might think it does. Unfortunately, not all sysctls provide definitions with `-d`. While this example is fairly easy, other MIBs might be much more difficult to guess.

Viewing sysctls

To view all the MIBs available in a particular subtree of the MIB tree, use the `sysctl` command with the name of the part of the tree you want to see. For example, to see everything under `kern`, enter this command:

```
# sysctl kern
kern.ostype: FreeBSD
kern.osrelease: 12.0-CURRENT
kern.osrevision: 199506
kern.version: FreeBSD 12.0-CURRENT #0 r322672: Fri Aug 18 16:31:34 EDT 2018
    root@storm:/usr/obj/usr/src/sys/GENERIC
--snip--
```

This list goes on for quite some time. If you're just becoming familiar with sysctls, you might use this to see what's available. To get the exact value of a specific sysctl, give the full MIB name as an argument:

```
# sysctl kern.securelevel
kern.securelevel: -1
```

The MIB `kern.securelevel` has the integer value `-1`. We'll discuss the meaning of this sysctl and its value in Chapter 9.

Changing sysctls

Some sysctls are read-only. For example, take a look at the hardware MIBs:

```
hw.model: Intel(R) Xeon(R) CPU E5-1620 v2 @ 3.70GHz
```

The FreeBSD Project has yet to develop the technology to change Intel hardware into ARM64 hardware via a software setting, so this sysctl is read-only. If you were able to change it, all you'd do is crash your system. FreeBSD protects you by not allowing you to change this value. An attempt to change it won't hurt anything, but you'll get a warning. On the other hand, consider the following MIB:

```
vfs.usermount: 0
```

This MIB determines whether users can mount removable media, such as CDROM and floppy drives, as discussed in Chapter 13. Changing this MIB requires no extensive tweaks within the kernel or modifications to hardware; it's only an in-kernel permissions setting. To change this value, use the `sysctl(8)` command, the sysctl MIB, an equal sign, and the desired value:

```
# sysctl vfs.usermount=1
vfs.usermount: 0 -> 1
```

The sysctl(8) program responds by showing the sysctl name, the old value, and the new value. This sysctl is now changed. A sysctl that can be tuned on the fly like this is called a *runtime tunable sysctl*.

Setting sysctls Automatically

Once you've tweaked your kernel's settings to your whim, you'll want those settings to remain after a reboot. Use the file */etc/sysctl.conf* for this. List each sysctl you want to set and the desired value in this file. For example, to set the `vfs.usermount` sysctl at boot, add the following on its own line in

/etc/sysctl.conf:

```
vfs.usermount=1
```

The Kernel Environment

The kernel is a program started by the boot loader. The boot loader can hand environment variables to the kernel, creating the *kernel environment*. The kernel environment is also a MIB tree, much like the sysctl tree. Many, but not all, of these environment variables later get mapped onto read-only sysctls.

Viewing the Kernel Environment

Use kenv(8) to view the kernel environment. Give it the name of a kernel environment variable to see just that variable, or run it without arguments to see the whole tree.

```
# kenv
LINES="24"
acpi.oem="SUPERM"
acpi.revision="2"
acpi.rsdp="0x000f04a0"
acpi.rsdt="0x7dff3028"
--snip--
```

These variables look an awful lot like the loader variables. Because they are the loader variables. They frequently relate to initial hardware probes. If your serial port uses an unusual memory address, the kernel needs to know about that before trying to probe it.

These environment settings are also called *boot-time tunable sysctls*, or *tunables*, frequently related to low-level hardware settings. As an example, when the kernel first probes a hard drive, it must decide whether it's going to provide ident-based or GPT ID-based labels. This decision must be made before anything in the kernel accesses the hard drive, and you can't change your mind without rebooting the machine.

Kernel environment variables can be set only from the loader. You can make changes manually at boot time or set them in */boot/loader.conf* to take effect at the next boot (see Chapter 4).

Much like *sysctl.conf*, setting tunable values in *loader.conf* will let you really mess up a machine. The good news is that these values are easily unset.

TOO MANY TUNABLES?

Don't become confused between sysctl values that can be set only at boot, sysctl values that can be tuned on the fly, and sysctls that can be set on the fly but have been configured to automatically adjust at boot. Remember that boot-time tunable sysctls involve low-level kernel functions, while runtime tunables involve higher-level functions. Having sysctls adjust themselves at boot is merely an example of saving your work—it doesn't change the category that the sysctl belongs to.

Dropping Hints to Device Drivers

You can use environment variables to tell device drivers needed settings. You'll learn about these settings by reading the driver man pages and other documentation. Additionally, much ancient hardware requires the kernel to address it at very specific IRQ and memory values. If you're old enough to remember plug-and-pray, "hardware configuration" floppy disks, and special slots for bus master cards, you know what I'm talking about and probably have one of these systems polluting your hardware closet even today. (If you're too young for that, buy one of us geezers a drink and listen to our horror stories.[2]) You can tell FreeBSD to probe for such hardware at any IRQ or memory address you specify, which is very useful when you have a card with a known configuration but the floppy that can change that configuration biodegraded years ago.

If you're truly unfortunate, you might have a machine with a built-in floppy disk drive. Look in */boot/device.hints* to find entries that configure this hardware:

```
hint.fdc.0.at="isa"
hint.❶fdc.❷0.❸port=❹"0x3F0"
```

```
hint.fdc.0.irq=❺"6"
hint.fdc.0.drq=❻"2"
```

These entries are all hints for the fdc(4) device driver ❶. The entry is used for fdc device number zero ❷. If you enable this device, a booting kernel will probe for a card at memory address (or port ❸) 0x3F0 ❹, IRQ 6 ❺, and DRQ 2 ❻. If it finds a device with these characteristics, it gets assigned the fdc(4) driver. If that device isn't a floppy drive, you'll have amusing crashes.[3]

TESTING BOOT-TIME TUNABLES

All of these hints and boot-time tunable sysctls are available in the boot loader and can be set interactively at the OK prompt, as discussed in Chapter 4. You can test settings without editing *loader.conf*, find the value that works, and only then make the change permanent in a file.

Boot-time tunables and sysctl let you adjust how a kernel behaves, but kernel modules let you add functionality to a running kernel.

Kernel Modules

Kernel modules are parts of a kernel that can be started, or loaded, when needed and unloaded when unused. Kernel modules can be loaded when you plug in a piece of hardware and removed with that hardware. This greatly expands the system's flexibility. Plus, a kernel with all possible functions compiled into it would be rather large. Using modules, you can have a smaller, more efficient kernel and load rarely used functionality only when it's required.

Just as the default kernel is held in the file */boot/kernel/kernel*, kernel modules are the other files under */boot/kernel*. Take a look in that directory to see hundreds of kernel module files. Each kernel module name ends in *.ko*. Generally speaking, the file is named after the functionality contained in the

module. For example, the file */boot/kernel/wlan.ko* handles the wlan(4) wireless layer. FreeBSD needs this module for wireless networking.

Viewing Loaded Modules

The kldstat(8) command shows modules loaded into the kernel.

```
   # kldstat
   Id Refs Address           Size     Name
❶ 1   36 0xffffffff80200000 204c3e0  kernel
❷ 2    1 0xffffffff8224e000 3c14f0   zfs.ko
❸ 3    2 0xffffffff82610000 d5f8     opensolaris.ko
❹ 5    1 0xffffffff82821000 ac15     linprocfs.ko
   --snip--
```

This desktop has three kernel modules loaded. The first is the kernel proper ❶; then, modules to support ZFS ❷ and the OpenSolaris kernel functions needed by ZFS ❸ follow. I experiment with Linux software on this host (see Chapter 17), so finding the linprocfs(5) module ❹ loaded is not a surprise.

Each module contains one or more submodules, which you can view using `kldstat -v`, but the kernel itself has a few hundred submodules—so be ready for a lot of output.

Loading and Unloading Modules

Loading and unloading kernel modules is done with kldload(8) and kldunload(8). For example, suppose I'm experimenting with IPMI on a test host. This requires the ipmi(4) kernel module. While I'd normally load this automatically at boot using *loader.conf*, I'm in the lab. I use the `kldload` command and the name of the kernel module or the file containing the kernel module for that feature:

```
# kldload /boot/kernel/ipmi.ko
```

If I happen to remember the name of the module, I can just use that. The module name doesn't need the *.ko* at the end of the file. I happen to recall the name of the IPMI module.

```
# kldload ipmi
```

Most often, my feeble brain relies on tab completion in my shell to remind me of the module's full and proper name.

Once I finish experimenting, I'll unload the module.[4] Specify the name of the kernel module as it appears in kldstat(8).

```
# kldunload ipmi
```

Any module that's actively in use, such as the *opensolaris.ko* module loaded whenever you use ZFS, will not be permitted to unload. Attempting to unload an active module gives you an error like this:

```
# kldunload opensolaris
kldunload: can't unload file: Device busy
```

Sysadmins load modules much more often than they unload them. Unloading modules is expected to work, and it works the overwhelming majority of the time, but it's arguably the most common way to panic a system. If unloading a module triggers a panic, file a bug report as per Chapter 24.

Loading Modules at Boot

Use */boot/loader.conf* to load modules at boot. The default *loader.conf* includes many examples of loading kernel modules, but the syntax is always the same. Take the name of the kernel module, chop off the trailing *.ko*, and add the string `_load="YES"`. For example, to load the module */boot/kernel/procfs.ko* automatically at boot, add this to *loader.conf*:

```
procfs_load="YES"
```

The hard part, of course, is knowing which module to load. The easy ones are device drivers; if you install a new network or SCSI card that your kernel doesn't support, you can load the driver module instead of reconfiguring the kernel. In this case, you'll need to find out which driver supports your card; the man pages and Google are your friends there. I'll be giving specific pointers to kernel modules to solve particular problems throughout this book.

Wait a minute, though—why would FreeBSD make you load a device driver to recognize hardware if it recognizes almost everything at boot? That's an excellent question! The answer is that you may have built your own custom kernel and removed support for hardware you're not using. You don't know how to build a kernel? Well, let's fix that right now.

Build Your Own Kernel

Eventually, you'll find that you can't tweak your kernel as much as you like using only sysctl(8) and modules, and your only solution will be to build a customized kernel. This sounds much harder than it is; we're not talking about writing code here—just editing a text file and running a couple of commands. If you follow the process, it's perfectly safe. If you *don't* follow the process, well, it's like driving on the wrong side of the road. (Downtown. During rush hour.) But the recovery from a bad kernel isn't that bad, either.

The kernel shipped in a default install is called *GENERIC*. GENERIC is configured to run on a wide variety of hardware, although not necessarily optimally. GENERIC boots nicely on most hardware from the last 15 years or so, and I frequently use it in production. When you customize your kernel, you can add support for specific hardware, remove support for hardware you don't need, or enable features not included in GENERIC.

DON'T REBUILD THE KERNEL

Once upon a time, building a kernel was considered a rite of passage. This is no longer the case. Most sysadmins need to rebuild a kernel only when they're playing with experimental features or specialty hardware.

Preparations

You must have the kernel source code before you can build a kernel. If you followed my advice back in Chapter 3, you're all set. If not, you can either go back into the installer and load the kernel sources, download the source code

from a FreeBSD mirror, or jump ahead to Chapter 18 and use svnlite(1). If you don't remember whether you installed the source code, look into your */usr/src* directory. If it contains a bunch of files and directories, you have the kernel sources.

Before building a new kernel, you must know what hardware your system has. This can be difficult to determine; the brand name on a component doesn't necessarily describe the device's identity or abilities. Many companies use rebranded generic components—I remember one manufacturer that released four different network cards under the same model name and didn't even put a version number on the first three. The only way to tell the difference was to keep trying different device drivers until one of them worked. This has been going on for decades—many different companies manufactured NE2000-compatible network cards. The outside of the box had a vendor's name on it, but the circuits on the card said *NE2000*. Fortunately, some vendors use a standard architecture for their drivers and hardware; you can be fairly sure that an Intel network card will be recognized by the Intel device driver.

The best place to see what hardware FreeBSD found on your system is the file */var/run/dmesg.boot*, discussed in Chapter 4. Each entry represents either a hardware or software feature in the kernel. As you work on a new kernel for a system, keep the *dmesg.boot* of that system handy.

Buses and Attachments

Every device in the computer is attached to some other device. If you read your *dmesg.boot* carefully, you can see these chains of attachments. Here's an edited set of boot messages to demonstrate:

```
❶ acpi0: <SUPERM SMCI--MB> on motherboard
❷ acpi0: Power Button (fixed)
❸ cpu0: <ACPI CPU> on acpi0
  cpu1: <ACPI CPU> on acpi0
❹ attimer0: <AT timer> port 0x40-0x43 irq 0 on acpi0
❺ pcib0: <ACPI Host-PCI bridge> port 0xcf8-0xcff on acpi0
❻ pci0: <ACPI PCI bus> on pcib0
```

Our first device on this system is acpi0 ❶. You might not know what that is, but you could always read `man acpi` to find out. (Or, if you must, you could

read the rest of this chapter.) There's a power button ❷ on the acpi0 device. The CPUs ❸ are also attached to acpi0, as is a timekeeping device ❹. Eventually we have the first PCI bridge, pcib0 ❺, attached to the acpi0 device. The first PCI bus ❻ is in turn attached to the PCI bridge.

So, your common PCI devices connect to a hierarchy of buses that, in turn, attach to a PCI bridge to talk to the rest of the computer. You could read *dmesg.boot* and draw a tree of all the devices on the system; while that isn't necessary, understanding what's attached where makes configuring a kernel much more likely to succeed.

If you're in doubt, use pciconf(8) to see what's actually on your system. `pciconf -lv` will list every PCI device attached to the system, whether or not the current kernel found a driver for it.

Back Up Your Working Kernel

A bad kernel can render your system unbootable, so you absolutely must keep a good kernel around at all times. The kernel install process keeps your previous kernel around for backup purposes, in the directory */boot/kernel.old*. This is nice for being able to fall back, but I recommend that you go further. See Chapter 4 for details on booting alternate kernels.

If you don't keep a known good backup, here's what can happen. If you build a new kernel, find that you made a minor mistake, and have to rebuild it again, the system-generated backup kernel is actually the first kernel you made—the one with that minor mistake. Your working kernel has been deleted. When you discover that your new custom kernel has the same problem, or an even more serious error, you'll deeply regret the loss of that working kernel.

A common place to keep a known good kernel is */boot/kernel.good*. Back up your working, reliable kernel like this:

```
# cp -a /boot/kernel /boot/kernel.good
```

If you're using ZFS, a boot environment might make more sense than copying (see Chapter 12).

Don't be afraid to keep a variety of kernels on hand. Disk space is cheaper than time. I know people who keep kernels in directories named by date so

that they can fall back to earlier versions if necessary. Many people also keep a current copy of the GENERIC kernel in */boot/kernel.GENERIC* for testing and debugging purposes. The only way to have too many kernels is to fill up your hard drive.

Configuration File Format

FreeBSD's kernel is configured via text files. There's no graphical utility or menu-driven system for kernel configuration; it's still much the same as in 4.4 BSD. If you're not comfortable with text configuration files, building a kernel is just not for you.

Each kernel configuration entry is on a single line. You'll see a label to indicate what sort of entry this is, and then a term for the entry. Many entries also have comments set off with a hash mark, much like this entry for the FreeBSD filesystem FFS:

```
options         FFS                     # Berkeley Fast Filesystem
```

Every complete kernel configuration file is made up of five types of entries: `cpu`, `ident`, `makeoptions`, `options`, and `devices`. The presence or absence of these entries dictates how the kernel supports the associated feature or hardware:

`cpu` This label indicates what kind of processor this kernel supports. The kernel configuration file for the boring old PC hardware includes several CPU entries to cover processors such as the 486 (I486_CPU), Pentium (I586_CPU), and Pentium Pro through modern Pentium 4 CPUs (I686_CPU). The kernel configuration for amd64/EM64T hardware includes only one CPU type, HAMMER, as that architecture has only one CPU family. While a kernel configuration can include multiple CPU types, they must be of similar architectures; a kernel can run on 486 and Pentium CPUs, but you can't have a single kernel run on both Intel-compatible and ARM processors.

`ident` Every kernel has a single `ident` line, giving a name for the kernel. That's how the GENERIC kernel gets its name; it's an arbitrary text string.

`makeoptions` This string gives instructions to the kernel-building software.

The most common option is `DEBUG=-g`, which tells the compiler to build a debugging kernel. Debugging kernels help developers troubleshoot system problems.

`options` These are kernel functions that don't require particular hardware. This includes filesystems, networking protocols, and in-kernel debuggers.

`devices` Also known as *device drivers*, these provide the kernel with instructions on how to speak to certain devices. If you want your system to support a piece of hardware, the kernel must include the device driver for that hardware. Some device entries, called *pseudodevices*, aren't tied to particular hardware, but instead support whole categories of hardware—such as Ethernet, random number generators, or memory disks. You might wonder what differentiates a pseudodevice from an option. The answer is that pseudodevices appear to the system as devices in at least some ways, while options have no device-like features. For example, the loopback pseudodevice is a network interface that connects to only the local machine. While no hardware exists for it, software can connect to the loopback interface and send network traffic to other software on the same machine.

Here's another snippet of a configuration file—the part that covers ATA controllers:

```
# ATA controllers
device          ahci     # AHCI-compatible SATA controllers
device          ata      # Legacy ATA/SATA controllers
device          mvs      # Marvell 88SX50XX/88SX60XX/88SX70XX/SoC SATA
device          siis     # SiliconImage SiI3124/SiI3132/SiI3531 SATA
```

Each of these devices is a different type of ATA controller. Compare these entries to a couple of our ATA entries in */var/run/dmesg.boot* :

```
atapci0: <Intel PIIX4 UDMA33 controller> port 0x1f0-0x1f7,0x3f6,0x170
-0x177,0x376,0xc160-0xc16f at device 1.1 on pci0
ata0: <ATA channel> at channel 0 on atapci0
ata1: <ATA channel> at channel 1 on atapci0
ada0 at ata0 bus 0 scbus0 target 0 lun 0
cd0 at ata1 bus 0 scbus1 target 0 lun 0
```

The kernel configuration has an ATA bus, `device ata`. It's a "legacy" ATA bus, whatever the word "legacy" means today. The dmesg snippet here starts

with the atapci device, the controller where ATA meets PCI. We then have two ATA buses, ata0 and ata1. Disk ada0 is on ata0, while CD drive cd0 is on ata1.

Without `device ata` in the kernel configuration, the kernel would not recognize the ATA bus. Even if the system figured out that the system has a DVD drive, the kernel wouldn't know the route to get information to and from it. Your kernel configuration must include all the intermediary devices for the drivers that rely on them. On the other hand, if your system doesn't have ATA RAID drives, floppy drives, or tape drives, you can remove those device drivers from your kernel.

If this host had an AHCI, MVS, or SIIS controller, those device names would show up in dmesg instead of ata.

Configuration Files

Fortunately, you don't normally create a kernel configuration file from scratch; instead, you build on an existing one. Start with the GENERIC kernel for your hardware architecture. It can be found in */sys/<arch>/conf* — for example, the i386 kernel configuration files are in */sys/i386/conf*, the amd64 kernel configuration files are in */sys/amd64/conf*, and so on. This directory contains several files, of which the most important are *DEFAULTS*, *GENERIC*, *GENERIC.hints*, *MINIMAL*, and *NOTES*:

DEFAULTS This is a list of options and devices that are enabled by default for a given architecture. That doesn't mean that you can compile and run *DEFAULTS*, but it is a starting point should you want to build a kernel by adding devices. Using *GENERIC* is easier, though.

GENERIC This is the configuration for the standard kernel. It contains all the settings needed to get standard hardware of that architecture up and running; this is the kernel configuration used by the installer.

GENERIC.hints This is the hints file that is later installed as */boot/device.hints*. This file provides configuration information for older hardware.

MINIMAL This configuration excludes anything that can be loaded from a module.

NOTES This is an all-inclusive kernel configuration for that hardware

platform. Every platform-specific feature is included in *NOTES*. Find platform-independent kernel features in */usr/src/sys/conf/NOTES*.

Many architectures also have architecture-specific configurations, needed only for that hardware. The i386 architecture includes the PAE kernel configuration, which lets you use more than 4GB of RAM on a 32-bit system. The arm architecture includes dozens of configurations, one for each of the many different platforms FreeBSD supports.

Sometimes, you'll find a kernel configuration that does exactly what you want. I want the smallest possible kernel. The *MINIMAL* kernel looks like a good place to start. Let's build it.

Building a Kernel

A base install of FreeBSD, combined with the operating system source code, includes all the infrastructure you need to easily build a kernel. All you need to do is tell the system which kernel configuration to build through the KERNCONF variable. You can set KERNCONF in */etc/src.conf* (or */etc/make.conf*, if you're really old-school).

```
KERNCONF=MINIMAL
```

If you're experimenting with building and running different kernels, though, it's best to set the configuration file on the command line when you build the kernel. Build the kernel with the `make buildkernel` command.

```
# cd /usr/src
# make KERNCONF=MINIMAL buildkernel
```

The build process first runs config(8) to find syntactical configuration errors. If config(8) detects a problem, it reports the error and stops. Some errors are blatantly obvious—for example, you might have accidentally deleted support for the Unix File System (UFS) but included support for booting off of UFS. One requires the other, and config(8) will tell you exactly what's wrong. Other messages are strange and obscure; those that may take the longest to figure out are like this:

```
MINIMAL: unknown option "NET6"
```

NET6 is the IPv6 option, isn't it? No, that's *I* NET6. Apparently some doofus examined the config file in a text editor and accidentally deleted a letter. The error is perfectly self-explanatory—once you're familiar with all the supported kernel options. Read these errors carefully!

Once config(8) validates the configuration, the kernel build process takes a few minutes on a modern machine. A successful build ends with a message like this.

```
--------------------------------------------------------------
>>> Kernel build for MINIMAL completed on Tue Sep 12 14:27:08 EDT 2017
--------------------------------------------------------------
```

After building the kernel, install it. Running `make installkernel` moves your current kernel to */boot/kernel.old* and installs the new kernel in */boot/kernel*. Installing a kernel is much faster than building it.

TRUSTING THE KERNEL

Eventually, you'll get to where you trust your kernel configuration and want to build and install it in a single command. The `make kernel` command builds and installs the kernel. Truly intense sysadmins run `make kernel && reboot`.

Once the install completes, reboot your server and watch the boot messages. If everything worked, you'll get something like the following, showing exactly what kernel is running and when it was built.

```
Copyright (c) 1992-2018 The FreeBSD Project.
Copyright (c) 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994
The Regents of the University of California. All rights reserved.
FreeBSD storm 12.0-CURRENT FreeBSD 12.0-CURRENT #0 r323136: Sat Sep  2
21:46:53 EDT 2018     root@storm:/usr/obj/usr/src/sys/MINIMAL  amd64
--snip--
```

The catch is, the MINIMAL kernel doesn't boot all hardware. It doesn't boot *most* hardware. And of the hardware MINIMAL can boot, it won't boot most FreeBSD installations on that hardware.

MINIMAL leaves everything that can be a module in a module. Disk

partitioning methods, both GPT and MBR, can be modules. You must load either *geom_part_gpt.ko* or *geom_part_mbr.ko* via *loader.conf* to boot MINIMAL. Filesystems are modules too, so you have to load those. In short, you have to load every stupid module required by the hardware and your installation decisions. MINIMAL is a good reference of what all kernels need, and a decent place to start designing your own kernel, but insufficient for production.

Booting an Alternate Kernel

So, what to do if your new kernel doesn't work, or if it works badly? Perhaps you forgot a device driver or accidentally cut out the `INET` option and can't access the internet. Sometimes it'll hang up so early in the boot process that the only thing you can do is reboot the host. Don't panic! You did keep your old kernel, right? Here's what to do.

Start by recording the error message. You'll need to research that message to find out how your new kernel failed you.[5] To fix the error, though, you'll need to boot a working kernel so you can build an improved kernel.

Back in Chapter 4, we discussed the mechanics of booting an alternate kernel. We'll go through the process of what to type here, but to see some of the in-depth details of loader management, you'll want to go back to the earlier section. For now, we'll focus on the reasons to boot an alternate kernel and on how to do it correctly.

Start by deciding which kernel you want to boot. Your old kernel should be in a directory under */boot*; in this section, we'll assume that you want to boot the kernel in */boot/kernel.good*. Reboot and interrupt the boot to get to the boot menu. The fifth option lets you choose a different kernel. The menu displays every kernel directory listed in the `kernels` option in *loader.conf*. While it lists `kernel` and `kernel.old` by default, I'll add `kernel.good`.

Once you install another new kernel, though, remember: the existing */boot/kernel* gets copied to */boot/kernel.old*, so your new kernel can be placed in */boot/kernel*. If that kernel doesn't boot, and your new kernel also doesn't boot, you'll be left without a working kernel. This kind of sucks. Be sure you keep a known good kernel on hand.

Custom Kernel Configuration

Maybe none of the provided kernel configurations are suitable for you. You need something different. FreeBSD lets you create whatever you want. It's easiest to modify an existing configuration, however. You can either copy an existing file or use `include` options. We'll start by modifying an existing file. Be sure you use the correct architecture directory, probably either */sys/amd64/conf* or */sys/i386/conf*.

Do not edit any of the files in the configuration directory directly. Instead, copy GENERIC to a file named after your machine or the kernel's function and then edit the copy. For this example, I'm building a minimal kernel to support VirtualBox systems. I copy the file *GENERIC* to a file called *VBOX* and open *VBOX* in my preferred text editor.

Trimming a Kernel

Once upon a time, memory was far more expensive than it is today and was available only in smaller quantities. When a system has 128MB of RAM, you want every bit of that to be available for work, not holding useless device drivers. Today, when a cheap laptop somehow suffers through the day with a paltry 64GB RAM, kernel size is almost irrelevant.

For most of us, stripping unnecessary drivers and features out of a kernel to shrink it is a waste of time and energy, but I would encourage you to do it once. It will teach you how to build a kernel so that when you have to test a kernel patch or something, you won't need to learn kernel building along with coping with the problem compelling the rebuild. It'll also help when you start experimenting with FreeBSD on tiny hosts like a BeagleBone or Raspberry Pi.

I want to build a kernel that supports VirtualBox kernels. I boot a working FreeBSD install on VirtualBox so I can get at *dmesg.boot*. I'll be going back and forth between the dmesg and the configuration, commenting out unneeded entries.

CPU Types

On most architectures, FreeBSD supports only one or two types of CPU. The amd64 platform supports only one, HAMMER. The i386 platform supports three, but two of those—the 486 and the original Pentium—are

wildly obsolete outside the embedded market.

```
cpu             I486_CPU
cpu             I586_CPU
cpu             I686_CPU
```

You need to include only the CPU you have. If you're not sure of the CPU in your hardware, check *dmesg.boot*. I have an ancient laptop that shows:

```
CPU: AMD Athlon(tm) 64 X2 Dual Core Processor 4200+ (2200.10-MHz 686-class CPU)
  Origin = "AuthenticAMD"  Id = 0x20fb1  Stepping = 1
  Features=0x178bfbff<FPU,VME,DE,PSE,TSC,MSR,PAE,MCE,CX8,APIC,SEP,MTRR,PGE,MCA,
CMOV,PAT,PSE36,CLFLUSH,MMX,FXSR,SSE,SSE2,HTT>
--snip--
```

As shown in bold, this is a 686-class CPU, which means that I can remove the I486_CPU and I586_CPU statements to make my kernel smaller.

Core Options

Following the CPU type configuration entries, we have a whole list of options for basic FreeBSD services, such as TCP/IP and filesystems. An average system won't require all of these, but having them present provides a great deal of flexibility. You'll also encounter options rarely used in your environment as well as those you can remove from your custom kernel configuration. We won't discuss all possible kernel options but will cover specific examples of different option types. I'll specifically mention those that can be trimmed from an internet server. The LINT file, man pages, and your favorite internet search engine can fill you in on the other options. If you're in doubt about an option, keep it. Or disable it and see what breaks.

Consider the following network-related options:

```
options         INET                # InterNETworking
options         INET6               # IPv6 communications protocols
options         IPSEC               # IP (v4/v6) security
options         IPSEC_SUPPORT       # Allow kldload of ipsec and tcpmd5
options         TCP_OFFLOAD         # TCP offload
options         TCP_HHOOK           # hhook(9) framework for TCP
options         SCTP                # Stream Control Transmission Protocol
```

These options support networking. INET is the standard old-fashioned TCP/IP, while INET6 supports IPv6. Much Unix-like software depends on

TCP/IP, so you certainly require both of these. IPSEC and IPSEC_SUPPORT let you use the IPSec VPN protocol. I certainly won't use these on my virtual machines, so I'll comment them out.

The TCP_OFFLOAD option lets the network stack offload TCP/IP computations to the network card. That sounds good, except the vnet(4) network interfaces on virtual machines don't perform that function. Off with its head!

The TCP_HHOOK option gives you a convenient man page to read. Would I use this option? Maaaybe. More importantly, I don't know what software I'm running will need it. I'll keep it.

The SCTP transport protocol is nifty, but totally useless to the virtual machines running on my laptop. Bye-bye.

```
options         FFS             # Berkeley Fast Filesystem
options         SOFTUPDATES     # Enable FFS soft updates support
options         UFS_ACL         # Support for access control lists
options         UFS_DIRHASH     # Improve performance on big directories
options         UFS_GJOURNAL    # Enable gjournal-based UFS journaling
```

The FFS option provides the standard FreeBSD filesystem, UFS. Even a ZFS host needs UFS support. Keep it. The other options are all related to FFS. We discuss FFS and its options in more detail than you care for in Chapter 11, but for right now, just trust me and go with it.

Soft updates ensure disk integrity even when the system shuts down incorrectly. As discussed in acl(9), UFS access control lists allow you to grant very detailed permissions on files, which I won't need on my virtual host. Whack!

UFS_DIRHASH enables directory hashing, making directories with thousands of files more efficient. Keep that. And I'm going to use soft updates journaling, not gjournaling, so UFS_GJOURNAL can go away.

```
options         MD_ROOT                 # MD is a potential root device
```

This option—and all other _ROOT options—lets the system use something other than a standard UFS or ZFS filesystem as a disk device for the root partition. The installer uses a memory device (MD) as a root partition. If you're using a diskless system (see Chapter 23), you'll need an NFS root partition. If you're running FreeBSD on a standard computer system, with a hard drive and a keyboard and whatnot, your kernel doesn't

need any of these features.

```
options         NFSCL                   # Network Filesystem Client
options         NFSD                    # Network Filesystem Server
options         NFSLOCKD                # Network Lock Manager
```

These two options support the Network File System (see Chapter 13). The vital question here is, do you need NFS? If so, do you need to be a server or a client? I'll include these.

```
options         MSDOSFS             # MSDOS filesystem
options         CD9660              # ISO 9660 filesystem
options         PROCFS              # Process filesystem (requires PSEUDOFS)
options         PSEUDOFS            # Pseudo-filesystem framework
```

These options support intermittently used filesystems, such as FAT, CDs, the process filesystem, and the pseudo-filesystem framework. We discuss many of these filesystems in Chapter 13, but they're all available as kernel modules. Kill them.

```
options         COMPAT_FREEBSD32        # Compatible with i386 binaries
options         COMPAT_FREEBSD4         # Compatible with FreeBSD4
options         COMPAT_FREEBSD5         # Compatible with FreeBSD5
options         COMPAT_FREEBSD6         # Compatible with FreeBSD6
--snip--
```

These compatibility options let your system run software built for older versions of FreeBSD or software that makes assumptions about the kernel that were valid for older versions of FreeBSD but are no longer true. If you're installing a system from scratch, you probably won't need compatibility with FreeBSD 4, 5, or 6, but a surprising amount of software requires compatibility with 32-bit FreeBSD. Keep the `COMPAT_FREEBSD32` option, or your system *will* break.

```
options         SCSI_DELAY=5000         # Delay (in ms) before probing SCSI
```

The `SCSI_DELAY` option specifies the number of milliseconds FreeBSD waits after finding your SCSI controllers before probing them, giving them a chance to spin up and identify themselves to the SCSI bus. If you have no SCSI hardware, you can remove this line.

```
options         SYSVSHM                 # SYSV-style shared memory
options         SYSVMSG                 # SYSV-style message queues
options         SYSVSEM                 # SYSV-style semaphores
```

These options enable System-V-style shared memory and interprocess communication. Many database programs use this feature.

Multiple Processors

The following entries enable symmetric multiprocessing (SMP) in i386 kernels:

```
options         SMP                     # Symmetric MultiProcessor Kernel
options         DEVICE_NUMA             # I/O Device Affinity
options         EARLY_AP_STARTUP
```

These probably don't hurt, but if you know you're running on a board with a single core, possibly a system that's very old or using embedded hardware, you can remove them.

Device Drivers

After all the options, you'll find device driver entries, which are grouped in fairly sensible ways. To shrink your kernel, you'll want to get rid of everything that your host isn't using—but what, exactly, is your host not using? Search for each device driver in *dmesg.boot*.

The first device entries are buses, such as `device pci` and `device acpi`. Keep these, unless you truly don't have that sort of bus in your system.

Next, we reach what most people consider device drivers proper—entries for floppy drives, SCSI controllers, RAID controllers, and so on. If your goal is to reduce the size of your kernel, this is a good place to trim heavily; remove all device drivers for hardware your computer doesn't have. You'll also find a section of device drivers for such mundane things as keyboards, video cards, USB ports, and so on. You almost certainly don't want to delete these.

The network card device driver section is quite long and looks much like the SCSI and IDE sections. If you're not going to replace your network card any time soon, you can eliminate drivers for any network cards you aren't using.

We won't list all the device drivers here, as there's very little to be learned from such a list other than the hardware FreeBSD supported at the time I wrote this section. Check the release notes for the version of FreeBSD

you're running to see what hardware it supports.

You'll also find a big section of drivers for virtualization. The most commonly used virtual interfaces are based on VirtIO, but you'll also see specific drivers for Xen, Hyper-V, and VMware. A kernel needs only the drivers for the virtualization platform it's run on. Kernels for real hardware don't need any of them, even if the host will have virtual machines running on it.

Pseudodevices

You'll find a selection of pseudodevices near the bottom of the GENERIC kernel configuration. As the name suggests, these are created entirely out of software. Here are some of the more commonly used pseudodevices.

```
device          loop            # Network loopback
```

The loopback device allows the system to communicate with itself via network sockets and network protocols. We'll discuss network connections in some detail in the next chapter. You might be surprised at just how many programs use the loopback device, so don't remove it.

```
device          random                  # Entropy device
device          padlock_rng             # VIA Padlock RNG
device          rdrand_rng              # Intel Bull Mountain RNGdevice
```

These devices provide pseudorandom numbers, required for cryptography operations and such mission-critical applications as games. Some of them require support in the underlying chipset. FreeBSD supports a variety of randomness sources, transparently aggregating them all into the random devices */dev/random* and */dev/urandom*.

```
device          ether           # Ethernet support
```

Ethernet has many device-like characteristics, and it's simplest for FreeBSD to treat it as a device. Leave this, unless you're looking for a learning opportunity.

```
device          vlan                    # 802.1Q VLAN support
device          tun                     # Packet tunnel
device          gif                     # IPv6 and IPv4 tunneling
```

These devices support networking features like VLANs and different sorts of tunnels.

```
device          md              # Memory "disks"
```

Memory disks allow you to store files in memory. This is useful for very fast, temporary data storage, as we'll learn in Chapter 13. For most (but not all) internet servers, memory disks are a waste of RAM. You can also use memory disks to mount and access disk images. If you're not using memory disks, you can remove them from your kernel.

Removable Hardware

The GENERIC kernel supports a few different sorts of removable hardware. If you have a laptop built in a year containing two consecutive nines or zeros, it might have Cardbus or even PCMCIA cards. Otherwise, you don't need that support in your kernel. FreeBSD supports hot-pluggable PCI cards, but if you don't have them? Throw those drivers out.

Including the Configuration File

Your kernel binary might be separated from the machine it's built on. I recommend using the INCLUDE_CONFIG_FILE option to copy the kernel configuration into the compiled kernel. You'll lose any comments, but at least you'll have the options and devices in this kernel and can duplicate it if needed. The sysctl `kern.conftxt` contains the kernel.

Once you have your trimmed kernel, try to build it. Your first kernel configuration will invariably go wrong.

Troubleshooting Kernel Builds

If your kernel build fails, the first troubleshooting step is to look at the last lines of the output. Some of these errors are quite cryptic, but others will be self-explanatory. The important thing to remember is that errors that say, "Stop in *some directory*" aren't useful; the useful error will be before these. We talked about how to solve these problems in "Asking for Help" on page 11: take the error message and toddle off to the search engine. Compile errors usually result from a configuration error.

Fortunately, FreeBSD insists upon compiling a complete kernel before

installing anything. A busted build won't damage your installed system. It will, however, give you an opportunity to test those troubleshooting skills we talked about way back in Chapter 1.

The most common sort of error is when the `make buildkernel` stage fails. It might look something like this:

```
--snip--
linking kernel.full
vesa.o: In function `vesa_unload':
/usr/src/sys/dev/fb/vesa.c:1952: undefined reference to ❶ `vesa_unload_ioctl'
vesa.o: In function `vesa_configure':
/usr/src/sys/dev/fb/vesa.c:1169: undefined reference to ❷ `vesa_load_ioctl'
*** Error code 1
--snip--
```

You'll see a few pages of Error code 1 messages, but the actual error appears before them.

Some line in our kernel requires the functions `vesa_unload_ioctl` ❶ and `vesa_load_ioctl` ❷, but the device or option that provides that function isn't in the kernel. Try an internet search for the errors. See whether there's a man page for those functions. If all else fails, search the source code.

```
# cd /usr/src/sys
# grep -R vesa_unload_ioctl *
dev/fb/vesa.h:int vesa_unload_ioctl(void);
dev/fb/vesa.c:  if ((error = vesa_unload_ioctl()) == 0) {
dev/syscons/scvesactl.c:vesa_unload_ioctl(void)
```

Wait—wasn't there a reference to a "syscons" driver in the GENERIC config file?

```
# syscons is the default console driver, resembling an SCO console
#device         sc
#options        SC_PIXEL_MODE          # add support for the raster text mode
```

I had commented out the sc(4) driver. Add it back in and try again.

There are more "proper" ways of figuring out what kernel devices require what devices. They all boil down to "read and comprehend the source code." Trial, error, research, and more trial and error turn out to be quicker for most of us.

Inclusions, Exclusions, and Expanding the Kernel

Now that you can build a kernel, let's get a little fancy and see how to use inclusions, the various *no* configurations, and the *NOTES* file.

NOTES

FreeBSD's kernel includes all sorts of features that aren't included in GENERIC. Many of these special features are intended for very specific systems or for weird corner cases of a special network. You can find a complete list of hardware-specific features in the file *NOTES* under each platform's kernel configuration directory—for example, */sys/amd64/conf/NOTES*. Hardware-independent kernel features—those that work on every platform FreeBSD supports—can be found in */sys/conf/NOTES*. If you have hardware that doesn't appear to be completely supported in the GENERIC kernel, take a look at *NOTES*. Some of these features are obscure, but if you have the hardware, you'll appreciate them. Let's take a look at a typical entry from *NOTES*:

```
# Direct Rendering modules for 3D acceleration.
device          drm             # DRM core module required by DRM drivers
device          mach64drm       # ATI Rage Pro, Rage Mobility P/M, Rage XL
device          mgadrm          # AGP Matrox G200, G400, G450, G550
device          r128drm         # ATI Rage 128
device          savagedrm       # S3 Savage3D, Savage4
device          sisdrm          # SiS 300/305, 540, 630
device          tdfxdrm         # 3dfx Voodoo 3/4/5 and Banshee
device          viadrm          # VIA
options         DRM_DEBUG       # Include debug printfs (slow)
```

Are you using any of these video cards on your desktop? Maybe you want a custom kernel that includes the appropriate device driver.

If the *NOTES* file lists all the features for every possible device, why not just use it as the basis for your kernel? First, such a kernel would use up far more memory than the GENERIC kernel. While even small modern machines have enough memory to run GENERIC without trouble, if the kernel becomes ten times larger without the corresponding increase in functionality, people would get annoyed. Also, many options are mutually exclusive. You'll find options that let you dictate how the kernel schedules processes, for example. The kernel can use only one scheduler at a time, and each scheduler runs its tendrils throughout the kernel. Adding all of them to

the kernel simultaneously would increase code complexity and decrease stability.

I make it a point to review *NOTES* every release or two, just to look for interesting new features.

Inclusions and Exclusions

FreeBSD's kernel configuration has two interesting abilities that can make maintaining a kernel easier: the `no` options and the `include` feature.

The `include` feature lets you pull a separate file into the kernel configuration. For example, if you have a kernel configuration that can be described as "GENERIC with a couple extra tidbits," you could include the GENERIC kernel configuration with an `include` statement:

```
include GENERIC
```

So, if you want to build a kernel that has all the functionality of GENERIC but also supports the DRM features of the VIA 3d chips, you could create a valid kernel configuration composed entirely of the following:

```
ident       VIADRM
include     GENERIC
options     drm
options     viadrm
```

You might think that this is actually more work than copying GENERIC to a new file and editing it, and you'd be correct. Why would you bother with this, then? The biggest reason is that as you upgrade FreeBSD, the GENERIC configuration can change. The GENERIC in FreeBSD 12.1 is slightly different from that in 12.0. Your new configuration is valid for both releases and in both cases can be legitimately described as "GENERIC plus my options."

This works well for including items but isn't very good for removing things from the kernel. Rather than manually recreating your kernel for every new FreeBSD version, you can use an `include` statement but exclude unneeded entries with the `nodevice` and `nooptions` keywords. Remove unwanted device drivers with `nodevice`, while `nooptions` disables unwanted options.

Take a look at the GENERIC-NODEBUG kernel configuration on a -current machine. It's the same as the GENERIC configuration, but it has all

of the debugging features disabled.

```
include GENERIC

ident   GENERIC-NODEBUG

nooptions       INVARIANTS
nooptions       INVARIANT_SUPPORT
nooptions       WITNESS
nooptions       WITNESS_SKIPSPIN
nooptions       BUF_TRACKING
nooptions       DEADLKRES
nooptions       FULL_BUF_TRACKING
```

We start by including the GENERIC kernel configuration. This kernel identifies itself as GENERIC-NODEBUG, though. The following seven `nooptions` statements turn off FreeBSD-current's standard debugging options. Developers use the GENERIC-NODEBUG kernel to see whether the kernel debugger is causing problems. If a kernel with debugging panics while a kernel without debugging does not panic, the debugging code suddenly looks suspiciously dubious.

Skipping Modules

If you've gone to the trouble of building a custom kernel, you probably know exactly which kernel modules your host needs. Why build all these dozens of kernel modules if you're never going to use them? You can turn off the building of modules with the `MODULES_OVERRIDE` option. Set `MODULES_OVERRIDE` to the list of modules you want to build and install.

```
# make MODULES_OVERRIDE='' kernel
```

Perhaps you want to build most of the modules, but you have reason to loathe a specific module. Exclude it from the build with `WITHOUT_MODULES`. Here, I exclude vmm from the build, because I don't want even the temptation of running bhyve(8) on VirtualBox. It's only a small step from there to running a dozen layers of virtualization and wondering why my laptop is slow.

```
# make KERNCONF=VBOX WITHOUT_MODULES=vmm kernel
```

Selectively building modules, combined with custom kernels, lets you lock yourself into really itty-bitty boxes. You'll only understand how itty-

bitty those boxes are when you find you're missing a feature you never thought you'd need. If you must build a kernel, be generous in what you keep.

Now that your local machine is tuned precisely the way you want it, let's consider the rest of the internet.

7
THE NETWORK

FreeBSD is famous for its network performance. The TCP/IP network protocol suite was first developed on BSD, and BSD, in turn, included the first major implementation of TCP/IP. While competing network protocols were considered more exciting in the 1980s, the wide availability, flexibility, and liberal licensing of the BSD TCP/IP stack made it the de facto standard. This isn't just a historical curiosity; today, Facebook is actively looking for engineers who can make Linux's network performance match that of FreeBSD. The project is expected to take several years.

Many system administrators today have a vague familiarity with the basics of networking but don't really understand how it all hangs together. Good sysadmins understand the network, however. Knowing what an IP address really is, how a netmask works, and how a port number differs from a protocol number is a necessary step toward mastering your profession. We'll cover some of these issues in this chapter. For a start, you must understand the network layers.

While this chapter gives a decent overview of TCP/IP, it won't cover many of the numerous details, gotchas, and caveats. If you need to learn more about TCP/IP, pick up a book on the subject. For an overview, check out my book *Networking for Systems Administrators* (Tilted Windmill Press, 2015). Eventually you'll need a deep dive into networking; proceed directly to Charles M. Kozierok's *The TCP/IP Guide* (No Starch Press, 2005).

The dominant internet protocol is TCP/IP (Transmission Control Protocol over Internet Protocol). TCP is a transport protocol, while IP is a network protocol, but they're so tightly intertwined that they're generally

referred to as a single entity. We'll look at how the network works, then discuss IP versions 4 and 6, and proceed to TCP and UDP.

Network Layers

Each layer of the network handles a specific task within the network process and interacts only with the layers above and below it. People learning TCP/IP often laugh when they hear that all these layers simplify the network process, but this is really true. The important thing to remember right now is that each layer communicates only with the layer directly above it and the layer directly beneath it.

The classic Open System Interconnection (OSI) network protocol stack has seven layers, is exhaustively complete, and covers almost any situation with any network protocol and any application. The internet, however, is just one such situation, and this isn't a book about networking or networked applications in general. We're limiting our discussion to TCP/IP networks, such as the internet and almost all corporate networks, so we need to consider only four layers of the network stack.

The Physical Layer

At the very bottom, we have the physical layer: the network card and the wire, fiber, or radio waves leaving it. This layer includes the physical switch; the hub, or base station; the cables attaching that device to the router; and the fiber that runs from your office to the telephone company. The telephone company switch is part of the physical layer, as are transcontinental fiber-optic cables. If it can be tripped over, dropped, or chainsawed, it's part of the physical layer. From this point on, we'll refer to the physical layer as the *wire*, although this layer can be just about any sort of medium.

This is the easiest layer to understand—it's as simple as having intact hardware. If your wire meets the requirements of the physical protocol, you're in business. If not, you're bankrupt. Without a physical layer, the rest of the network can't work—period. End of story. One of the functions of internet routers is to connect one sort of physical layer to another—for example, converting local Ethernet into optical fiber. The physical layer has

no decision-making abilities and no intelligence; everything that runs over it is dictated by the datalink layer.

Datalink: The Physical Protocol

The datalink layer, or the physical protocol, is where things get interesting. This layer transforms information into the actual ones and zeros that are sent over the physical layer in the appropriate encoding for that physical protocol. For example, IP version 4 (IPv4) over Ethernet uses Media Access Control (MAC) addresses and the Address Resolution Protocol (ARP); IP version 6 (IPv6) over Ethernet uses Neighbor Discovery Protocol (NDP or sometimes ND). In addition to the popular Ethernet datalink layers, FreeBSD supports others, including Point-to-Point Protocol (PPP) and High-Level Data Link Control (HDLC), as well as combinations such as the PPP over Ethernet (PPPoE) used by some home broadband vendors. While FreeBSD supports all of these datalink protocols, it doesn't support *every* datalink protocol ever used. If you have unusual network requirements, check the documentation for your version of FreeBSD to see whether it's supported.

Some physical protocols have been implemented over many different physical layers. Ethernet, for instance, has been transmitted over twinax, coax, CAT3, CAT5, CAT6, CAT7, optical fiber, HDMI, and radio waves. With minor changes in the device drivers, the datalink layer can address any sort of physical layer. This is one of the ways in which layers simplify the network. We'll discuss Ethernet in detail in "Understanding Ethernet" on page 140 at the end of this chapter, as it's the most common network type FreeBSD systems use. By understanding Ethernet on FreeBSD, you'll be able to manage other protocols on FreeBSD as well—once you understand those protocols, of course!

In addition to exchanging information with the physical layer, the datalink layer communicates with the network layer.

The Network Layer

The network layer? Isn't the whole thing a network?

Yes, but the network layer is more specific. It maps connectivity between network nodes, answering questions like, "Where are other hosts?" and

"Can you reach this particular host?" This logical protocol provides a consistent interface to programs that run over the network, no matter what sort of physical layer you're using. The network layer used on the internet is *Internet Protocol (IP)*. IP provides each host with a unique[1] address, known as an *IP address*, so that any other host on the network can find it. You need to understand IP, both version 4 and version 6.

The network layer is where we truly abstract away the underlying physical media. Is IP running over Ethernet? ATM? Carrier pigeon? Who cares? It's got an IP address, so we can talk to it. Move on.

The network layer talks to the datalink layer below it and the transport layer above it.

Heavy Lifting: The Transport Layer

The transport layer deals with real data for real applications and perhaps even real human beings. The three common transport layer protocols are ICMP, TCP, and UDP.

Internet Control Message Protocol (ICMP) manages basic connectivity messages between hosts with IP addresses. If IP provides a road and addresses, ICMP provides traffic lights and highway exit signs. Most of the time, ICMP just runs in the background and you never have to think about it.

The other well-known transport protocols are *User Datagram Protocol (UDP)* and *Transmission Control Protocol (TCP)*. How common are these?

Well, the Internet Protocol suite is generally called TCP/IP. These protocols provide services such as multiplexing via port numbers and transmitting user data. UDP is a bare-bones transport protocol, offering the minimum services needed to transfer data over the network. TCP provides more sophisticated features, such as congestion control and integrity checking.

In addition to these three, many other protocols run above IP. The file */etc/protocols* contains a fairly comprehensive list of transport protocols that use IP as an underlying mechanism. You won't find non-IP protocols here, such as Digital's LAT, but it contains many more protocols than you'll ever see in the real world. For example, here are the entries for IP and ICMP, the network-layer protocols commonly used on the internet:

```
❶ip   ❷0   ❸IP    ❹# Internet protocol, pseudo protocol number
  icmp  1    ICMP    # Internet control message protocol
```

Each entry in */etc/protocols* has three key fields: an unofficial name ❶, a protocol number ❷, and any aliases ❸. The protocol number is used within network requests to identify traffic. You'll see it if you ever fire up a packet sniffer or start digging deeper into your network for any reason. As you can see, IP is protocol 0 and ICMP is protocol 1—if that's not the groundwork for everything else, it's hard to see what could be! TCP is protocol 6, and UDP is protocol 17. You'll also see comments ❹ giving slightly more detail about each protocol.

The transport layer speaks to the network layer below and to the applications above it.

Applications

Applications are definitely a part of the network. Applications open requests for network connectivity, send data over the network, receive data from the network, and process that data. Web browsers, email clients, JSP servers, and so on are all network-aware applications. Applications have to communicate only with the network protocol and the user. Problems with the user layer are beyond the scope of this book.[2]

The Network in Practice

So, you understand how everything hooks together and are ready to move on, right? Don't think so. Let's see how this works in the real world. Some of this explanation touches on stuff that we'll cover later in this chapter, but if you're reading this book, you're probably conversant enough with networks to be able to follow it. If you're having trouble, reread this section after reading the remainder of this chapter. (Just buy a second copy of this book, cut these pages out of the second copy, and glue them in at the end of this chapter.)

Suppose a user connected to the internet via your network wants to look at Yahoo! The user accesses his web browser and enters the URL.

The browser application knows how to talk to the next layer down in the

network, which is the transport layer. After kneading the user's request into an appropriate form, the browser asks the transport layer for a TCP connection to a particular IP address on port 80. (Purists will note that we're skipping the DNS request part of the process, but it's quite similar to what's being described and would only confuse our example.)

The transport layer examines the browser's request. Since the application has requested a TCP connection, the transport layer allocates the appropriate system resources for that sort of connection. The request is broken up into digestible chunks and handed down to the network layer.

The network layer doesn't care about the actual request. It's been handed a lump of data to be carried over the internet. Much like your mail carrier delivers letters without caring about the contents, the network layer just bundles the TCP data with the proper addressing information. The resulting mass of data is called a *packet*. The network layer hands these packets down to the datalink layer.

The datalink layer doesn't care about the contents of the packet. It certainly doesn't care about IP addressing or routing. It's been given a lump of zeros and ones, and it has the job of transmitting those zeros and ones across the network. All it knows is how to perform that transmission. The datalink layer may add the appropriate header and/or footer information to the packet for the physical medium used, creating a *frame*. Finally, it hands the frame off to the physical layer for transmission on the local wire, wave, or other media.

EACH INSIDE THE OTHER?

Yes, your original web request has been encapsulated by the TCP protocol. That request has been encapsulated again at the transport layer by the IP protocol and once more by the datalink protocol. All these headers are piled on at the front and back of your original request. Have you ever seen that picture of a small fish being swallowed by a slightly larger fish, which is in turn being eaten by a larger fish, and so on? It's exactly like that. Or, if you prefer, a frame is like the outermost matryoshka doll. Unwrap one protocol and you'll

find another.

The physical layer has no intelligence at all. The datalink layer hands it a bunch of zeros and ones, and the physical layer transmits them to another physical device. It has no idea what protocol is being spoken or how those digits might be echoed through a switch, hub, or repeater, but one of the hosts on this network is presumably the router of the network.

When the router receives the zeros and ones, it hands them up to the datalink layer. The datalink layer strips its framing information and hands the resulting packet up to the network layer within the router. The router's network layer examines the packet and decides what to do with it based on its routing tables. It then hands the packet down to the appropriate datalink layer. This might be another Ethernet interface or perhaps a PPP interface out of a T1.

Your wire can go through many physical changes as the data travels. Your cable internet line could be aggregated into an optical fiber DS3, which is then transformed into an OC192 cross-country link. Thanks to the wonders of layering and abstraction, neither your computer nor your user needs to know anything about any of these.

When the request reaches its destination, the computer at the other end of the transaction accepts the frame and sends it all the way back up the protocol stack. The physical wire accepts the zeros and ones and sends them up to the datalink layer. The datalink layer strips the Ethernet headers off the frame and hands the resulting packet up to the network. The network layer strips off the packet header and shuffles the remaining segments up to the transport layer. The transport layer reassembles the segments into a data stream, which it then hands to an application—in this case, a web server. The application processes the request and returns an answer, which descends the protocol stack and travels across the network, bouncing up and down through various datalink layers on the way as necessary. This is an awful lot of work to make the machine go through just so you can get your "404 Page Not Found" error.

This example shows why layering is so important. Each layer knows only what it absolutely must about the layers above and below it, making it possible to swap out the innards of layers if desired. When a new datalink

protocol is created, the other layers don't have to change; the network protocol just hands a properly formatted request to the datalink layer and lets that layer do its thing. When you have a new network card, you only need a driver that interfaces with the datalink layer and the physical layer; you don't have to change anything higher in the network stack, including your application. Imagine a device driver that had to be installed in your web browser, your email client, and every other application you had on your computer, including the custom-built ones. You would quickly give up on computing and take up something sane and sensible, like skydiving with anvils.

Getting Bits and Hexes

As a system administrator, you'll frequently come across terms like *48-bit address* and *18-bit netmask*. I've seen a surprising number of sysadmins who just nod and smile when they hear this, all the while thinking, "Yeah, whatever, just tell me what I need to know to do my job." Unfortunately, math is a real part of the job, and you *must* understand bits. While this math is not immediately intuitive, understanding it is one of the things that separates amateurs from professionals. You don't read a book like this if you want to stay an amateur.

Maybe you're muttering, "But I already know this!" Then skip it. But don't cheat yourself if you don't.

You probably already know that a computer treats all data as zeros and ones, and that a single zero or one is a *bit*. When a protocol specifies a number of bits, it's talking about the number as seen by the computer. A 32-bit number has 32 digits, each being either zero or one. You were probably introduced to *binary* math, or *base 2*, back in elementary school and remembered it just long enough to pass the test. It's time to dust off those memories. Binary math is simply a different way to work with the numbers we see every day.

We use *decimal* math, or *base 10*, every day to pay the pizza guy and balance the checkbook. Digits run from 0 to 9. When you want to go above the highest digit you have, you add a digit on the left and set your current digit to zero. This is the whole "carry the one" thing you learned many years ago and now probably do without conscious thought. In binary math, the

digits run from 0 to 1, and when you want to go above the highest digit you have, you add a digit on the left and set your current digit to 0. It's exactly the same as decimal math with eight fingers missing. As an example, Table 7-1 shows the first few decimal numbers converted to binary.

Table 7-1: Decimal and Binary Numbers

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

When you have a 32-bit number, such as an IP address, you have a string of 32 ones and zeros. Ethernet MAC addresses are 48-bit numbers and have 48 ones and zeros.

Just for fun, Unix also uses hexadecimal numbers in some cases, such as MAC addresses and netmasks. Hexadecimal digits are 4 bits long. The binary number 1111, the full 4 bits, is equivalent to 15; this means that the digits in hexadecimal math run from 0 to 15. At this point, a few of you are looking at the 2-digit number 15 that's supposed to be a single digit and wondering what I'm smoking and where you can get your own supply. Hexadecimal math uses the letters A through F as digits for the numbers 10 through 15. When you count up to the last digit and want to add one, you set the current digit to zero and add a digit to the left of the number. For example, to count to 17 in hexadecimal, you say, "1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11." Take off a shoe and count along once or twice until you get the idea.

Hexadecimal numbers are usually marked with a *0x* in front. The number

0x12 is the hexadecimal equivalent of decimal 18, while the number 18 is plain old 18. If a hex number is not marked by a leading 0x, it's in a place where the output is always in hexadecimal, such as MAC addresses. The letters A to F are also a dead giveaway, but not entirely reliable; many hex numbers have no letters at all, just as many decimal numbers have no odd digits.

When you're working with hexadecimal, decimal, and binary numbers, the simplest thing to do is break out a scientific calculator. Today's medium-end or better calculators have functions to convert between the three systems, as do most software calculators.

BIT BY BYTES

Computer systems tend to work in bytes, where an 8-bit number is represented by a single character. The one exception is in the network stack, where everything is in bits. Thus, we have a 5-mega*byte* file on a machine with a 10-mega*bit* network connection. Don't confuse the two!

Network Stacks

A network stack is the software that lets a host communicate with other hosts over the network. A host can run with an IPv4-only network stack, an IPv6-only network stack, or a dual-stacked setup. FreeBSD enables both by default.

You're probably familiar with an IPv4-only stack. Most hosts have run on IPv4 for much of the past 30 years. An IPv4-only stack can communicate only over IPv4. Today, an IPv4-only stack gets you access to most of the internet, with a few deliberate exceptions. That will not be true in a few years.

Likewise, an IPv6-only stack can communicate only with IPv6 hosts. The majority of large internet sites support IPv6, but you'll find a few annoying exceptions.[3] Using only IPv6 will cut you off from some popular internet sites.

The most common server configuration these days is a dual-stack setup. Client hosts try to use both IPv4 and IPv6, preferring one over the other. The last few versions of Microsoft Windows have preferred IPv6.

We'll look at the more familiar IPv4 first and then use IPv4 as a reference to discuss IPv6.

IPv4 Addresses and Netmasks

An IP address is a unique 32-bit number assigned to a particular node on a network. Some IP addresses are more or less permanent, such as those assigned to vital servers. Others change as required by the network, such as those used by dial-up clients. Individual machines on a shared network get adjoining IP addresses.

Rather than expressing that 32-bit number as a single number, an IP address is broken up into four 8-bit numbers, usually shown as decimal numbers. While 203.0.113.1 is the same as 11001011.00000000.01110001.00000001 or the single number 11001011000000000111000100000001, the four decimal numbers are easiest for our weak minds to deal with.

IP addresses are issued in chunks by internet service providers. Frequently these chunks are very small—say, 16 or 32 IP addresses. If your system is colocated on a server farm, you might only get a few IP addresses out of a block.

A *netmask*, which might also be called a *prefix length* or *slash*, is a label indicating the size of the block of IP addresses assigned to your local network. The size of your IP block determines your netmask—or, your netmask determines how many IP addresses you have. If you've done networking for any length of time, you've seen the netmask 255.255.255.0 and know that it's associated with a block of 256 IP addresses. You might even know that the wrong netmask prevents your system from working. In today's world, however, that simple netmask is becoming less and less common. Netmasks made up of 255s and 0s are easy to look at but waste IP addresses.[4] And IPv4 addresses are an extremely scarce resource.

When you get a block of IP addresses for your server, it'll probably look something like 203.0.113.128/25. This isn't a class in binary math, so I won't make you draw it out and do the conversion, but think of an IP address as a

string of binary numbers. On your network, you can change the bits on the far right, but not the ones on the far left. The only question is, "Where is the line that separates right from left?" There's no reason for that boundary to be on one of those convenient 8-bit lines that separate the decimal versions of the address. A prefix length is simply the number of fixed bits on your network. A /25 means that you have 25 fixed bits. You can play with 7 bits. You get a decimal netmask by setting the fixed bits to 1 and your network bits to 0, as in the following example of a /25 netmask:

```
11111111.11111111.11111111.10000000
```

A binary 11111111 is a decimal 255, while 1000000 is 128. Your netmask is 255.255.255.128. It's very simple, if you think in binary. You won't have to work with this every day, but if you don't understand the underlying binary concepts, the decimal conversion looks deranged. With practice, you'll learn to recognize certain strings of decimals as legitimate binary conversions.

What does all this mean in practice? First off, blocks of IP addresses are issued in multiples of 2. If you have 4 bits to play with, you have 16 IP addresses ($2 \times 2 \times 2 \times 2 = 16$). If you have 8 bits to play with, you have 256 addresses ($2^8 = 256$). If someone says that you have exactly 19 IP addresses, you're either sharing an Ethernet with other people or they're wrong.

It's not uncommon to see a host's IP address with its netmask attached—for example, 198.51.100.4/26. This gives you everything you need to get the host on the local network. (Finding the default gateway is another problem, but by convention, it's most often the top or bottom address in the block.)

Computing Netmasks in Decimal

You probably don't want to repeatedly convert between decimal and binary. Not only is it uncomfortable; it also increases your chances of making an error. Here's a trick to calculate your netmask while remaining in decimal land.

You need to find how many IP addresses you have on your network. This will be a multiple of 2 almost certainly smaller than 256. Subtract the number of IP addresses you have from 256. This is the last number of your netmask. You'll still need to recognize legitimate network sizes. If your IP address is 203.0.113.100/26, you'll need to know that a /26 is 26 fixed bits, or

64 IP addresses. Look at the last number of your IP address, 100. It certainly isn't between 0 and 63, but it's between 64 and 127. The other hosts on your IP block have IP addresses ranging from 203.0.113.64 to 203.0.113.127, and your netmask is 255.255.255.192 (256 – 64 = 192).

At this point, I should mention that netmasks frequently appear as hex numbers. You might feel like giving up the whole thing as a bad job, but to simplify your life, Table 7-2 shows netmasks, IP information, and related goodness for /24 and smaller networks.

Table 7-2: Netmask and IP Address Conversions

Prefix	Binary mask	Decimal mask	Hex mask	Available IPs
/24	00000000	255.255.255.0	0xffffff00	256
/25	10000000	255.255.255.128	0xffffff80	128
/26	11000000	255.255.255.192	0xffffffc0	64
/27	11100000	255.255.255.224	0xffffffe0	32
/28	11110000	255.255.255.240	0xfffffff0	16
/29	11111000	255.255.255.248	0xfffffff8	8
/30	11111100	255.255.255.252	0xfffffffc	4
/31	11111110	255.255.255.254	0xfffffffe	2
/32	11111111	255.255.255.255	0xffffffff	1

Unusable IP Addresses

You now understand how slashes, netmasks, and IP address assignments work together and how, for example, a /28 has 16 IP addresses. Unfortunately, you can't use all the IP addresses in a block. The first IP address in a block is the *network number*, which is used for internal bookkeeping.

Traditionally, the last number in any block of IP addresses was called the *broadcast address*. According to the original IP specifications, every machine on a network was supposed to respond to a request for this address. This allowed you to ping the broadcast address to quickly determine which IP addresses were in use. For example, on a typical /24 network, the broadcast

address was *x .y .z* .255. In the late 1990s, however, this feature was transformed into an attack technique and was disabled by default on almost every operating system and most network appliances.[5] If you need this feature, set the sysctl `net.inet.icmp.bmcastecho` to `1`. In most environments, the broadcast address is a waste of an IP address. In any case, you can't assign the first or the last IP address in a network to a device without causing network problems. (Yes, this makes /31 networks useless.) Some systems fail gracefully; others fail gracelessly. Go ahead and try it sometime—preferably after hours, unless you want a good story to tell at your next job.

Assigning IPv4 Addresses

You might think that each computer on a network has an IP address, but this isn't strictly true. Every network *interface* has an IP address. Most computers have only one network interface, so for them, the difference is nonexistent. If you have multiple network cards, however, each card has a separate IP address. You can also have multiple IP addresses on a single card through aliasing. On the other hand, with special configuration you can bond multiple cards into a single network interface, giving the computer one virtual interface despite the many cards. While these distinctions are small, remember them when troubleshooting.

The IP address 127.0.0.1 is always attached to every host's loopback interface. It can be reached only from the local machine.

IPv6 Addresses and Subnets

The original engineers of IPv4 thought that 4.29 billion IP addresses would be enough for the whole world. Computers were expensive, after all, and only military and educational systems connected to the internet. It's not as if every person in the world would one day own multiple networked devices.

Oops.

Unused IPv4 addresses are no longer available. The prices for used IPv4 addresses are increasing. Eventually, IPv4 addresses will be priced beyond reach for most people. The world is unwillingly groaning toward IPv4's replacement—IP version 6.

Telecom networks and parts of the world outside North America already

use IPv6 pretty widely. Even if your network doesn't use IPv6 today, one day you'll unexpectedly discover that you needed it the week before.

IPv6 Basics

Like IPv4, IPv6 is a network-layer protocol. TCP, UDP, ICMP, and other protocols run atop it. Recall that IPv4 uses 32-bit addresses, usually expressed as four groups of decimal numbers from 0 to 255—for example, 203.0.113.13. IPv6 uses 128-bit addresses, expressed as eight groups of four hexadecimal characters separated by colons—for example, 2001:db8:5c00:0:90ff:bad:c0de:cafe.

A 128-bit address space is unimaginably huge, but let's try to imagine it. Count every human being that's ever lived. Now count the number of cells in each of them—not just in their body but also all the bacterial cells in their bodies. IPv6 is roomy enough to assign each of those cells an address space larger than the entirety of IPv4.

The good news is that you don't need to relearn the basics of networking. Hosts need an IP address, a netmask, and a default gateway. You can almost —*almost*—substitute an IPv6 address for an IPv4 address and watch everything just work. A web server doesn't care whether it binds to port 80 on 203.0.113.13 or 2001:db8:5c00:0:90ff:bad:c0de:cafe. The server accepts requests it receives and responds appropriately. That said, software does need to change slightly because our web server must be able to log connections from both IPv4 and IPv6 addresses. These changes have wide-reaching repercussions, and we'll be discovering new edge cases for decades. But, in general, once you understand the new rules for IPv6, all of your networking knowledge is applicable.

Understanding IPv6 Addresses

As noted, IPv6 addresses are 128 bits, expressed as eight colon-delimited groups of four hexadecimal characters each. As with decimal IP addresses, you don't need to display leading zeros in each group. The address 2001:db8:5c00:0:90ff:bad:c0de:cafe could be written as 2001:0db8:5c00:0000:90ff:0bad:c0de:cafe, but just as we wouldn't write 203.000.113.013, we strip out the leading zeros in an IPv6 address.

IPv6 addresses often contain long strings of zeros because of the way IPv6

subnets. As I write this, the IPv6 address of Sprint's website is 2600:0:0:0:0:0:0:0. When consecutive groups contain only zeros, they're replaced with two colons. You can display this IP address as 2600::. You can do the double-colon substitution only once per address, however. Addresses like 2001::a::1 would be ambiguous. Does 2001::a::1 represent 2001:0:0:0:0:a:0:1, 2001:0:0:0:a:0:0:1, 2001:0:0: a:0:0:0:1, or 2001:0:a:0:0:0:0:0:1? No way to tell.

You've probably seen a port number added to an IPv4 address, such as 203.0.113.13:80. Using this terminology with IPv6 addresses would make them even uglier and confuse everyone. An IP address and port combination like 2001:db8:5c00:0:90ff:bad:c0de:cafe:80 is not ambiguous, but unless you read it very carefully, you might think it's an IP address ending in 80. If you're expressing an IP and port combination, enclose the address in square brackets, as in [2001:db8:5c00:0:90ff:bad:c0de:cafe]:80.

IPv6 Subnets

IPv6 addresses have colons every 16 bits, so the obvious and natural ways to divide networks are at the /16, /32, /48, /64, /80, /96, and /112. The original IPv6 standards recommend subnetting only on these boundaries (repeating one of IPv4's greatest mistakes), but that's increasingly being rejected in favor of IPv4-style subnetting anywhere. IPv6 subnets are always expressed as a slash, also known as a prefix length, so you won't see a netmask like ffff:ffff:ffff:ffff:: analogous to IPv4.

ISPs are usually issued a /32 or /48 and are expected to issue end-user networks, such as a typical client, a /64 network. A /64 has 2^{64} subnets, or 18,446,744,073,709,551,616 addresses. If your home or office runs out of IP addresses, you need to stop networking individual blueberries.

When you subnet at 16-bit boundaries, each network has 65,536 subnets of the next smaller size. A /32 contains 65,536 /48 networks, and a /48 contains 65,536 /64 networks.

This is a long-winded way of explaining why I don't provide handy charts of IPv6 subnets and network size. Do an internet search for "IPv6 subnet calculator" to use one of the many on the internet.

Link-Local Addresses

Addresses beginning with *fe8x:* (where *x* is any hexadecimal character) are local to their interface. Every interface has such *link-local* addresses that are valid only on a specific local network. Even if an IPv6 network has no router, hosts on the local directly attached network can find each other and communicate using these local addresses. Link-local networks are always /64 subnets. You'll see identical IPv6 subnets on other interfaces and on networks completely disconnected from your network. That's okay. These addresses are local to the link. For example, here's a link-local address from a test machine.

```
fe80::bad:c0de:cafe%vtnet0
```

The link-local address of this interface is `fe80::bad:c0de:cafe`. The trailing `%vtnet0` indicates that this address is local to the interface `vtnet0` and isn't usable on any other interface on the machine. If your machine has an interface vtnet1, and a host on that network tries to reach the address `fe80::bad:c0de:cafe`, this machine will not respond. This particular address is valid only for hosts on the network segment directly attached to interface `vtnet0`.

You might note that the link-local address has a section in common with the public IPv6 address on this interface. That's because an autoconfigured IPv6 address is usually calculated from the interface's physical address; it doesn't matter whether that autoconfigured address is public or local to the link.

Assigning IPv6 Addresses

IPv6 clients on a /64 or larger network can normally autoconfigure their network through *router discovery*. Router discovery resembles a stripped-down DHCP service. The router broadcasts gateway and subnet information, and the hosts configure themselves to use it.

Modern versions of router discovery include very basic DHCP-style options, such as DNS servers. Not all IPv6 providers include these options in their router discovery configuration, however. If you want to provide sophisticated autoconfiguration for phones or diskless hosts, or if your provider doesn't offer DNS information in their configuration, you'll need to set up an IPv6 DHCP server.

Servers should not use IPv6 autoconfiguration. A server usually needs a static IP, even in IPv6.

Hosts on a network smaller than /64 must be manually configured.

The address ::1 always represents the local host and is assigned to the loopback address.

TCP/IP Basics

Now that you have a simple overview of how the IP system works, let's consider the most common network protocols in more depth. The dominant transport protocol on the internet is the Transmission Control Protocol over Internet Protocol, or TCP/IP. Although TCP is a transport protocol and IP is a network protocol, the two are so tightly intertwined that they're generally referred to as a single entity.

We'll start with the simplest, ICMP, and proceed to UDP and TCP. All of these protocols run over both IPv4 and IPv6. While the versions of each protocol vary according to the underlying IP protocol, they behave essentially the same.

ICMP

The Internet Control Message Protocol (ICMP) is the standard for transmitting routing and availability messages across the network. Tools such as ping(8) and traceroute(8) use ICMP to gather their results. IPv4 and IPv6 have slightly different versions of ICMP, sometimes called *ICMPv4* and *ICMPv6*.

While some people claim that you must block ICMP for security reasons, ICMP is just as diverse as the better-understood protocols TCP and UDP. Proper IPv4 network performance requires large chunks of ICMPv4. If you feel you must block ICMP, do so selectively. For example, blocking source quench messages breaks path maximum transmission unit (pMTU) discovery, which is like faceplanting into a crate of broken glass and rusty nails. If you don't understand that last sentence, don't block ICMP.

IPv6 dies without ICMPv6, as IPv6 doesn't support packet fragmentation. If you use IPv6, never block ICMPv6 as a whole. Blocking parts of ICMPv6 without destroying your network requires careful research

and testing.

UDP

The User Datagram Protocol (UDP) is the most bare-bones data transfer protocol that runs over IP. It has no error handling, minimal integrity verification, and no defense whatsoever against data loss. Despite these drawbacks, UDP can be a good choice for particular sorts of data transfer, and many vital internet services rely on it.

When a host transmits data via UDP, the sender has no way of knowing whether the data ever reached its destination. Programs that receive UDP data simply listen to the network and accept what happens to arrive. When a program receives data via UDP, it cannot verify the source of that data—while a UDP packet includes a source address, this address is easily faked. This is why UDP is called *connectionless*, or *stateless*.

With all of these drawbacks, why use UDP at all? Applications that use UDP most often have their own error-correction handling methods that don't mesh well with the defaults provided by protocols such as TCP. For example, simple client DNS queries must time out within just a few seconds or the user will call the helpdesk and whine. TCP connections time out only after two minutes. Since the computer wants to handle its failed DNS requests much more quickly, simple DNS queries use UDP. In cases where DNS must transfer larger amounts of data (for example, for zone transfers), it intelligently switches to TCP. Real-time streaming data, such as video conferencing, also uses UDP. If you miss a few pixels of the picture in a real-time video conference, retransmitting that data would simply add congestion. You can't go back in time to fill in those missing chunks of the picture, after all! You'll find similar reasoning behind almost all other network applications that use UDP.

Because the UDP protocol itself doesn't return anything when you connect to a port, there's no reliable way to remotely test whether a UDP port is reachable (although tools like nmap try to do so).

UDP is also a *datagram* protocol, meaning that each network transmission is complete, self-contained, and received as a single integral unit. While the application might not consider a single UDP packet a complete request, the network does. TCP is entirely different.

TCP

The Transmission Control Protocol (TCP) includes such nifty features as error correction and recovery. The receiver must acknowledge every packet it gets; otherwise, the sender will retransmit any unacknowledged packets. Applications that use TCP can expect reliable data transmission. This makes TCP a *connected*, or *stateful*, protocol, unlike UDP.

TCP is also a *streaming* protocol, meaning that a single request can be split amongst several network packets. While the sender might transmit several chunks of data one after the other, the recipient could receive them out of order or fragmented. The recipient must keep track of these chunks and assemble them properly to complete the network transaction.

For two hosts to exchange TCP data, they must set up a channel for that data to flow across. One host requests a connection, the other host responds to the request, and then the first host starts transmitting. This setup process is known as the *three-way handshake*. The specifics are not important right now, but you should know that this process happens. Similarly, once transmission is complete, the systems must do a certain amount of work to tear down the connections.

TCP is commonly used by applications—such as email programs, FTP clients, and web browsers—for its fairly generic set of timeouts and transmission features.

How Protocols Fit Together

You can compare the network stack to sitting with your family at a holiday dinner. The datalink layer (ARP, in the case of IPv4 over Ethernet) lets you see everyone else at the table. IP gives every person at the table a unique chair, except for the three young nephews using piano bench NAT. ICMP provides basic routing information, such as, "The quickest way to the peas is to ask Uncle Chris to hand them to you." TCP is where you hand someone a dish and the other person must say, "Thanks," before you let go. Finally, UDP is like tossing a roll at Aunt Betty; she might catch it, it might bounce off her forehead, or it could be snatched out of midair by the dog who has watched for her opportunity since the meal began.

Transport Protocol Ports

Have you ever noticed that computers have too many ports? We're going to add TCP and UDP ports into the stew. *Transport protocol ports* permit one server to serve many different services over a single transport protocol, multiplexing connections between machines.

When a network server program starts, it attaches, or *binds*, to one or more logical ports. A logical port is just an arbitrary number ranging from 1 to 65535. For example, internet mail servers bind to TCP port 25. Each TCP or UDP packet arriving at a system has a field indicating its desired destination port. Each incoming request is flagged with a desired destination port number. If an incoming request asks for port 25, it's connected to the mail server listening on that port. This means that other programs can run on different ports, clients can talk to those different ports, and nobody except the sysadmin gets confused.

The */etc/services* file contains a list of port numbers and the services that they're commonly associated with. It's possible to run almost any service on any port, but by doing so, you'll confuse other internet hosts that try to connect to your system. If someone tries to send you email, their mail program automatically connects to port 25 on your system. If you run email on port 77 and you have a web server on port 25, you'll never get your email and your web server will start receiving spam. The */etc/services* file has a very simple five-column format.

```
❶qotd  ❷17/❸tcp  ❹quote        ❺#Quote of the Day
```

This is the entry for the `qotd` service ❶, which runs on port 17 ❷ in the TCP protocol ❸. It's also known as the `quote` service ❹. Finally, we have a comment ❺ that provides more detail; apparently *qotd* stands for *quote of the day*. Services are assigned the same port number in both TCP and UDP, even though they usually run only on one and not the other—for example, qotd has ports 17/tcp and 17/udp.

Many server programs read */etc/services* to learn which port to bind to on startup, while client programs read */etc/services* to learn which port they should try to connect to. If you run servers on unusual ports, you might have to edit this file to tell the server where to attach to.

As in all standards, there are often good reasons for breaking the rules. The SSH daemon, sshd, normally listens on port 22/tcp, but I've run it on ports 23 (telnet), 80 (HTTP), and 443 (HTTPS) for various reasons. Configuring this depends on the server program you're using.

Reserved Ports

Ports below 1024 in both TCP and UDP are called *reserved ports*. These ports are assigned only to core internet infrastructure and important services such as DNS, SSH, HTTP, LDAP, and so on—services that should legitimately be offered only by a system or network administrator. Only programs with root-level privileges can bind to low-numbered ports. A user can provide, say, a game server on a high-numbered port if the system policy allows—but that's a little different from setting up an official-looking web page that's visible to everyone and states that the main purpose of the machine is to be a game server! The port assignment for these core protocols is generally carved in stone.

You can view and change the reserved ports with the sysctls `net.inet.ip.portrange.reservedhigh` and `net.inet.ip.portrange.reservedlow`.

Every so often, someone thinks that they can disable this "bind-only-by-root" feature and increase their system's security—after all, if your application can be run as a regular user instead of root, wouldn't that increase system security? Most programs that run on reserved ports actually start as root, bind to the port, and then drop privileges to a special restricted user that has even less privilege than a regular user. These programs are designed to start as root and frequently behave differently when run as a regular user. A few programs, such as the Apache web server, are written so they can be started safely by a non-root user, but others are not.

Understanding Ethernet

Ethernet is extremely popular in corporate and home networks and is the most common connection media for FreeBSD systems. Ethernet is a shared network; many different machines can connect to the same Ethernet and can communicate directly with each other. This gives Ethernet a great advantage over other network protocols, but Ethernet has physical distance limitations that make it practical only for offices, co-location facilities, and other

comparatively small networks.[6]

Many different physical media have supported Ethernet over the years. Once upon a time, most Ethernet cables were thick chunks of coaxial cable. Today, most are comparatively thin CAT6 cables with eight strands of very thin wire inside them. You might also encounter Ethernet over optical fiber or radio. For purposes of our discussion, we'll assume that you're working with CAT6 or better cable, today's most popular choice. No matter what physical media you use, the theory of Ethernet doesn't change—remember, the physical layer is abstracted away.

Protocol and Hardware

Ethernet is a broadcast protocol, which means that every packet you send on the network can be sent to every workstation on the network. (Note that I said *can be*; some Ethernet hardware limits recipients of these broadcasts.) Either your network card or its device driver separates the data intended for your computer from the data meant for other computers. One side effect of Ethernet's broadcast nature is that you can eavesdrop on other computers' network traffic. While this can be very useful when diagnosing problems, it's also a security issue. Capturing clear-text passwords is trivial on an old-fashioned Ethernet. A section of Ethernet where all hosts can communicate directly with all other hosts without involving a router is called a *broadcast domain*, or *segment*.

Ethernet segments are connected via hubs or switches. An Ethernet *hub* is a central piece of hardware to physically connect many other Ethernet devices. Hubs simply forward all received Ethernet frames to every other device attached to the network. Hubs broadcast all Ethernet traffic that they receive to every attached host and other attached hubs. Each host is responsible for filtering out the traffic it doesn't want. Hubs are old-school Ethernet and rarely seen today.

Switches have largely supplanted hubs. A switch is like a hub, but it filters which traffic it sends to each host. It identifies the physical addresses of attached devices and, for the most part, forwards frames only to the devices they are meant for. Since each Ethernet host has a finite amount of bandwidth, switching reduces the load on individual systems by decreasing the amount of traffic each host must sort through.

Switch Failure

Switches fail, despite what Cisco would have you believe. Some failures are obvious, such as those where the magic black smoke is leaking out of the back of the box. When a switch loses its magic smoke, it stops working. Others are more subtle and make it appear that the switch is still working.

Every switch manufacturer must decide how to handle subtle errors. Either the switch can shut down until it is attended to, or it can attempt to alert its manager and continue forwarding packets to the best of its ability. If you're a vendor, the choice is obvious—you stumble along as best you can so your customers don't think your switches are crap. This means your switch can start to act like a hub and you might not know about it. The bad news is that if you were relying on the switch to prevent leakage of secure information, you're fated for disappointment. More than one switch has failed on me in this way, so don't be too surprised when it happens to you.

Installing a syslog server (see Chapter 21) and having your switches log to it can mitigate this risk. While logging won't prevent switch failure, it will simplify listening to a dying switch when it tries to complain.

Ethernet Speed

Ethernet originally supported only a couple of megabits per second but has expanded to handle tens-of-gigabits speeds. Most Ethernet cards are *gigabit* speed, meaning they can handle a gigabit per second, but you'll find a few 10Gbs or 100Gbs cards in high-speed applications. If a card is labeled *gigabit*, it doesn't mean it can actually push that much traffic—I've seen gigabit cards choke on a tenth that much bandwidth. Card quality is important when you want to push bandwidth, and the quality of the entire computer is important when pushing serious bandwidth.

Let the switch and the card negotiate their settings on their own through *autonegotiation*. While some old hands might remember disabling autonegotiation on older Ethernet cards, gigabit and faster Ethernet requires autonegotiation to function.

MAC Addresses

Every Ethernet card has a unique identifier, a Media Access Control (MAC) address. This 48-bit number is sometimes called an *Ethernet address* or

physical address. When a system transmits data to another host on the Ethernet, it first broadcasts an Ethernet request asking, "Which MAC address is responsible for this IP address?" If a host responds with its MAC address, further data for that IP is transmitted to that MAC address.

IPv4 uses the Address Resolution Protocol (ARP) to map IP addresses to hosts. Use arp(8) to view your FreeBSD system's knowledge of the ARP table. The most common usage is the `arp -a` command, which shows all of the MAC addresses and hostnames that your computer knows of.

```
# arp -a
gw.blackhelicopters.org (198.51.100.1) at 00:00:93:34:4e:78 on igb0 [ethernet]
sipura.blackhelicopters.org (198.51.100.5) at 00:00:93:c2:0f:8c on igb0 [ethernet]
```

This full listing of ARP information is known as the *ARP table*, or *MAC table*. (The terms MAC and ARP are frequently used interchangeably, so don't worry about it too much.) Here we see that the host *gw.blackhelicopters.org* has an IP address of 198.51.100.1 and a MAC address of 00:00:93:34:4e:78, and that you can reach these hosts on the local system's interface, `igb0`.

If a MAC address shows up as `incomplete`, the host cannot be contacted on the local Ethernet. In this case, check your physical layer (the wire), the remote system, and the configuration of both systems.

IPv6 uses Neighbor Discovery Protocol (NDP) to map IPv6 addresses to MAC addresses. It's a separate protocol from ARP to encompass router discovery. Use ndp(8) to view the host's MAC table and corresponding IPv6 addresses. The output deliberately resembles that of arp(1).

```
# ndp -a
Neighbor                          Linklayer Address  Netif  Expire    S Flags
fe80::fc25:90ff:fee8:1270%vtnet0  fe:25:90:e8:12:70  vtnet0 4s        R R
www.michaelwlucas.com             00:25:90:e8:12:70  vtnet0 permanent R
fe80::225:90ff:fee8:1270%vtnet0   00:25:90:e8:12:70  vtnet0 permanent R
```

The output deliberately resembles that of arp(8) but is slightly more tabular. The `Neighbor` column shows either the IPv6 address, the hostname, or the link-local address of each neighbor. The `Linklayer Address` column shows the MAC address of the neighbor. The `Netif` column displays the network interface this host is attached to, while the `Expire` column shows when the cached entry will expire. The `S` (state) column shows further information about the entry. A state of `R` means the host is reachable, while an `I`

(incomplete) means the host is unreachable. The only `Flags` entry you're likely to see is `R`, indicating this host is advertising itself as a router. For more states and flags, see ndp(8).

Why two separate commands? Both arp(8) and ndp(8) exist to map IP addresses to MAC addresses. Some hosts might be available only via one protocol or the other. IPv6-only hosts will not show up in your ARP table, and IPv4-only hosts will not appear in the NDP table.

For both arp(8) and ndp(8), the `-n` flag turns off hostname lookups. This is highly useful when you debug network issues and can't get DNS resolution.

Now that you know how the network works, configuring an internet connection is pretty straightforward.

8
CONFIGURING NETWORKING

Now that you know enough networking to be dangerous, you can configure a network connection. While FreeBSD supports many different protocols, we'll focus on the nearly ubiquitous Ethernet connection, generally delivered over CAT5 or CAT6 cables.[1]

We'll start with the essentials for getting a host on the network and able to access other internet hosts. Raw TCP/IP connectivity isn't enough, however; you also need the ability to resolve host names to IP addresses, so we'll cover that next. Then we'll talk about measuring network activity, performance, VLANs, and aggregating links.

Before you can do any of that, though, you need some information.

Network Prerequisites

If your network offers Dynamic Host Configuration Protocol (DHCP), you can connect to the network as a client without knowing anything about the network. A static IP address makes much more sense on a server, however. While the installer will configure the network for you, eventually every server needs changes. Both IPv4 and IPv6 require the following information:

- An IP address
- The netmask for that IP address and protocol
- The IP address of the default gateway

Armed with this information, attach your system to the network with ifconfig(8) and route(8) and then make the configuration permanent in */etc/rc.conf*.

Configuring Changes with ifconfig(8)

The ifconfig(8) program displays the interfaces on your computer and lets you configure them. Start by listing the existing interfaces on your system by running ifconfig(8) without any arguments:

```
# ifconfig
❶em0: flags=8843<❷UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
        options=85259b<RXCSUM,TXCSUM,VLAN_MTU,VLAN_HWTAGGING,VLAN_HWCSUM,TSO4,
LRO,WOL_MAGIC,VLAN_HWFILTER,VLAN_HWTSO>
        inet ❸203.0.113.43 netmask 0xfffffff0 broadcast 198.51.100.47
        inet6 ❹fe80::225:90ff:fee8:1270%em0 prefixlen 64 scopeid 0x1
        inet6 ❺2001:db8::bad:c0de:cafe prefixlen 64
        ether ❻00:25:90:db:d5:94
        media: ❼Ethernet autoselect (1000baseTX <full-duplex>)
        status: ❽active

rl0: flags=8802<BROADCAST,SIMPLEX,MULTICAST> mtu 1500
        options=8<VLAN_MTU>
        ether 00:20:ed:72:3b:5f
        media: Ethernet autoselect (10baseT/UTP)
        status: ❾no carrier
❿lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> metric 0 mtu 16384
        options=600003<RXCSUM,TXCSUM,RXCSUM_IPV6,TXCSUM_IPV6>
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x2
        inet 127.0.0.1 netmask 0xff000000
        nd6 options=21<PERFORMNUD,AUTO_LINKLOCAL>
```

Our first network interface is em0 ❶, or the first network card that uses the em(4) driver. The em(4) man page reveals that this is an Intel PRO/1000 card. You'll then see basic information about this card ❷, including that it is in the UP state, meaning it's either working or trying to work. It's assigned the IPv4 address 203.0.113.43 ❸ and the netmask 0xfffffff0 (or 255.255.255.240, per Table 7-2). This card has two IPv6 addresses, the link-local address (beginning with fe80) ❹ and the global IPv6 address ❺. You'll also see the MAC address ❻ and the connection speed ❼. Finally, the `status`

entry shows that this card is active ❽: a cable is plugged in and we have a link light.

The second card, rl0, has almost none of this information associated with it. One key fact is the `no carrier` signal ❾: it's not plugged in and there is no link light. This card is not in use.

Finally we have the interface lo0 ❿, the loopback. This interface has the IPv4 address 127.0.0.1 and IPv6 address ::1 on every machine. This loopback address is used when the machine talks to itself. This is a standard software interface, which does not have any associated physical hardware. Do not attempt to delete the loopback interface, and do not change its IP address—things will break in an amusing way if you do so. FreeBSD supports other software interfaces, such as disc(4), tap(4), gif(4), and many more.

Adding an IP to an Interface

The install process will configure any network cards you have working at install time. If you didn't configure the network for all of your cards during the setup process, or if you add or remove network cards after finishing the install, you can assign an IP address to your network card with ifconfig(8). You need the card's assigned IP address and netmask.

```
# ifconfig interface-name inet IP-address netmask
```

For example, if your network card is em0, your IP address is 203.0.113.250, and your netmask is 255.255.255.0, you would type:

```
# ifconfig em0 inet 203.0.113.250 255.255.255.0
```

Specify the netmask in dotted-quad notation as above or in hex format (0xffffff00). Perhaps simplest of all is to use slash notation, like this:

```
# ifconfig em0 inet 203.0.113.250/24
```

To configure an IPv6 address, add the `inet6` keyword between the interface name and the address.

```
# ifconfig em0 inet6 2001:db8::bad:c0de:cafe/64
```

The ifconfig(8) program can also perform any other configuration your

network cards require, letting you work around hardware bugs in features such as the various sorts of checksum offloading, like setting media type and duplex mode for sub-gigabit interfaces. You'll find supported options in the man pages for the driver and ifconfig(8). Here, I disable checksum offloading and TCP segmentation offloading on my em0 interface, even while I set the IP address.

```
# ifconfig em0 inet 203.0.113.250/24 -tso -rxcsum
```

To make this persist across reboots, add an entry to */etc/rc.conf* that tells the system to configure the card at boot. An IPv4 entry has the form `ifconfig_`*`interfacename`*`="`*`ifconfig arguments`*`"`. For example, configuring the idle rl0 card would require an entry much like this:

```
ifconfig_rl0 ="inet 203.0.113.250/24"
```

An IPv6 entry has the form `ifconfig_`*`interfacename`*`_ipv6="`*`ifconfig arguments`*`"`.

```
ifconfig_rl0_ipv6="2001:db8::bad:c0de:cafe/64"
```

Once you have a working configuration for your interface, copy your ifconfig(8) arguments into a */etc/rc.conf* entry.

Testing Your Interface

Now that your interface has an IP address, try to ping the IPv4 address of your default gateway. If you get a response, as shown in the following example, you're on the local network. Interrupt the ping with CTRL-C.

```
# ping 203.0.113.1
PING 203.0.113.1 (203.0.113.1): 56 data bytes
64 bytes from 203.0.113.1: icmp_seq=0 ttl=64 time=1.701 ms
64 bytes from 203.0.113.1: icmp_seq=1 ttl=64 time=1.436 ms
^C
--- 203.0.113.1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 1.436/1.569/1.701/0.133 ms
```

For IPv6, use ping6(8) instead of ping(8). If you use router discovery, the default route will almost always be a link-local address.

```
# ping6 2001:db8::1
PING6(56=40+8+8 bytes) 2001:db8::bad:c0de:cafe --> 2001:db8::1
```

```
16 bytes from 2001:db8::1, icmp_seq=0 hlim=64 time=0.191 ms
16 bytes from 2001:db8::1, icmp_seq=1 hlim=64 time=0.186 ms
16 bytes from 2001:db8::1, icmp_seq=2 hlim=64 time=0.197 ms
--snip--
```

If you don't get any answers, your network connection isn't working. Either you have a bad connection (check your cables and link lights) or you have misconfigured your card.

Set Default Route

The default route is the address where your system sends all traffic that's not on the local network. If you can ping the default route's IPv4 address, set it via route(8).

```
# route add default 203.0.113.1
```

That's it! You should now be able to ping any public IPv4 address on the internet.

Adding the default IPv6 route is much the same, but you need to add the `-6` command line flag to change the IPv6 routing table.

```
# route -6 add default 2001:db8::1
```

If you didn't choose nameservers during the system install, you'll have to use the IP address rather than the hostname.

Once you have a working default router, make it persist across reboots by adding the proper `defaultrouter` and `ipv6_defaultrouter` entries in */etc/rc.conf*:

```
defaultrouter="203.0.113.1"
ipv6_defaultrouter="2001:db8::1"
```

Multiple IP Addresses on One Interface

A FreeBSD system can respond to multiple IP addresses on one interface. This is especially useful for jails (see Chapter 22). Specify additional IPv4 addresses for an interface with ifconfig(8) and the keywords `inet` and `alias`. The netmask on an IPv4 alias is always /32, regardless of the size of the network address block the main address uses.

```
# ifconfig em0 inet alias 203.0.113.225/32
```

IPv6 aliases use the actual prefix length (slash) of the subnet they're on. Be sure you use the `inet6` keyword.

```
# ifconfig em0 inet6 alias 2001:db8::bad:c0de:caff/64
```

Once you add an alias to the interface, the additional IP address appears in ifconfig(8) output. The main IP always appears first, and aliases follow.

```
  # ifconfig fxp0
  fxp0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
          options=b<RXCSUM,TXCSUM,VLAN_MTU>
          inet6 fe80::225:90ff:fee8:1270%vtnet0 prefixlen 64 scopeid 0x1
❶         inet6 2001:db8::bad:c0de:cafe prefixlen 64
          inet6 2001:db8::bad:c0de:caff prefixlen 64
          inet 203.0.113.250 netmask 0xffffff00 broadcast 203.0.113.255
❷         inet 203.0.113.225 netmask 0xffffffff broadcast 203.0.113.255
          ether 00:02:b3:63:e4:1d
  --snip--
```

Here we see our brand new IPv4 ❶ and IPv6 ❷ aliases. Hosts that ping your aliased addresses will get a response from this server.

Once you have the aliases working as you like, make them persist across reboots by adding additional `ifconfig` statements in */etc/rc.conf*:

```
ifconfig_em0_alias0="inet 203.0.113.225/32"
ifconfig_em0_alias1="inet6 2001:db8::bad:c0de:caff/64"
```

The only real difference between this entry and the standard *rc.conf* 's "here's my IP address" entry are the `alias0` and `alias1` chunks. The `alias` keyword tells FreeBSD that this is an aliased IP, and the `0` and `1` are unique numbers assigned to each alias. Every alias set in */etc/rc.conf* must have a unique number, and this number must be sequential. If you skip a number, aliases after the gap won't be installed at boot. This is the most common interface misconfiguration I've seen.

Many daemons, such as inetd(8) and sshd(8), can be bound to a single address (see Chapter 20), so you can run multiple instances of the same program on the same server using multiple addresses.

ALIASES AND OUTGOING CONNECTIONS

All connections from your FreeBSD system use the system's real IP address. You might have 2,000 addresses bound to one network card, but when you `ssh` from that machine, the connection comes from the primary IP address. Keep this in mind when writing firewall rules and other access-control filters. Jails initiate all connections from the jail IP address, but we won't cover jails until Chapter 22.

Renaming Interfaces

FreeBSD names its network interfaces after the device driver used by the network card. This is a fine old tradition in the Unix world and common behavior among most industrial operating systems. Some operating systems name their network interfaces by the type of interface—for example, Linux calls its Ethernet interfaces *eth0*, *eth1*, and so on. At times, it makes sense to rename an interface, either to comply with an internal standard or to make its function more apparent. For example, I have one device with 12 network interfaces, each plugged into a different network. Each network has a name such as *test*, *QA*, and so on. Renaming these network interfaces to match the attached networks makes sense.

While FreeBSD is flexible on interface names, some software isn't—it assumes that a network interface name is a short word followed by a number. This isn't likely to change any time in the near future, so it's best practice to use a short interface name ending in a digit. Use ifconfig(8)'s `name` keyword to rename an interface. For example, to rename *em1* to *test1*, you would run:

```
# ifconfig em1 name test1
```

Running ifconfig(8) without arguments shows that you have renamed that interface.

```
--snip--
test1: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        options=b<RXCSUM,TXCSUM,VLAN_MTU>
```

```
--snip--
```

Make this change permanent with the `ifconfig_interface_name` option in */etc/rc.conf*.

```
ifconfig_em1_name="test1"
```

FreeBSD renames interfaces early in the boot process, before setting IP addresses or other values. This means that any further interface configuration must reference the new interface name rather than the old. Full configuration of a renamed interface with IP addresses and aliases would look something like this:

```
ifconfig_em1_name="dmz2"
ifconfig_dmz2="inet 203.0.113.2 netmask 255.255.255.0"
ifconfig_dmz2_alias0="inet 203.0.113.3"
```

DHCP

Very few networks use DHCP for everything, including servers. A DHCP server will set the server's IP address, netmask, nameservers, and default gateway for you. If your network administrator configures servers via DHCP, you can tell the network card to take its configuration via DHCP with the following:

```
ifconfig_em0="DHCP"
```

Reboot!

Now that you have your network interfaces fully configured, be sure to reboot to test any changes you made to */etc/rc.conf*. If FreeBSD finds an error in */etc/rc.conf*, especially in network configuration, you'll have problems accessing the system remotely. It's much better to learn that you made a typo under controlled conditions as opposed to the middle of your sleeping hours.

If you feel like living dangerously, you can run `service netif restart` with the interface name to reconfigure only a single interface.

```
# service netif restart em0
```

Skip the interface name, and this will restart all interfaces. It's not a perfect test, but it will catch a bunch of daftness. A reboot is always the best test.

The Domain Name Service

The *Domain Name Service*, or *DNS*, is one of those quiet, behind-the-scenes services that doesn't get half the attention it deserves. Although most users have never heard of it, DNS makes the internet as we know it work. Also called *name service*, DNS provides a map between hostnames and IP addresses. It also provides the reverse map, of IP addresses to hostnames. Without DNS, web browsers and email programs couldn't use the nice and convenient hostnames like *www.michaelwlucas.com* or *www.nostarch.com*; instead, you'd have to browse the web by typing in appalling things like *https://2001:19f0:5c00:9041:225:90ff:fee8:1270*. This would greatly reduce the internet's popularity.[2] To most end users, a DNS failure is an internet failure, end of story. While we won't discuss building your own authoritative nameserver, we must cover configuring your server to use DNS.

A host that trawls the internet to dig up DNS mappings is called a *nameserver*, or *DNS server*. DNS servers aren't difficult to run, but most individuals don't need one. DNS servers are needed only by organizations who run their own servers (and lunatics who have dozens of hosts in their basement, like me). Nameservers come in two varieties, authoritative and recursive.

Authoritative nameservers provide DNS mappings for the public to find an organization's nameservers. As the operator of *michaelwlucas.com*, I must provide authoritative nameservers for that domain and let the public query them. These authoritative nameservers answer queries only about the domains I manage. Configuring an authoritative nameserver is beyond the scope of this book.

Recursive nameservers service client requests. When you try to browse to *https://www.michaelwlucas.com*, your local recursive nameserver searches the internet for my authoritative nameserver. Once the recursive nameserver retrieves the hostname-to-IP mapping, it returns that response to your client. This book shows you how to use recursive nameservers and how to enable your own recursive nameserver.

The system's *resolver* is responsible for configuring how the host performs DNS queries and relaying the responses to programs. Configuring the resolver is a vital part of system administration. Even DNS servers need a configured resolver, because the host won't know it's a nameserver unless you tell it so. Configuring a resolver requires answering a few questions:

- Where does the server look for DNS information?
- What local overrides do you want?
- What are the local domain names?
- Which nameservers should be queried?

The answers to these questions are configured in */etc/nsswitch.conf* and */etc/resolv.conf*.

Host/IP Information Sources

This should be easy. A server gets its host information from a nameserver, right? I just spent a few paragraphs telling you that, didn't I?

The real world isn't quite that simple, though. Perhaps you have a small home network with only three machines. You want each machine to be able to find each other by hostname, but you don't want to run a local authoritative nameserver. Or maybe you're on a large corporate network where completing DNS changes takes weeks, and you have a couple test systems that need to talk to each other. FreeBSD, like all Unix-like operating systems, can get information from both DNS and from the plaintext hosts file */etc/hosts*.

When FreeBSD needs to know the address of a host (or the hostname of an address), by default the query goes first to the hosts file and then the configured nameservers. This means that you can locally override nameserver results, which is very useful for hosts behind a NAT or on large corporate networks with odd requirements. In some cases, you might need to reverse this order to query DNS first and the hosts file second. Set this order in */etc/nsswitch.conf*.

NAME SERVICE SWITCHING

The file */etc/nsswitch.conf* is used not only by the resolver, but also by all other name services. A networked operating system includes many different name services. The TCP/IP ports in */etc/services* are a name service as well as network protocol names and numbers. Determining a user's UID and GID requires a different sort of name lookup (see Chapter 9). */etc/nsswitch.conf* determines ordering for all of these queries and more. We're discussing only hostname lookups here, but Chapter 20 covers more on name service switching.

Each entry in */etc/nsswitch.conf* is a single line containing the name of the name service, a colon, and a list of information sources. Here's the hostname service lookup configuration:

```
hosts: files dns
```

The resolver queries the information sources in the order listed. If you have an additional information source, such as nscd(8), list it here. The documentation sources for these add-ons should include the name of the service.

Local Names with /etc/hosts

The */etc/hosts* file matches internet addresses to hostnames. Once upon a time, before the Domain Name Service, the internet had a single hosts file that provided the hostnames and IP addresses of every node on the internet. Sysadmins submitted their host changes to a central maintainer, who issued a revised hosts file every few months. Sysadmins would then download the hosts file and install it on all of their machines. This worked fine when the whole internet had four systems on it, and was even acceptable when there were hundreds of hosts. As soon as the internet began its exponential growth, however, this scheme became totally unmaintainable.

While the hosts file is very effective, it works only on the machine it's installed on and must be maintained by the sysadmin. The public DNS has largely supplanted */etc/hosts*, but it's still useful in environments where you

don't want to run local authoritative DNS[3] or you're behind an IPv4 NAT device. Using the hosts file makes perfect sense if you have one or two servers at home, or if someone else manages your authoritative nameservers. Once you have enough hosts that the thought of updating the hosts file makes you ill, it's time to learn to build an authoritative nameserver.

Each line in */etc/hosts* represents one host. The first entry on each line is an IP address, and the second is the fully qualified domain name of the host, such as *mail.michaelwlucas.com*. Following these two entries, you can list an arbitrary number of aliases for that host.

For example, a small company might have a single server handling email, serving FTP, web pages, and DNS, as well as performing a variety of other functions. A desktop on that network might have a hosts file entry like this:

```
203.0.113.3     mail.mycompany.com    mail ftp www dns
```

With this */etc/hosts* entry, the desktop could find the server with either the full domain name or any of the brief aliases listed. This won't get you to Facebook, however. For that, you need nameservice.

Configuring Nameservice

Tell your host how to query nameservers with the file */etc/resolv.conf*. You probably want to provide a local domain or a domain search list and then list the nameservers.

Local Domain and Search List

If your organization has many machines, typing out complete hostnames can quickly get old. If you're doing maintenance and need to log into every web server, by the time you get to *www87.BertJWRegeerHasTooManyBlastedComputers.com* you'll need treatment for impending carpal tunnel syndrome. You can either provide a local domain or a list of domains to search on the first line of */etc/resolv.conf*.

The *domain* keyword tells the resolver which local domain to check, by default, for all hostnames. All of my test hosts are in the domain *michaelwlucas.com*, so I could set that as the default domain.

```
domain  michaelwlucas.com
```

Once you specify a local domain, the resolver will automatically append the domain to any short hostname. If I type `ping www`, the resolver will append the local domain and send ping(8) to *www.michaelwlucas.com*. If I give a complete hostname, such as *www.bertjwregeer.com*, though, the resolver doesn't add the default domain.

Maybe I have more than one domain I'd like to search. Use the *search* keyword to give a list of domain names to try, in order. Like `domain`, `search` must be the first line of *resolv.conf*.

```
search  michaelwlucas.com bertjwregeer.com mwl.io
```

When you use a brief hostname, such as *www*, the resolver appends the first domain name in the search list. If there's no answer, it repeats the query with the second domain name, and then the third. If I run `ping petulance`, the resolver searches for *petulance.michaelwlucas.com*, *petulance.bertwjregeer.com*, and *petulance.mwl.io*. If no such host exists in any of these domains, the search fails.

If you have neither domain nor search entries in */etc/resolv.conf*, but the machine's hostname includes a domain name, the resolver uses the local machine's domain name.

The Nameserver List

Now that your resolver knows which domains to try, tell it which nameservers to query. List each in */etc/resolv.conf* on its own line, in order of preference. Use the keyword `nameserver` and the DNS server's IP address. The resolver queries the listed nameservers in order. A complete *resolv.conf* might look like this:

```
domain mwl.io
nameserver 127.0.0.1
nameserver 203.0.113.8
nameserver 192.0.2.8
```

This resolver is ready to rock.

Note the first nameserver entry, though. The address 127.0.0.1 is always attached to the local host. This machine is running a local recursive nameserver. You can too!

Caching Nameserver

Your host needs to perform a DNS lookup every single time it must contact a host. A busy server makes a whole bunch of queries, and by itself, the resolver doesn't cache any of these responses. A host needs to connect to Google 500 times in a minute? That's 500 DNS lookups. While setting up an authoritative DNS server requires a specific skill set, configuring a local recursive server requires only one line in */etc/rc.conf*. This lets your FreeBSD host cache its DNS responses while reducing network congestion and improving performance.

Enable the local nameserver with the *rc.conf* variable `local_unbound_enable`.

```
# sysrc local_unbound_enable=YES
local_unbound_enable: NO -> YES
```

You can now start the local nameserver.

```
# service local_unbound start
```

When you start the service the first time, `unbound` configures itself. It extracts your system's nameservers from */etc/resolv.conf* and configures itself to forward all queries to those nameservers. The setup process then edits */etc/resolv.conf* to point all queries at the local nameserver, running on the IP address 127.0.0.1.

When your host makes a DNS query, the resolver queries `unbound`. The local nameserver checks its cache to see whether it has a valid and unexpired answer for the query. If it doesn't have a cached response, `unbound` queries your preferred nameservers.

I recommend enabling `local_unbound` on every server that isn't a DNS server.

Network Activity

Now that you're on the network, how can you see what's going on? There are several ways to look at the network, and we'll consider each in turn. Unlike many commercial operating systems, FreeBSD commands such as netstat(8) and sockstat(1) give you more information about the network than can possibly be healthy.

Current Network Activity

The general-purpose network management program netstat(8) displays different information depending on the flags it's given. One common question people have is, "How much traffic is my system pushing right now?" The netstat(8) `-w` (for *wait*) option displays how many packets and bytes your system is processing. The `-w` flag takes one argument, the number of seconds between updates. Adding the `-d` (for *drop*) flag tells netstat(8) to include information about packets that never made it to the system. Here, we ask netstat(8) to update its display every five seconds:

```
# netstat -w 5 -d
            input        (Total)           output
   packets  errs idrops      bytes    packets  errs      bytes colls drops
       ❶34   ❷0     ❸0    ❹44068        ❺23   ❻0      ❼1518    ❽0   ❾0
         33    0       0     42610          23    0        1518      0    0
--snip--
```

Nothing appears to happen when you enter this command, but in a few seconds, the display prints a single line of information. The first three columns describe inbound traffic, while the next three describe outbound traffic. We see the number of packets received since the last update ❶, the number of interface errors for inbound traffic since the last update ❷, and the number of inbound dropped packets ❸. The input information ends with the number of bytes received since the last update ❹. The next three columns show the number of packets the machine transmitted since the last update ❺, the number of errors in transmission since the last update ❻, and how many bytes we sent ❼. We then see the number of network collisions that have occurred since the last update ❽, and the number of packets that have been dropped ❾. For example, in this display, the system received 34 packets ❶ since `netstat -w 5 -d` started running.

Five seconds later, netstat(8) prints a second line describing the activity since the first line was printed.

You can make the output as detailed as you want and run it as long as you like. If you'd like to get updates every second, just run `netstat -w 1 -d`. If once a minute is good enough for you, `netstat -w 60 -d` will do the trick. I find a five-second interval most suitable when I'm actively watching the network,

but you'll quickly learn what best fits your network and your problems.

Hit CTRL-C to stop the report once you've had enough.

What's Listening on Which Port?

Another popular question is, "Which ports are open and what programs are listening on them?" FreeBSD includes sockstat(1), a friendly tool to answer this question. It shows both active connections and ports available for client use.

The sockstat(1) program not only lists ports listening to the network, but also any other ports (or *sockets*) on the system. Use the `-4` flag to see IPv4 sockets and `-6` to view IPv6. Here's trimmed sockstat(1) output from a *very* small server:

```
   # sockstat -4
   USER       COMMAND   PID   FD PROTO   LOCAL ADDRESS       FOREIGN ADDRESS
   mwlucas  ❶sshd       11872 4  tcp4   ❷203.0.113.43:22    ❸24.192.127.92:62937
❹  root       sendmail  11433 4  tcp4    *:25               *:*
❺  www        httpd     9048  16 tcp4    *:80               *:*
❻  root       sshd      573   3  tcp4    *:23               *:*
❼  root       sshd      426   3  tcp4    *:22               *:*
❽  bind       named     275   20 udp4    203.0.113.8:53     *:*
❾  bind       named     275   20 tcp4    203.0.113.8:53     *:*
```

The first column gives us the username that's running the program attached to the port in question. The second column is the name of the command. We then have the process ID of the program and the file descriptor number attached to the socket. The next column shows what transport protocol the socket uses—either `tcp4` for TCP on TCP/IP version 4, or `udp4` for UDP on TCP/IP version 4. We then list the local IP address and port number, and finally the remote IP address and port number for each existing connection.

Take a look at our very first entry. I'm running the program sshd ❶. A man page search takes you to sshd(8), the SSH daemon. The main sshd(8) daemon forked a child process on my behalf to handle my connection, so we see multiple instances of sshd(8) with different process IDs. I'm connected to the local IP address 203.0.113.43 ❷ on TCP port 22. The remote end of this connection is at the IP address 24.192.127.92 ❸ on port 62937. This is an

SSH connection from a remote system to the local computer.

Other available connections include Sendmail ❹, the mail server, running on port 25. Note that this entry doesn't have any IP address listed as the foreign address. This socket is listening for incoming connections. Our httpd process ❺ is listening for incoming connections on port 80.

The astute among you might notice that this server has two SSH daemons available for incoming connections, one on port 23 ❻ and one on port 22 ❼. As */etc/services* shows, SSH normally runs on port 22 while port 23 is reserved for telnet. Anyone who telnets to this machine will be connected to an SSH daemon, which won't work as they expect. The suspicious among you might suspect that this SSH server was set up to waltz around firewalls that only filter traffic based on source and destination ports and not the actual protocol. (I have no comment on such allegations.)

The last two entries are for a nameserver, named, awaiting incoming connections on port 53. This nameserver is listening for both UDP ❽ and TCP ❾ connections and is attached to the single IP address 203.0.113.8.

Port Listeners in Detail

While sockstat(1) provides a nice high-level view of network service availability, you can get a little more detailed information about individual connections with netstat(8). To view open network connections, use netstat(8)'s `-a` flag. The `-n` flag tells netstat(8) not to bother translating IP addresses to hostnames; not only can this translation slow down the output, it can cause ambiguous output. Finally, the `-f inet` option tells netstat(8) to worry only about IPv4 network connections, while `-f inet6` addresses IPv6. Here's matching netstat output from the same machine we just ran sockstat(1) on:

```
# netstat -na -f inet
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0     48  203.0.113.43.22        24.192.127.92.62937    ESTABLISHED
tcp4       0      0  *.25                   *.*                    LISTEN
tcp4       0      0  *.23                   *.*                    LISTEN
tcp4       0      0  *.80                   *.*                    LISTEN
tcp4       0      0  *.22                   *.*                    LISTEN
tcp4       0      0  203.0.113.8.53         *.*                    LISTEN
udp4       0      0  203.0.113.8.53         *.*
```

Here, we get no idea of what program is attached to any port. The first entry in each column is the transport protocol used by the socket—mostly TCP, but the last line shows UDP.

The `Recv-Q` and `Send-Q` columns show the number of bytes waiting to be handled by this connection. If you see nonzero `Recv-Q` numbers for some connection most of the time, you know that the program listening on that port can't process incoming data quickly enough to keep up with the network stack. Similarly, if the `Send-Q` column keeps having nonzero entries, you know that either the network or the remote system can't accept data as quickly as you're sending it. Occasional queued packets are normal, but if they don't go away, you might want to investigate why things are slow. You must watch your own system to learn what's normal.

The `Local Address` is, as you might guess, the IP address and network port number on the local system that the network connection is listening on. The network port appears at the end of the entry and is separated from the IP address by a dot. For example, 203.0.113.43.22 is the IP address 203.0.113.43, port 22. If the entry is an asterisk followed by a period and a port number, that means that the system is listening on that port on all available IP addresses. The system is ready to accept a connection on that port.

The `Foreign Address` column shows the remote address and port number of any connection.

Finally, the `(state)` column shows the status of the TCP handshake. You don't need to know all of the possible TCP connection states, so long as you learn what's normal. `ESTABLISHED` means that a connection exists and that data is probably flowing. `LAST_ACK`, `FIN_WAIT_1`, and `FIN_WAIT_2` mean that the connection is being closed. `SYN_RCVD`, `ACK`, and `SYN+ACK` are all parts of connection creation (the three-way handshake from Chapter 7). `LISTEN` indicates that the port is ready for incoming connections. In the preceding example, one TCP connection is running and four are ready to accept clients. As UDP is stateless, those connections list no state information.

By reading this output and combining it with information provided by sockstat(1), you can learn exactly which programs are behaving well and which are suffering bottlenecks.

If you're not interested in listening sockets but only those with active

connections, use netstat(8)'s `-b` option instead of `-a`. Running `netstat -nb -f inet` displays only connections with foreign systems.

You can also use netstat -T to display TCP retransmits and out-of-order packets on individual connections. Retransmits and misordered packets are symptoms of dropped packets.

Network Capacity in the Kernel

The FreeBSD kernel handles network memory by using mbufs. An *mbuf* is a chunk of kernel memory used for networking. You'll keep tripping across mentions of mbufs throughout the FreeBSD network stack documentation, so it's important to have at least a vague idea of them.

FreeBSD automatically allocates network capacity at boot time based on the amount of physical RAM in the system. We assume that if you have a system with 64GB RAM, you want to use more memory for networking than on a little box with 1GB RAM. View how FreeBSD uses its resources with `netstat -s` and `netstat -m`. Let's look at the shortest one first.

To get a generic view of kernel memory used for networking, run `netstat -m`. The output can be divided into two general categories: how much is used and how many requests failed. The following output is trimmed to include only a few examples of these, but they all follow the same general format:

```
# netstat -m
--snip--
❶32/372/404/❷25600 mbuf clusters in use (current/cache/total/max)
--snip--
0/0/0 requests for mbufs ❸denied (mbufs/clusters/mbuf+clusters)
--snip--
```

Here we see how many mbuf clusters are used ❶. You'd probably guess that these are related to mbufs, and you'd be right. You don't have to know exactly what mbuf clusters are; the important thing is that you know how many you can allocate ❷ and can see that you're under that limit.

Similarly, we can see how many different requests for mbufs the kernel has denied ❸. This system hasn't rejected any requests for mbufs, which means that we aren't having performance problems due to memory shortages. If your system starts rejecting mbuf requests because it's out of memory, you're in trouble. See "Optimizing Network Performance" next.

While `netstat -m` produces a dozen lines of output, `netstat -s` runs for pages and pages. It provides per-protocol performance statistics. Much like `netstat -m`, you can break up these statistics into categories of how much was done and how many problems you had. Run both of these commands occasionally on your systems and review the results so you know what passes for normal on your servers and can recognize abnormal numbers when you have problems.

Optimizing Network Performance

Now that you can see what's going on, how could you improve FreeBSD's network performance? There's a simple rule of thumb when considering optimizing: *don't*. Network performance is generally limited only by your hardware. Many applications can't process data as quickly as your network can provide. If you think that you need to optimize your performance, you're probably looking in the wrong spot. Check Chapter 21 for hints on investigating performance bottlenecks.

Generally speaking, network performance should be adjusted only when you experience network problems. This means that you should have output from `netstat -m` or `netstat -s` indicating that the kernel is having resource problems. If the kernel starts denying requests for resources or dropping connection requests, look at the hints in this section. If you have issues or if you think you should be getting better performance, look at the hardware first.

Optimizing Network Hardware

Not all network hardware is created equal. While anyone in IT hears this frequently, FreeBSD's open nature makes this obvious. For example, here's a comment from the source code of the rl(4) network card driver:

```
The RealTek 8139 PCI NIC redefines the meaning of 'low end.' This is
probably the worst PCI ethernet controller ever made, with the possible
exception of the FEAST chip made by SMC. The 8139 supports bus-master
DMA, but it has a terrible interface that nullifies any performance
gains that bus-master DMA usually offers.
```

This can be summarized as, "This card sucks and blows at the same time.

Buy another card." While this is the most vitriolic comment that I've seen in the FreeBSD source code, and this particular hardware is very hard to find today, the drivers for certain other cards say the same thing in a more polite manner. Optimizing network performance with low-end hardware is like putting a high-performance racing transmission in your 1974 Gremlin. Replacing your cheap network card will probably fix your problems. Generally speaking, Intel makes decent network cards; they maintain a FreeBSD driver for their wired network cards and provide support so that the FreeBSD community can help maintain the drivers. (Wireless cards are another story.) Similarly, many companies that build server-grade machines make a point of using server-grade network cards. Some companies provide a FreeBSD driver but do not provide documentation for their hardware. This means that the driver probably works, but you're entirely dependent upon the vendor's future fondness of FreeBSD for your updates. Companies that specialize in inexpensive consumer network equipment are not your best choice for high-performance cards—after all, the average home user has no idea how to pick a network card, so they go by price alone. If in doubt, check the FreeBSD-questions mailing list archives for recent network card recommendations.

Similarly, switch quality varies wildly. The claim that a switch speaks the protocol used in gigabit connections doesn't mean that you can actually push gigabit speed through every port! I have a 100Mb switch that bottlenecks at 15Mbps and a "gigabit" switch that seems to choke at about 50Mbps. I recommend that you think of a switch's speed as a protocol or a language: I could claim that I speak Russian, but 30 years after my studies ceased, my speech bottlenecks at about three words a minute. My Russian language interface is of terrible quality. Again, switches designed for home use are not your best choice in a production environment.

If getting decent hardware doesn't solve your problems, read on.

Memory Usage

FreeBSD uses the amount of memory installed in a system to decide how much memory space to reserve for mbufs. Don't adjust the number of mbufs you create unless `netstat -m` tells you that you're short on mbuf space. If you have an mbuf problem, the real fix is to add memory to your machine. This will make FreeBSD recompute the number of mbufs created at boot and

solve your problem. Otherwise, you'll just shift the problem to a different part of the system or a different application. You might configure gobs of memory for network connections and find that you've smothered your database server. If you're sure you want to proceed, though, here's how you do it.

Two sysctl values control mbuf allocation, `kern.maxusers` and `kern.ipc.nmbclusters`. The first, `kern.maxusers`, is a boot-time tunable. Your system automatically determines an appropriate `kern.maxusers` value from the system hardware at boot time. Adjusting this value is probably the best way to scale your system as a whole. In older versions of FreeBSD, `kern.maxusers` preallocated memory for networking and refused to release it for other tasks, so increasing `kern.maxusers` could badly impact other parts of the system. Modern FreeBSD does not preallocate network memory, however, so this is just an upper limit on networking memory. If `kern.maxusers` is too small, you'll get warnings in */var/log/messages* (see Chapter 21).

The sysctl `kern.ipc.nmbclusters` specifically controls the number of mbufs allocated by the system for data sitting in socket buffers, waiting to be sent to or read by an application. Although this is runtime tunable, it's best to set it early at boot by defining it in */etc/sysctl.conf* (see Chapter 6). If you set this too high, however, you can actually starve the kernel of memory for other tasks and panic the machine.

```
# sysctl kern.ipc.nmbclusters
kern.ipc.nmbclusters: 25600
```

Mbufs are allocated in units called *nmbclusters* (sometimes called *mbuf clusters*). While the size of an mbuf varies, one cluster is about 2KB. You can use simple math to figure out how much RAM your current nmbcluster setting requires and then calculate sensible values for your system and applications. This example machine has 25,600 nmbclusters, which means the kernel can use up to about 50MB RAM for networking purposes. This is negligible on my test laptop's gig of RAM, but it might be unsuitable on an embedded system.

To calculate an appropriate number of mbuf clusters, run `netstat -m` when the server is really busy. The second line of the output will give you the number of mbufs in use and the total number available. If your server at its busiest doesn't use nearly as many nmbclusters as it has available, you're

barking up the wrong tree—stop futzing with mbufs and replace your hardware already.[4] For example:

```
❶32/❷372/❸404/❹25600 mbuf clusters in use (current/cache/total/max)
```

This system is currently using 32 nmbclusters ❶ on this machine and has cached 372 previously used nmbclusters ❷. With this total of 404 clusters ❸ in memory at this time, our capacity of 25,600 clusters ❹ is 1.5 percent utilized. If this is your real system load, actually reducing the number of nmbclusters might make sense.

My personal rule of thumb is that a server should have enough mbufs to handle twice its standard high load. If your server uses 25,000 nmbclusters during peak hours, it should have at least 50,000 available to handle those brief irregular peaks.

ONCE-IN-A-LIFETIME VS. STANDARD LOAD

Always distinguish planning for once-in-a-lifetime events from planning for normal load. When the US Government's Affordable Care Act health insurance registration site went live, millions of users immediately tried to sign up. The first few days, the site was fiendishly slow. After a week, the hardware handled the load without trouble. This was certainly correct capacity planning.

Maximum Incoming Connections

The FreeBSD kernel provides capacity to handle a certain number of incoming new TCP connections. This doesn't refer to connections that the server previously received and is handling, but rather to clients who are attempting to initiate connections simultaneously. For example, the web pages currently being delivered to clients don't count, but the incoming requests that have reached the kernel but not the web server process do. It's a *very* narrow window, but some sites do overflow it.

The sysctl `kern.ipc.somaxconn` dictates how many simultaneous connection attempts the system will try to handle. This defaults to 128, which might not be enough for a highly loaded web server. If you're running a high-capacity server where you expect more than 128 new requests to be arriving simultaneously, you probably need to increase this sysctl. If users start complaining that they can't connect, this might be your culprit. Of course, very few applications will accept that many simultaneous new connections; you'll probably have to tune your app well before you hit this point.

Polling

Some gigabit cards can improve their performance with *polling*. Polling takes the time-honored idea of interrupts and IRQs and boots it out the window, replacing it with regular checks for network activity. In the classic interrupt-driven model, whenever a packet arrives at the network card, the card demands attention from the CPU by generating an interrupt. The CPU stops whatever it's doing and handles that data. This is grand, and even desirable, when the card doesn't process a huge amount of traffic. Once a system starts handling large amounts of data, however, the card generates interrupts continuously. Instead of constantly interrupting, the system is more efficient if the kernel grabs network data from the card at regular intervals. This regular checking is called *polling*. Generally speaking, polling is useful only if you push large amounts of traffic.

Polling isn't available as a kernel module as of this writing, since it requires modifications to device drivers. This also means that not all network cards support polling, so be sure to check polling(4) for the complete list. Enable polling by adding `DEVICE_POLLING` to your kernel configuration. After your reboot, enable polling on a per-interface basis with ifconfig(8).

```
# ifconfig em0 polling
```

Similarly, disable polling with the argument `-polling`. The ifconfig(8) command also displays if polling is enabled on an interface. As you can enable and disable polling on the fly, enable polling when your system is under a heavy load and see whether performance improves.

Polling is used only on older cards. 10GB cards and faster can't poll.

Other Optimizations

FreeBSD has about 200 networking-related sysctls. You have all the tools you need to optimize your system so greatly that it no longer passes any traffic at all. Be very careful when playing with network optimizations. Many settings that seem to fix problems actually fix only one set of problems while introducing a whole new spectrum of issues.

Some software vendors (i.e., Samba) recommend particular network sysctl changes. Try them cautiously, and watch for unexpected side effects on other programs before accepting them as your new default. TCP/IP is a terribly, terribly complicated protocol, and FreeBSD's defaults and autotuning reflect years of experience, testing, and sysadmin suffering.

Also remember that FreeBSD is over two decades old. Mailing list and forum posts from more than a few years ago are probably not useful in network tuning.

Network Adapter Teaming

As network servers become more and more vital to business, redundancy becomes more important. We have redundant hard drives in a server and redundant bandwidth into a data center, but what about redundant bandwidth into a server? On a smaller scale, as you move around your office, you might move your laptop between wired and wireless connections. It would be really nice not to lose your existing SSH sessions because you unplugged a cable.

FreeBSD can treat two network cards as a single entity, allowing you to have multiple connections with a single switch. This is commonly called *network adapter teaming*, *bonding*, or *link aggregation*. FreeBSD implements adapter teaming through lagg(4), the link aggregation interface.

TRUNKING, TEAMING, AND VLANS

Some vendors use the word *trunk* to describe link aggregation. Other vendors use the word *trunk* to describe one cable with multiple networks (VLANs). FreeBSD avoids

this argument by not using the word *trunk*. If someone starts discussing trunks in your presence, ask them what kind they're talking about.

The kernel module lagg(4) provides a lagg0 virtual interface. You assign physical interfaces to the lagg0 interface, making them part of the aggregated link. While you could use lagg(4) with only one physical interface, aggregating links only makes sense when you have two or more physical interfaces to assign to the aggregated link. The lagg(4) module allows you to implement seamless roaming between wired and wireless networks, failover, and several different aggregation protocols.

Aggregation Protocols

Not all network switches support all link aggregation protocols. FreeBSD has basic implementation of some complicated high-end protocols and also includes very basic failover setups. The three I recommend are Fast EtherChannel, LACP, and failover. (There are more schemes, which you can read about in lagg(4).)

Cisco's *Fast EtherChannel (FEC)* is a reliable link aggregation protocol, but it works only on high- to medium-end Cisco switches running particular versions of Cisco's operating system. If you have an unmanaged switch, Fast EtherChannel is not a viable choice. Fast EtherChannel is complicated to configure (on the switch), so I recommend FEC only when it is already your corporate standard for link aggregation.

The *Link Aggregation Control Protocol (LACP)* is an industry standard for link aggregation. The physical interfaces are bonded into a single virtual interface with approximately the same bandwidth as all of the individual links combined. LACP provides excellent fault tolerance, and almost all switches support it. I recommend LACP unless you have a specific requirement for Fast EtherChannel or a switch that chokes when you use LACP.

If you do have a switch that chokes on LACP, use *failover*. The failover method sends traffic through one physical interface at a time. If that interface goes down, the connection fails over to the next connection in the

pool. While you don't get aggregated bandwidth, you do get the ability to attach your server to multiple switches for fault tolerance. Use failover to let your laptop roam between wired and wireless connections.

Configuring lagg(4)

The lagg interface is virtual, meaning there is no physical part of the machine that you could point to and say, "That is interface lagg0." Before you can configure the interface, you must create it. FreeBSD lets you create interfaces with `ifconfig` *`interfacename`* `create`, but you can also do this in */etc/rc.conf* with the `cloned_interfaces` statement.

Configuring a lagg(4) interface in *rc.conf* has three steps: creating the interface, bringing up the physical interfaces, and aggregating them. Here, we create a single lagg0 interface out of two Intel gigabit Ethernet cards, em0 and em1.

```
cloned_interfaces="lagg0"
ifconfig_em0="up"
ifconfig_em1="up"
ifconfig_lagg0="laggproto lacp laggport em0 laggport em1 inet 203.0.113.1/24"
```

First, you list lagg0 as a cloned interface, so FreeBSD will create this interface at boot. Then, bring interfaces em0 and em1 up, but don't configure them. Finally, tell the lagg0 interface what aggregation protocol to use (LACP), what physical interfaces belong to it, and its network information. These few lines of configuration give you a high-availability Ethernet connection.

Virtual LANs

A *virtual LAN*, or *VLAN*, lets you get multiple Ethernet segments on a single piece of wire. You'll sometimes see VLANs called *802.1q*, *tagging*, or a combination of these terms. You can use these multiple networks by configuring additional logical interfaces attached to a physical interface. The physical wire can still carry only so much data, however, so all VLANs and the regular network (or *native VLAN*) that share the wire use a common pool of bandwidth. If you need a FreeBSD host on multiple Ethernet segments simultaneously, this is one way to do it without running more

cable.

VLAN frames that arrive at your network card are like regular Ethernet frames, tagged with an additional header that says "This is part of VLAN number whatever." Each VLAN is identified by a tag from 1 to 4096. The native VLAN arrives without any tagging whatsoever. The network often (but not always) calls this *VLAN 1* internally.

Configuring a VLAN on your FreeBSD host doesn't magically connect the host to the VLAN. The network must be configured to send those VLANs to your host. You must work with the network team to get access to the VLANs.

Configuring VLAN Devices

Use ifconfig(8) to create VLAN interfaces. You must know the physical interface and the VLAN tag.

```
# ifconfig interface.tag create vlan tag vlandev interface
```

Here, I create an interface for VLAN 2 and attach it to the interface em0.

```
# ifconfig em0.2 create vlan 2 vlandev em0
```

I can now configure interface em0.2 as I would a physical interface.

```
# ifconfig em0.2 inet 192.0.2.236/28
```

In reality, I'd probably do all of this in a single command.

```
# ifconfig em0.2 create vlan 2 vlandev em0 inet 192.0.2.236/28
```

That's everything. Now use ifconfig(8) to display your new interface.

```
# ifconfig em0.2
em0.2: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
        options=503<RXCSUM,TXCSUM,TSO4,LRO>
        ether 00:25:90:db:d5:94
        inet 198.22.65.236 netmask 0xffffff00 broadcast 255.255.255.240
        inet6 fe80::225:90ff:fedb:d594%em0.2 prefixlen 64 scopeid 0x6
        nd6 options=21<PERFORMNUD,AUTO_LINKLOCAL>
     ❶ media: Ethernet autoselect (100baseTX <full-duplex>)
        status: active
     ❷ vlan: 2 vlanpcp: 0 parent interface: em0
```

```
❸ groups: vlan
```

This looks almost exactly like any other physical interface. The media information ❶ comes directly from the underlying interface. You'll see a label with VLAN information ❷ and a note that this is grouped with the other VLAN interfaces ❸.

Configuring VLANs at Boot

Configure VLANs at boot with *rc.conf* variables. First, use a `vlan_` variable tagged with the interface name to list the VLANs attached to that interface. Here, I tell FreeBSD to enable VLAN 2 and 3 on interface em0 and assign an IP configuration to each.

```
vlans_em0="2 3"
ifconfig_em0_2="inet 192.0.2.236/28"
ifconfig_em0_3="inet 198.51.100.50/24"
```

If the underlying interface has no configuration, you need to at least bring it up. The VLAN interfaces won't work unless the physical interface is on.

```
ifconfig_em0="up"
```

You now have virtual LANs at boot. Congratulations!

Now that you have a working network, let's get a little more local and look at basic system security.

9
SECURING YOUR SYSTEM

Securing your system means ensuring that your computer's resources are used only by authorized people for authorized purposes. Even if you have no important data on your system, you still have valuable CPU time, memory, and bandwidth. Many folks who thought that their systems were too unimportant for anyone to bother breaking into found themselves an unwitting relay for an attack that disabled a major corporation. You don't want to wake up one morning to the delightful sound of law enforcement agents kicking in your door because your insecure computer broke into a bank.

Sure, there are things worse than having some punk kid take over your servers—say, having the neighborhood loan shark break both your legs. Discovering that your organization's web page now says, "Ha ha, you've been r00ted!" is a decent competitor for second place. Even more comprehensible intrusions cause huge headaches. Most of the actual intrusions I've been involved with (not as an attacker, but as a consultant to the victim) have originated from countries with government censorship, and traffic analysis showed that the intruders were actually just looking for unrestricted access to news sites. While I fully sympathize with these people, when I'm depending upon the stable operation of my servers to run my business, their intrusion is unacceptable.

Over the last few years, taking over remote computers has become much easier. Point-and-click programs for subverting computers can be found through search engines. When one bright attacker writes an exploit, several thousand bored teenagers with nothing better to do can download it and

make life difficult for the rest of us. Even if the data on your system is worthless, you must secure the system's resources.

Generally speaking, operating systems are not broken into; the programs running on operating systems are. Even the most paranoiac, secure-by-default operating system in the world can't protect badly written programs from themselves. Occasionally, one of those programs can interact with the operating system in such a way as to actually compromise the operating system. The most well-known of these are *buffer overflows*, where an intruder's program is dumped straight into the CPU's execution space and the operating system runs it. FreeBSD has undergone extensive auditing to eliminate buffer overflows as well as myriad other well-understood security issues, but that's no guarantee that they've been eradicated. New functions and programs appear continuously, and they can interact with older functions and each other in unexpected ways.

FreeBSD provides many tools to help you secure your system against attackers, both internal and external. While no one of these tools is sufficient, all are desirable. Treat everything you learn about system security as a tool in a kit, not as the answer to all your problems. For example, while simply raising a system's securelevel will not make your system secure, it can help when combined with reasonable permissions, file flags, regular patching, password control, and all the other things that make up a good security policy. We'll cover more advanced security tools in Chapter 19, but without the basic protections discussed here, those tools won't help secure your system.

Who Is the Enemy?

We'll arbitrarily lump potential attackers into four groups: script kiddies, disaffected users, botnets, and skilled attackers. You'll find a more detailed classification in books dedicated to security, but that's not what you're here for. These categories are easily explained, easily understood, and include 99 percent of all the attackers you're likely to encounter.

Script Kiddies

The most numerous human attackers, *script kiddies*, are not sysadmins.

They're not skilled. They download attack programs that work on a point-and-click basis and go looking for vulnerable systems. They're the equivalent of purse snatchers, preying upon old ladies holding their bags just a little bit too loosely. Fortunately, script kiddies are easy to defend against: just keep your software up-to-date and follow good computing practices. Like locusts, script kiddies are easy to squash, but there are just so darned *many* of them!

Disaffected Users

The second group, your own users, causes the majority of security problems. Your organization's employees are the people most likely to know where the security gaps are, to feel that the rules don't apply to them, and to have the time to spend breaking your security. If you tell an employee that company policy forbids him access to a computer resource, and if the employee feels that he should have access to it, he's likely to search for a way around the restriction. Anyone who feels that he's so fabulously special that the rules don't apply to him is a security risk. Worse, when an employee who knows all the dirty laundry gets angry, bad things can happen. You might have all your servers patched and a downright misanthropic firewall installed, but if anyone who knows the password is *Current93* can dial the back room modem, you're in trouble.

The best way to stop people like these is simply not to be sloppy. Don't leave projects insecurely half-finished or half-documented. When someone leaves the company, disable his account, change all administrative passwords, inform all employees of that person's departure, and remind them not to share confidential knowledge with that person. Have a computer security policy with real violation penalties and have HR enforce it. And get rid of the unsecured modem, the undocumented telnet server running on an odd port, or whatever hurried hack you put into place thinking that nobody would ever find it.

Botnets

Botnets are more numerous than either of the above, but they're not human. They're machines compromised by malware and controlled from a central point. Botnets can include millions of machines. The malware authors control the botnets and use them for anything from searching for more

vulnerable hosts to sending spam or breaking into secure sites. Most botnets are composed of Windows and Linux machines, but there's no reason why FreeBSD operating systems can't be assimilated into botnets.

Fortunately, botnet defense is much like script kiddie defense; keeping your software patched and following good computing practices goes a long way.

Motivated Skilled Attackers

The most dangerous group—skilled attackers—are competent system administrators, security researchers, penetration specialists, and criminals who want access to your specific resources. Computer penetration is a lucrative criminal field these days, especially if the victim has resources that can be used for distributed denial-of-service (DDos) attacks or mass spam transmission. Compromising a web farm and turning it to evil is profitable. If you have valuable company secrets, you might be targeted by one of these intruders. If one of these people *really* wants to break into your network, he'll probably get there.

Still, security measures that stop the first three groups of people change the tactics of the skilled attacker. Instead of breaking into your computers over the network, he'll have to show up at your door dressed as a telco repairman lugging a packet sniffer, or dumpster-dive searching for old sticky notes with scribbled passwords. This dramatically increases his risk, possibly making an intrusion more trouble than it's worth. If you can make the intruder's break-in plan resemble a Hollywood script *no matter how much he knows about your network*, your security is probably pretty good.

Hackers, Intruders, and Related Scum

You'll frequently hear the word *hacker* used to describe people who break into computers. This word has different meanings depending on the speaker. In the technical world, a hacker is someone who's interested in the inner workings of technology. Some hackers are interested in everything; others have a narrow area of specialization. In the open source community, *hacker* is a title of respect. The main FreeBSD

technical list is *FreeBSD-hackers@FreeBSD.org*. In the popular media, however, a hacker is someone who breaks into computer systems, end of story.

To reduce confusion, I recommend completely avoiding the word *hacker*. In this book, I call people who break into computers *intruders*.* Technical wizards can be called by a variety of names, but they rarely object to "Oh Great and Powerful One."

* Two editions later, and my editor *still* won't let me print what I call them in person.

FreeBSD Security Announcements

The best defense against any attackers is to keep your system up to date. This means you must know when to patch your system, what to patch, and how. An outdated system is a script kiddie's best friend.

The FreeBSD Project includes volunteers who specialize in auditing source code and watching for security issues with both the base operating system and add-on software. These developers maintain a very low-volume mailing list, *FreeBSD-security-notifications@FreeBSD.org*, and subscribing is a good idea. While you can monitor other mailing lists for general announcements, the security notifications list is a single source for FreeBSD-specific information. To subscribe to the security notifications mailing list, see the instructions on *http://lists.freebsd.org/*. The FreeBSD security team releases advisories on that mailing list as soon as they're available.

Read advisories carefully and quickly act on those that affect you, as you can be certain that script kiddies are searching for vulnerable machines. FreeBSD makes applying security patches pretty easy, as Chapter 18 discusses.

User Security

Remember when I said that your own users are your greatest security risk? Here's where you learn to keep the little buggers in line. FreeBSD has a variety of ways to allow users to do their work without giving them free rein on the system. We'll look at the most important tools here, starting with adding users in the first place.

Creating User Accounts

FreeBSD uses the standard Unix user management programs such as passwd(1), pw(8), and vipw(8). FreeBSD also includes a friendly interactive user-adding program, adduser(8). Only *root* may add users, of course. Just type `adduser` on the command line to enter an interactive shell.

The first time you run adduser(8), it prompts you to set appropriate defaults for all new user settings. Use the following example session to help you determine appropriate defaults for your system.

```
   # adduser
❶ Username: xistence
❷ Full name: Bert Reger
❸ Uid (Leave empty for default):
```

The username ❶ is the name of the account. Users on my systems get a username of their first initial, middle initial, and last name. You can assign usernames by whatever scheme you dream up. Here, I let the user pick their own username, an indulgence I always later regret. The full name ❷ is the user's real name. FreeBSD then lets you choose a numerical user ID (UID) ❸. FreeBSD starts numbering UIDs at 1,000; while you can change this, all UIDs below 1,000 are reserved for system use. I recommend just pressing ENTER to take the next available UID.

```
❶ Login group [xistence]:
❷ Login group is xistence. Invite xistence into other groups? []: www
❸ Login class [default]:
❹ Shell (sh csh tcsh nologin) [sh]: tcsh
❺ Home directory [/home/xistence]:
❻ Home directory permissions (Leave empty for default):
```

The user's default group ❶ is important—remember, Unix permissions

are set by owner and group. The FreeBSD default of having each user in their own group is usually the most sensible way for most setups. Any of the big thick books on system administration offers several grouping schemes—feel free to use whatever matches your needs. You can add this user to other groups ❷ in addition to the primary group at this time, if appropriate.

A login class ❸ specifies what level of resources the user has access to. We'll talk about login classes later in this section.

The shell ❹ is the command line environment. While the system default is */bin/sh*, I prefer tcsh.[1] If you're deeply attached to another shell, feel free to use it instead. Knowledgeable users can change their own shells.

The home directory ❺ is where the user's files reside on disk. The user and that user's primary group own this directory. You can set custom permissions ❻ on the directory if you want, probably so that other users can't view this user's directory.

```
❶ Use password-based authentication? [yes]:
❷ Use an empty password? (yes/no) [no]:
❸ Use a random password? (yes/no) [no]: y
❹ Lock out the account after creation? [no]: n
```

The password options give you a certain degree of flexibility. If all of your users are comfortable with key-based SSH authentication, perhaps you can get away without using passwords. In the meantime, the rest of us are stuck with passwords ❶.

Use an empty password ❷ if you want the user to set his or her own password via the console. Whoever connects to that account first gets to set the password. This makes an empty password a good idea right up there with smoking inside a hydrogen dirigible.

A random password ❸, on the other hand, is a good idea for a new account. The random password generator FreeBSD provides is good enough for day-to-day use. Random passwords are usually hard to remember, which encourages the user to change his password as soon as possible.

When an account is locked ❹, nobody can use it to log in. This is generally counterproductive.

After entering all this information, adduser spits everything back at you for review and confirmation or rejection. Once you confirm, adduser verifies the account setup and provides you with the randomly generated password. It then creates the user's home directory, copies the shell configuration files from */etc/skel*, and asks whether you want to set up another user.

Configuring Adduser: /etc/adduser.conf

Creating new users on some Unix systems requires you to manually edit */etc/passwd*, rebuild the password database, edit */etc/group*, create a home directory, set permissions on that home directory, install dotfiles, and so on. This makes handling your local customizations routine—if you set everything by hand, you can manage your local account setup easily. The adduser(8) program provides a set of sensible defaults. For sites with different requirements, */etc/adduser.conf* lets you set those requirements as defaults while retaining the high degree of automation. To create *adduser.conf file*, run `adduser -C` and answer the questions.

```
❶ Uid (Leave empty for default):
❷ Login group []:
❸ Enter additional groups []: staff
❹ Login class [default]: staff
❺ Shell (sh csh tcsh nologin) [sh]: tcsh
❻ Home directory [/home/]: /nfs/u1/home
❼ Use password-based authentication? [yes]:
❽ Use an empty password? (yes/no) [no]:
❾ Use a random password? (yes/no) [no]: yes
❿ Lock out the account after creation? [no]: no
```

You might want to start numbering UIDs somewhere other than 1,000. If you want higher initial UIDs, enter it in the `Uid` space ❶. Don't start below 1,000.

The login group ❷ is the default user group. An empty login group means the user account defaults to having its own unique user group (the FreeBSD default).

You can specify any additional user groups ❸ that new accounts belong to by default, as well as the login class ❹. I set both of these to `staff` so that all

new users get added to that group and assigned that class.

Choose a default shell ❺ for your users.

Your home directory location ❻ might vary from the standalone FreeBSD standard. In this example, I've specified a typical style of NFS-mounted home directories used when many users have accounts on many machines.

Choose the default password behavior for new users. You can specify whether users should use passwords at all ❼ and whether the initial password should be empty ❽ or random ❾.

Finally, dictate whether new accounts should default to being locked or not ❿.

You'll find more configuration settings in adduser.conf(5). While you can set account characteristics here, the format of this file is considered internal to adduser(8). The setting names can change with any FreeBSD release. To change *adduser.conf*, re-run `adduser -C`.

Editing Users

Managing users isn't just about creating and deleting accounts. You'll need to change those accounts from time to time. While FreeBSD includes several tools for editing accounts, the simplest are passwd(1), chpass(1), vipw(8), and pw(8). These work on the tightly interrelated files */etc/master.passwd*, */etc/passwd*, */etc/spwd.db*, and */etc/pwd.db*. We'll start with the files and then review the common tools for editing those files.

The files */etc/master.passwd*, */etc/passwd*, */etc/spwd.db*, and */etc/pwd.db* hold user account information. Each file has a slightly different format and purpose. The file */etc/master.passwd* is the authoritative source of user account information and includes user passwords in encrypted form. Normal users don't have permission to view the contents of */etc/master.passwd*. Regular users need access to basic account information, however; how else can unprivileged system programs identify users? The file */etc/passwd* lists user accounts with all privileged information (such as the encrypted password) removed. Anyone can view the contents of */etc/passwd* to get basic account information.

Many programs need account information, and parsing a text file is

notoriously slow. In this day of laptop supercomputers, the word *slow* isn't very meaningful, but this was a very real problem back when disco freely roamed the earth. For that reason, BSD-derived systems build a database file out of */etc/master.passwd* and */etc/passwd*. (Other Unix-like systems have similar functionality in different files.) The file */etc/spwd.db* is taken directly from */etc/master.passwd* and contains sensitive user information, but it can be read only by root. The file */etc/pwd.db* can be read by anyone, but it contains the limited subset of information contained in */etc/passwd*.

Any time any standard user management program changes the account information in */etc/master.passwd*, FreeBSD runs pwd_mkdb(8) to update the other three files. For example, the three programs passwd(1), chpass(1), and vipw(8) all allow you to make changes to the master password file, and all three programs trigger pwd_mkdb to update the related files.

Changing a Password

Use passwd(1) to change passwords. A user can change his own password, and root can change anyone's password. To change your own password, just enter `passwd` at the command prompt.

```
# passwd
Changing local password for mwlucas
Old Password:
New Password:
Retype New Password:
```

When changing your own password, passwd(1) first asks for your current password. This is to ensure that nobody else can change your password without your knowledge. It's always good to log out when you walk away from your terminal, but when you don't, this simple check in passwd(1) prevents practical jokers from really annoying you. Then enter your new password twice, and it's done. When you're the superuser and want to change another user's password, just give the username as an argument to passwd.

```
# passwd mwlucas
Changing local password for mwlucas
New Password:
Retype New Password:
```

Note that root doesn't need to know the user's old password; the root

user can change any user account on the system in any manner desired.

USER MANAGEMENT AND $EDITOR

User management tools such as chpass and vipw (as well as many other system management tools) bring up a text editor window where you make your changes. These tools generally check the environment variable `$EDITOR` to see which text editor you prefer. `$EDITOR` lets you default to vi, Emacs, or any other editor installed. I recommend Vigor, a vi(1) clone with an animated-paperclip help system that might make users of older Microsoft Office versions feel more comfortable.

Changing Accounts with chpass(1)

The account has more information associated with it than just the password. The chpass(1) utility lets users edit everything they can reach in their account. For example, if I run chpass(1) as a regular user, I get an editor with the following text:

```
#Changing user information for mwlucas.
Shell: /bin/tcsh
Full Name: Michael W Lucas
Office Location:
Office Phone:
Home Phone:
Other information:
```

I'm allowed to edit six informational fields in my account. The first, my shell, can be set to any shell listed in */etc/shells* (see "Shells and */etc/shells*" on page 178). I can change my full name; perhaps I want my full middle name listed, or maybe I wish to be known to other system users as *Mr. Scabies*. I can update my office location and office phone so my coworkers can find me easily. This is another feature that was very useful on the university campuses where BSD grew up and where system users rarely had an idea of anyone's physical location. Now that we have extensive online directories and many more computers, it's less useful. I generally set my home phone number to 911 (999 in the UK), and I put a little bit of personal information

in the `Other` space.

Also note what I *can't* change as a regular user. The sysadmin sets my home directory, and I may not change it even if the system has a new hard drive with lots of empty space for my MP3 collection. My UID and GID numbers, similarly, are assigned by the system or the sysadmin.

On the other hand, if I run `chpass xistence`, its heightened privileges give me a very different view.

```
   #Changing user information for xistence.
   Login: xistence
❶ Password: $6$D9b4FFD0kHK2sPSP$bXUFTQqV/QposXw2KTlswzpvoz4HBo8...
   Uid [#]: 1001
   Gid [# or name]: 1001
   Change [month day year]:
   Expire [month day year]:
   Class:
   Home directory: /home/xistence
   Shell: /bin/tcsh
   Full Name: Bert Regeer
   Office Location:
   Office Phone:
   Home Phone:
   Other information:
```

As root, you can do anything you like to the poor user. Changing his login to *megaloser* is only the start of the havoc you can wreak. You even get access to the user's hashed password ❶. Don't alter this field, unless you're comfortable computing password hashes. Use passwd(1) to more safely and reliably change the user's password. You can also change the user's home directory, although chpass(1) doesn't move the user's files; you must copy them by hand.

You can also set a date for password changes and account expiration. Password expiration is useful if you've just changed a user's password and you want him to change it upon his first login. Account expiration is useful when someone asks for an account but insists it's needed only for a limited time. You can forget to go back to delete that account, but FreeBSD never forgets. Both of these fields take a date in the form *month day year*, but you need only the first three letters of the month. For example, to make a user's password expire on June 8, 2028, I would enter `Jun 8 2028` in the `Change` space. Once the user changes his password, the password expiration field is blanked out again, but only the system administrator can extend an account

expiration date.

The Big Hammer: vipw(8)

While chpass(1) is fine for editing individual accounts, what happens when you must edit many accounts? Suppose your system has hundreds of users and a brand new hard disk for the home partition. Do you really want to run chpass(1) hundreds of times? That's where vipw(8) comes in.

Directly edit */etc/master.passwd* with vipw(8). When you finish your edits, vipw(8) checks the password file's syntax to be sure you haven't ruined anything. Then, it saves the new password file and runs pwd_mkdb(8). Although vipw(8) can protect your password file from many basic mistakes, if you're clever, you can still muck things up. You must understand the format of the password file to use vipw(8) properly.

If the information in */etc/master.passwd* conflicts with information in other files, */etc/master.passwd* wins. For example, the primary group that appears in */etc/master.passwd* is correct, even if */etc/group* doesn't show the user as a member. This "*master.passwd* is always correct" logic is deeply ingrained throughout user management.

Each line in */etc/master.passwd* is a single account record, containing 10 colon-separated fields. These fields are the following:

Username

This field is either an account name created by the sysadmin or a username created at install time to provide some system service. FreeBSD includes users for system administration, such as root, daemon, games, and so on. Each of these users owns a part of the base system. FreeBSD also provides accounts for common services, such as the *www* user reserved for use by web servers. Add-on software might add its own system accounts as well.

Encrypted Password

The second field is the encrypted password. System users don't have a password, so you can't log in as one of them. User accounts have a string of random-looking characters here.

User ID

The third field is the *user ID number*, or *UID*. Every user has a unique UID.

Group ID

Similarly, the fourth field is the *group ID number*, or *GID*. This is the user's primary group. Usually this is identical to the UID, and the group has the same name as the username.

User's Class

The next field is the user's class as defined in */etc/login.conf* (see "Restricting System Usage" on page 188).

Password Expiration

This field is the same as the password expiration date set via chpass(1), but here the time gets stored as seconds from the epoch. Use `date -j` and the `+%s` output format to generate epochal seconds from a real date. To convert midnight, June 1, 2018, to epochal seconds, run `date -j 201806010000 '+%s'`.

Account Expiration

This field enables you to make the account shut itself off on a certain day. Just set the account expiration date as you would for password expiration.

Personal Data

This field is also known as the *gecos* field for obscure historical reasons. It contains the user's real name, office number, work phone number, and home phone number, all separated by commas. Do not use colons in this field; */etc/master.passwd* reserves colons as a field delimiter.

User's Home Directory

The ninth field is the user's home directory. While this defaults to */home/<username>*, you can move this anywhere appropriate. You'll also need to move the actual home directory and its files when you change this field. Users with a nonexistent home directory can't log in by default, although the `requirehome` setting in *login.conf* can change this.

User's Shell

The final field is the user's shell. If this field is empty, the system assigns the user the boring old */bin/sh*.

While chpass(1) lets you muck up individual user accounts, vipw(8) unleashes you on the entire userbase. Be careful with it!

Removing a User

The rmuser(8) program deletes user accounts. You'll be prompted for the username you want to delete and asked whether you want to remove that user's home directory. That's really all you have to do; destruction is much easier than creation, after all.

Scripting with pw(8)

The pw(8) command provides a powerful command line interface to user accounts. While useradd(8) walks you through setting up an account in a friendly manner, pw(8) lets you specify everything on a single command line. I find pw(8) cumbersome for day-to-day use, but if you manage many user accounts, it's invaluable.

One thing I do use pw(8) for is locking accounts. While a locked account is active, nobody can log in to it. I've used this to great effect when a client was behind on a bill; users call quite quickly when they can't log in, and yet their websites continue to come up and their email continues to accumulate.

```
# pw lock xistence
```

When Bert apologizes, I'll unlock his account.

```
# pw unlock xistence
```

If you need scripts to manage your users, definitely read the pw(8) man page.

Shells and /etc/shells

The *shell* is the program that provides the user's command prompt. Different shells behave differently and offer different shortcuts and features. Many

people are very attached to particular shells and complain bitterly if their shell isn't available on a system. The packages collection contains many shells.

The file */etc/shells* contains a list of all legitimate user shells. When you install a shell from a port or a package, it adds an appropriate entry in */etc/shells*. If you compile your own shell from source, without using a FreeBSD port, you must list the shell by its complete path in */etc/shells*.

The FTP daemon won't allow a user to log in via FTP if his shell isn't listed in */etc/shells*. If you use */sbin/nologin* as an FTP-only user shell, you must add it to this file, although a better way to handle such users is with login classes, as discussed later in this chapter.

root, Groups, and Management

Unix security has been considered somewhat coarse because one superuser, *root*, can do anything. Other users are lowly peons who endure the shackles root places upon them. The problem is, root doesn't have a wide variety of shackles on hand and can't individualize them very well. While there's some truth to this, a decent administrator can combine groups and permissions to handle almost any problem securely.

The root Password

Certain actions require absolute control of the system, including manipulating core system files such as the kernel, device drivers, and authentication systems. Such activities are designed to be performed by root.

To use the root password, you can either log in as root at a console login prompt or, if you're a member of the group wheel, log in as yourself and use the switch user command su(1). (We'll discuss groups later in this section.) I recommend su; it logs who uses it and can be used on a remote system. The command is very simple to use:

```
# su
Password:
#
```

Next, check your current user ID with the id(1) command:

```
# id
uid=0(root) gid=0(wheel) groups=0(wheel), 5(operator)
#
```

You now own the system—and I do mean *own* it. Consider every keystroke; carelessness can return your hard drive to the primordial state of unformatted empty wasteland. And share the root password sparingly, if at all, because anyone who has the root password can inflict unlimited damage on the system.

Remember, only the users in the group *wheel* can use the root password to become root through su(1). Anyone can use the root password at the system console, which is why physical protection of your system is vital. If you give the root password to a regular user who doesn't have physical access to the console, they can type `su` and enter the root password as many times as they want, and it still won't work.

This naturally leads to the question, "Who needs root access?" Much of the configuration discussed in this book requires use of the root password. Once you have the system running properly, you can greatly decrease or discontinue use of the root password. For those remaining tasks that absolutely require root privileges, I recommend the sudo package, and probably my book *Sudo Mastery* (Tilted Windmill Press, 2013). One of the simplest ways to reduce the need for root access is through the proper use of groups.

Groups of Users

Unix-like operating systems classify users into *groups*, each group consisting of people who perform similar administrative functions. A sysadmin can define a group called *webmasters*, add the accounts of the people editing web pages to that group, and set the privileges on the web-related files so that the members of that group can edit those files. She can also create a group called *email*, add the email administrators to that group, and set the permissions of mail-related files accordingly. Using groups in this manner is a powerful and oft-neglected tool for system management.

Any user can identify the groups she belongs to with id(1). The preceding example showed that the user root is in the groups wheel and operator. Root is a special user, however, and can do anything she pleases. Here's my

account, which is a little more realistic for an average user:

```
# id
uid=1001(mwlucas) gid=1001(mwlucas) groups=1001(mwlucas),0(wheel),68(dialer),1
0001(webmaster)
```

My UID is 1001, and my username is *mwlucas*. My GID, primary group ID, is 1001, and my primary group is named *mwlucas* as well. This is all pretty standard for the first user on a system, and even in later users, the only thing that changes is the numbers assigned to the account and primary group. More interesting is what other groups I'm assigned to: in addition to my primary group, I'm in the groups wheel, dialer, and webmaster. Wheel members may use the root password to become root, dialer members may use tip(1) without becoming root, and webmaster members can edit the web files on the local system. Each of these groups has special privileges on my system, and as a member of those groups, I inherit those privileges.

Group information is defined in */etc/group*.

/etc/group

The file */etc/group* contains all group information except for the user's primary group (which is defined with the user account in */etc/master.passwd*). Each line in */etc/group* contains four colon-delimited fields: the group name, the group password, the group ID number, and a list of members.

Here's a sample entry:

```
wheel:*:0:root,mwlucas,xistence
```

The group name is a human-friendly name for the group. This group is named *wheel*. Group names are arbitrary; you can call a group of users *lackeys* if you wish. Choose group names that give you an idea of what the groups are for; while you might remember that your lackeys may edit the company web page, will your coworkers understand that?

The second field, the group password, was a great idea that turned out to be a security nightmare. Modern Unix-like systems don't do anything with the group password, but the field remains because old programs expect to find something in this space. The asterisk is a placeholder to placate such software.

The third field gives the group's unique numeric group ID (GID). Many

programs use the GID rather than name to identify a group. The wheel group has a GID of 0, and the maximum GID is 65535.

Last is a comma-delimited list of all users in the group. The users *root*, *mwlucas*, and *xistence* are members of the group wheel.

Changing Group Memberships

If you want to add a user to a group, add his username to the end of the line for that group. For example, the wheel group is the list of users that may use the root password. Here, I add *rwatson* to the wheel group:

```
wheel:*:0:root,mwlucas,xistence,rwatson
```

Mind you, the odds of me convincing rwatson (leading security researcher and ex–FreeBSD Foundation President) to assume sysadmin duties on any of my systems range from negligible to nonexistent, but it's worth a try.

Creating Groups

To create a new group, you need only a name for the group and a group ID number. Technically, you don't even need a member for the group; some programs run as members of a group, and FreeBSD uses the group permissions to control those programs just as the users are controlled.

Traditionally, GIDs are assigned the next number up the list. GID is an arbitrary number between 0 and 65535. Generally speaking, GIDs below 1000 are reserved for operating system usage. Programs that need a dedicated group ID usually use one in this range. User accounts start numbering their GIDs at 1001 and go up. Some special groups might start numbering at 65535 and go down.

Using Groups to Avoid Root

In addition to being a security concern, the root password distribution policy can cause dissension in any organization. Many sysadmins refuse to share the root password with people who're responsible for maintaining part of the system but don't offer an alternative and thereby prevent people from doing their job. Other sysadmins hand out root to dang near anyone who wants it and then complain when the system becomes unstable. Both attitudes are

untenable in the long run. Personally, I don't want root on your system. While having root privileges can be convenient, a lack of responsibility when the system breaks is more convenient.

One common situation is where a junior sysadmin is responsible for a particular portion of the system. I've had many DNS administrators work under me;[2] these people don't ever install software, recompile the kernel, or perform other sysadmin tasks. They only answer emails, update zone files, and reload the named daemon. New sysadmins often believe they need root access to do this sort of work. Nope. You can use groups.

Establishing your own groups, consisting of people who perform similar administrative functions, lets you avoid distributing the root password and still allow people to do their work. In this section, we'll implement group-level access control over nameserver files. The same principles apply to any files you choose to protect. Mail and web configuration files are other popular choices for group-based management.

System Accounts

FreeBSD reserves some user account names for integrated programs. We discuss these unprivileged accounts in Chapter 19. For example, the nameserver runs under the user account bind and the group bind. If an intruder compromises the nameserver, she can access the system only with the privileges of the user bind.

Don't have users log in as these users. They're not set up as interactive accounts by design. What's more, do not allow the group of the system account user to own the files created for that function. Create a separate user and group to own program files. That way, our hypothetical intruder can't even edit the files used by the DNS server, further minimizing potential damage. If the program regularly updates the files (e.g., a database's backend storage), you must give the program access rights, but chances are that a human being doesn't ever need to edit that file. Similarly, there's no reason a database should be able to edit its own configuration file.

Administrative Group Creation

The simplest way to create a group that owns files is to employ adduser(8) to make a user that owns them and then to utilize that user's primary group as the group for the files. Because we already have a user called *bind*, we'll

create an administrative user *dns*. The username isn't important, but you should choose a name that everyone will recognize.

Give your administrative user a shell of *nologin*, which sets a shell of */sbin/nologin*. This prevents anyone from actually logging in as the administrative user.

If you want, you could specify a particular UID and GID for these sorts of users. I've been known to choose UID and GID numbers that resemble those used by their related service accounts. For example, the user bind has a UID and GID of 53. I could give the user dns a UID of 10053 to make it easily recognizable. At other times, I start numbering my administrative groups at 65535 and work my way down. It doesn't matter as long as I'm completely consistent within an organization.

Do not add this administrative user to any other groups. Under no circumstances add this user to a privileged group, such as wheel!

Every user needs a home directory. For an administrative user, a home directory of */nonexistent* works well. This user's files are elsewhere in the system, after all.

Lastly, let adduser(8) disable the account. While the shell prevents logins, an extra layer of defense won't hurt.

Now that you have an administrative user and a group, you can assign ownership of files to that user. A user and a group own every file. You can see existing file ownership and permissions with `ls -l`. (If you've forgotten how Unix permissions work, read ls(1) and chmod(1).) Many sysadmins pay close attention to file owners, somewhat less attention to worldwide permissions, and only glance at the group permissions.

```
# ls -l
total 3166
-rw-r-----  1 mwlucas  mwlucas    79552 Nov 11 17:58 rndc.key
-rw-rw-r--  1 mwlucas  mwlucas  3131606 Nov 11 17:58 mwl.io.db
```

Here, I've created two files. The first file, *rndc.key*, can be read and written by the user *mwlucas*. It can be read by anyone in the group *mwlucas*, but no one else can do anything with it. The file *mwl.io.db* can be read or written by the user *mwlucas* or anyone in the group *mwlucas*, but others can only read the file. If you're in the group *mwlucas*, you can edit the file *mwl.io.db* without becoming root.

Change a file's owner and group with chown(1). You must know the name of the user and group whose ownership you want to change. In this case, we want to change both files to be owned by the user dns and the group dns.

```
# chown dns:dns rndc.key
# chown dns:dns mwl.io.db
# ls -l
total 3166
-rw-r-----  1 dns  dns    79552 Nov 11 17:58 rndc.key
-rw-rw-r--  1 dns  dns  3131606 Nov 11 17:58 mwl.io.db
```

These files are now owned by the user dns and the group dns. Anyone who is in the group dns can edit *mwl.io.db* without using the root password. Finally, this file can be read by the user bind, who runs the nameserver. Add your DNS administrators to the dns group in */etc/group*, and abruptly they can do their jobs.

The DNS administrators might think they need the root password for restarting the nameserver program itself. However, this is easily managed with rndc(8). Other tasks can be managed with cron jobs or with the add-on program sudo(8).

If you don't want an administrative user but only a group, use vigr(8) to edit */etc/group*.

Interesting Default Groups

FreeBSD ships with several default groups. Most are used by the system and aren't of huge concern to a sysadmin—you should know that they're there, but that's different than working with them on a day-to-day basis. In Table 9-1, I present for your amusement and edification the most useful, interesting, and curious of the default groups. Adding your own groups simplifies system administration, but the groups listed here are available on every FreeBSD system.

Table 9-1: FreeBSD System Groups

Group name	Purpose
audit	Users who can access audit(8) information
authpf	Users who can authenticate to the PF packet filter

bin	Group owner of general system programs
bind	Group for the BIND DNS server software
daemon	Used by various system services, such as the printing system
_dhcp	DHCP client operations
dialer	Users who can access serial ports; useful for modems and tip(1)
games	Owner of game files
guest	System guests (almost never used)
hast	Files used by hastd(8)
kmem	Programs that can access kernel memory, such as fstat(1), netstat(1), and so on
mail	Owner of the mail system
mailnull	Default group for sendmail(8) or other mail server
man	Owner of uncompressed man pages
network	Owner of network programs like ppp(8)
news	Owner of the Usenet News software (probably not installed)
nobody	Primary group for unprivileged user nobody, intended for use by NFS
nogroup	Group with no privileges, intended for use by NFS
operator	Users that can access drives, generally for backup purposes
_pflogd	Group for PF logging
proxy	Group for FTP proxy in PF packet filter
smmsp	Group for Sendmail submissions
sshd	Owner of the SSH server (see Chapter 20)
staff	System administrators (from BSD's college roots, when users were staff, faculty, or students)
sys	Another system group
tty	Programs that can write to terminals, like wall(1)
unbound	Files and programs related to the unbound(8) DNS server
uucp	Group for programs related to the Unix-to-Unix Copy Protocol

video	Group that can access DRM and DRI video devices
wheel	Users who may use the root password
www	Web server programs (not files)
_ypldap	Files needed by the LDAP-backed YP server ypldap(8)

I know very few people using either internet news or UUCP, and you might think you could reuse those groups for other purposes. You're really better off creating a new group than risking confusion later, however. Group ID numbers are not in short supply.

Tweaking User Security

You prevent any single user from utilizing too much memory, processor time, or other system resources by setting limits on the account. Now that even small computers have very fast processors and lots of memory, these limits aren't as important, but it's still very useful in systems with dozens or hundreds of users. You can also control where users may log in from.

Restricting Login Ability

FreeBSD checks */etc/login.access* every time a user tries to log in. If *login.access* contains rules that forbid logins from that user, the login attempt fails immediately. This file has no rules by default, meaning that anyone who provides a valid username and password has no restrictions.

The */etc/login.access* file has three colon-delimited fields. The first either grants (+) or denies (-) the right to log in; the second is a list of users or groups; and the third is a list of connection sources. You can use an ALL or ALL EXCEPT syntax, which allows you to make simple but expressive rules. Rules are checked on a first-fit basis. When login(1) finds a rule where the user and the connection source match, the connection is immediately accepted or rejected, making rule order vital. The default is to allow logins.

For example, to allow only members of the wheel group to log in from the system console, you might try this rule:

```
+:wheel:console
```

The problem with this rule, however, is that it doesn't actually deny users login privileges. Since the default is to accept logins, and since all this rule does is explicitly grant login privileges to the users in the wheel group, nothing changes. Bert certainly isn't in the wheel group, but if he tries to log in, no rule denies him access.

You could try two rules like this:

```
+:wheel: console
-:ALL:console
```

This set of rules would achieve the desired effect but is longer than you need. Use ALL EXCEPT instead.

```
-:ALL EXCEPT wheel: console
```

This rule rejects unwanted logins most quickly and runs less risk of administrator error. As a rule, it's best to build *login.access* lists by rejecting logins, rather than permitting them. FreeBSD immediately rejects nonwheel users at the console upon hitting this rule.

Change the default from "allow access" to "deny access" by adding a final rule.

```
-:ALL:ALL
```

Any login request that doesn't match an earlier permit rule gets denied.

The last field in *login.access*, the connection source, can use hostnames, host addresses, network numbers, domain names, or the special values LOCAL and ALL. Let's see how they work.

Hostnames

Hostnames rely upon DNS or the hosts file. If you suspect that your nameserver might suffer an intrusion or attack, avoid hostnames; intruders can give a hostname any IP address that they like and fool your system into accepting the connection, and a nameserver failure could lock you out completely. Still, it's possible to use a rule like this:

```
-:ALL EXCEPT wheel:fileserver.mycompany.com
```

Users in the wheel group can log in from the fileserver, but nobody else

can.

Host Addresses and Networks

Host addresses work like hostnames, but they're immune to DNS failures or spoofing.

```
-:ALL EXCEPT wheel:203.0.113.5
```

A network number is a truncated IP address, like this:

```
-:ALL EXCEPT wheel:203.0.113.
```

This network number allows anyone in the wheel group to log in from a machine whose IP address begins with 203.0.113 and denies everyone else access from those IP addresses.

LOCAL

The most complicated location is `LOCAL`, which matches any hostname without a dot in it (generally, only hosts in the local domain). For example, *www.mwl.io* thinks that any machine in the domain *mwl.io* matches `LOCAL`. DNS spoofing can easily evade this filter. Although my desktop claims that it has a hostname of *storm.mwl.io*, its IP address has reverse DNS that claims it's somewhere in my cable modem provider's network. The host *www.mwl.io* thinks that my desktop isn't in the same domain and hence isn't local. As such, I can't use the `LOCAL` verification method.

Similarly, anyone who owns a block of IP addresses can give their addresses any desired reverse DNS. The `LOCAL` restriction is best avoided.

ALL and ALL EXCEPT

`ALL` matches everything, and `ALL EXCEPT` matches everything but what you specify. These are the most useful connection sources, in my opinion. For example, if you had a highly secure machine only accessible from a couple of management workstations, you could have a rule like this:

```
-:ALL EXCEPT wheel:ALL EXCEPT 203.0.113.128 203.0.113.44
```

Tie It All Together

The point of these rules is to build a login policy that matches your real-world policies. If you provide generic services but only allow your system administrators to log on remotely, a one-line *login.access* prevents any other users from logging in:

```
-:ALL EXCEPT wheel:ALL
```

This is great if you can live with a restriction this tight. On the other hand, I've worked at several internet service providers that used FreeBSD to provide client services. Lowly customers weren't allowed to log onto the servers unless they had a shell account. System administrators could log in remotely, as could the DNS and web teams (members of the dns and webmasters groups). Only sysadmins could log onto the console, however.

```
-:ALL EXCEPT wheel:console
-:ALL EXCEPT wheel dns webmasters:ALL
```

Set this up in *login.access* once, and let group membership control all of your remote logins forever after.

Restricting System Usage

You can provide more specific controls with login classes. Login classes, managed through */etc/login.conf*, define the resources and information provided for users. Each user is assigned a class, and each class has limits on the system resources available. When you change the limits on a class, all users get the new limits when they next log in. Set a user's class when creating the user account, or change it later with chpass(1).

Class Definitions

The default *login.conf* starts with the default class, the class used by accounts without any other class. This class gives the user basically unlimited access to system resources and is suitable for application servers with a limited number of users. If this meets your needs, don't adjust the file at all.

Each class definition consists of a series of variable assignments that define the user's resource limits, accounting, and environment. Each variable assignment in the class definition begins and ends with a colon. The backslash character is a continuation character to indicate that the class

continues on the next line, which makes the file more readable. Here's a sample of the beginning of one class:

```
❶default:\
      ❷:passwd_format=❸sha512:\
        :copyright=/etc/COPYRIGHT:\
        :welcome=/etc/motd:\
--snip--
```

This class is called `default` ❶. I've shown three of the dozens of variables in this class. The variable `passwd_format` ❷, for example, is set to `sha512` ❸. These variable assignments and the class name describe the class, and you can change the user's experience on the system by assigning the user to another class.

Some of *login.conf* 's variables don't have a value and instead change account behavior just by being present. For example, the `requirehome` variable takes effect just by being included in the class. If this value is present, the user must have a valid home directory.

```
      :requirehome:\
```

After editing *login.conf*, you must update the login database to make the changes take effect.

```
# cap_mkdb /etc/login.conf
```

This rebuilds the database file */etc/login.conf.db* that's used for fast lookups, much like */etc/spwd.db*.

The default */etc/login.conf* includes several example classes of users. If you want an idea of what sort of restrictions to put on users for various situations, check those examples. The following section offers ideas about what can be set in a login class. For a complete listing of supported settings in your version of FreeBSD, read man login.conf(5).

Resource Limits

Resource limits allow you to control how much of the system any one user can monopolize at any one time. If you have several hundred users logged in to one machine and one of those users decides to compile LibreOffice, that person will consume far more than his fair share of processor time, memory,

and I/O. By limiting the resources one user can monopolize, you can make the system more responsive for all users.

Table 9-2 defines the resource-limiting *login.conf* variables.

Table 9-2: Some *login.conf* Variables for Limiting Resource Use

Variable	Description
`cputime`	The maximum CPU time any one process may use
`filesize`	The maximum size of any one file
`datasize`	The maximum memory size of data that can be consumed by one process
`stacksize`	The maximum amount of stack memory usable by a process
`coredumpsize`	The maximum size of a core dump
`memoryuse`	The maximum amount of memory a process can lock
`maxproc`	The maximum number of processes the user can have running
`openfiles`	The maximum number of open files per process
`Sbsize`	The maximum socket buffer size a user's application can set

Note that resource limits are frequently set per process. If you permit each process 200MB of RAM and allow each user 40 processes, you've just allowed each user about 8GB of memory. Perhaps your system has a lot of memory, but does it really have that much?

Current and Maximum Resource Limits

In addition to the limits listed previously, you can specify current and maximum resource limits. *Current* limits are advisory, and the user can override them at will. This works well on a cooperative system, where multiple users willingly share resources but you want to notify those users who exceed the standard resource allocation. Many users want to be good citizens, and readily cooperate when they're told they're pushing their limits.[3] Users cannot exceed *maximum* limits.

If you don't specify a limit as current or maximum, FreeBSD treats it as a maximum limit.

To specify a current limit, add `-cur` to the variable name. To make a maximum limit, add `-max`. For example, to set a current and a maximum limit on the number of processes the user can have, use this input:

```
--snip--
:maxproc-cur: 30:\
:maxproc-max: 60:\
--snip--
```

One counterpart to resource limits is resource accounting. These days, accounting isn't as important as it was when today's inexpensive computers would cost millions of dollars, so we won't discuss it in this book. It's more important to restrict a single user from consuming your system than to bill for every CPU cycle someone uses. You should know that the capability exists, however.

If you need more complicated resource restrictions, investigate rctl(8).

Class Environment

You can also define environment settings in */etc/login.conf*. This can work better than setting them in the default *.cshrc* or *.profile* because *login.conf* settings affect all user accounts immediately upon their next login. Some shells, such as zsh(1), don't read either of these configuration files, so using a class environment sets the proper environment variables for those users.

All of the environment fields recognize two special characters. A tilde (~) represents the user's home directory, while a cash symbol ($) represents the username. Here are a few examples from the default class that illustrate this:

```
        :setenv=MAIL=❶/var/mail/$,BLOCKSIZE=K,FTP_PASSIVE_MODE=YES:\
        :path=/sbin /bin /usr/sbin /usr/bin /usr/games /usr/local/sbin /usr/
local/bin /usr/X11R6/bin ❷~/bin:\
```

By using the `$` character, the environment variable `MAIL` is set to */var/mail/<username>* ❶. Similarly, the last directory in the `PATH` variable is the *bin* subdirectory in the user's home directory ❷.

Table 9-3 lists some common *login.conf* environment settings.

Table 9-3: Common *login.conf* Environment Settings

Variable	Description
[illegible]	[illegible]

`hushlogin`	If present, no system information is given out during login.
`ignorenologin`	If present, these users can log in even when */var/run/nologin* exists.
`manpath`	A list of directories for the `$MANPATH` environment variable.
`nologin`	If present, the user cannot log in.
`path`	A list of directories for the `$PATH` environment variable.
`priority`	Priority (nice) for the user's processes (see Chapter 21).
`requirehome`	User must have a valid home directory to log in.
`setenv`	A comma-separated list of environment variables and their values.
`shell`	The full path of a shell to be executed upon login. This overrides the shell in */etc/master.passwd*. The user's `$SHELL`, however, contains the shell from the password file, resulting in an inconsistent environment. Playing games with this is an excellent way to annoy your users.
`term`	The default terminal type. Just about anything that tries to set a terminal type overrides this.
`timezone`	The default value of the `$TZ` environment variable.
`umask`	Initial umask setting; should always start with 0, see builtin(1).
`welcome`	Path to the login welcome message, usually */etc/motd*.

Remember, changes to a class affect all users in that class. If a user needs a change from the class settings, you'll need to change their class.

Password and Login Control

Unlike the environment settings, many of which can be set in places other than the login class, most login and authentication options can be controlled only from the login class. Here are some common authentication options:

`passwd_format`

This option sets the cryptographic hash used to store passwords in */etc/master.passwd*. The default is `sha512`, for SHA512 hashing. Other

permissible options are `des` (DES), `blf` (Blowfish), `md5`, and `sha256` (SHA256). DES and Blowfish are most useful when you want to share password files between different Unix-like operating systems, but are very weak. SHA256 is for compatibility with older password files, from before SHA512 was the default.

`mixpasswordcase`
If present, FreeBSD complains if the user changes his password to an all-lowercase word. Despite the name, all-uppercase passwords satisfy this option.

`host.allow`, `host.deny`
These values let users in this class use rlogin and rsh. Avoid them like the fuzzy green meat your creepy roommate tried to feed you that one time.

`times.allow`
This option allows you to schedule when users may log in with a comma-delimited list of days and times. Days are given as the first two letters of the day's name (Su, Mo, Tu, We, Th, Fr, and Sa). Time is in standard 24-hour format. For example, if a user can log in only on Wednesdays between 8 AM and 5 PM, you'd use this entry:

```
        :times.allow=We8-17:\
```

`times.deny`
This option allows you to specify a time window when the user can't log in. Note that this does not kick off users who are already logged in. The format is the same as for `times.allow`. If `times.allow` and `times.deny` overlap, `times.deny` takes precedence.

You can't make that overworking developer go home, but you can keep him from opening another terminal window.

File Flags

All Unix-like operating systems have the same filesystem permissions, assigning read, write, and execute privileges for a file to the file's owner, its group, and all others. FreeBSD extends the permissions scheme with *file*

flags, which work with permissions to enhance your system's security.

Many flags have different effects depending on the system securelevel, which we'll cover in the next section. Understanding securelevels requires an understanding of file flags, while file flags rely on securelevels. For the moment, just nod and smile when you encounter a mention of securelevels; all becomes clear in the next few pages.

A few file flags are useful only in specialized cases. We'll look only at the most commonly useful flags. See chflags(1) for the complete list.

Many flags have multiple names; while only one name appears in ls(1) output, you can use any name at the command line. These alternate names exist because people got tired of getting an error when they typed `sappend` instead of `sappnd`. Here, I show the flag's primary name first and then the user-friendly aliases.

`sappnd`, `sappend`

This system-level, append-only flag can be set only by root. Files with this flag can be added to but can't be removed or otherwise edited. This flag is particularly useful for log files. Setting `sappnd` on a user's *.history* file can be interesting if the account is compromised. Since a common intruder tactic is to remove *.history* or symlink it to */dev/null* so that the admin can't see what happened, `sappnd` ensures that script kiddies cannot cover their tracks in this manner. It's almost funny to review the record of someone trying to remove a `sappnd` file; you can almost see the attacker's frustration grow as he tries various methods.[4] This flag can't be removed when the system is running at securelevel 1 or higher.

`schg`

Only root can set the system-level immutable flag. Files with this flag set can't be changed in any way. They can't be edited, moved, replaced, or overwritten. Basically, the filesystem itself prevents all attempts to alter this file. The flag can't be removed when the system is running at securelevel 1 or greater.

`sunlnk`

Only root can set the system-level undeletable flag on a file. The file can be edited or altered, but it can't be deleted. This isn't as secure as the previous two flags because if a file can be edited, it can be emptied. It's

still useful for certain circumstances, however. I've used it when a program insisted on deleting its own log files upon a crash. It's not generally useful to set on any standard system files, however. This flag can't be removed when the system is running at securelevel 1 or higher.

uappnd

This user-level, append-only flag can be set only by the file owner or root. Like the system-level append-only flag `sappnd`, a file with this flag set can be added to but not otherwise edited or removed. This flag is most useful for logs from personal programs and the like; it's primarily a means to let users prevent accidental removal of their own files. The owner or root can remove this flag.

uchg

This user-level, immutable flag can be set only by the owner or root. Like the `schg` flag, this immutable flag prevents anyone from changing the file. Again, root can override this, and it can be disabled by the user at any securelevel. This flag helps prevent mistakes, but it's not a way to secure your system.

uunlnk

This user-level, undeletable flag can be set only by the owner or root. A file with this flag set can't be deleted by the owner. Root can override that, and the user can turn this flag off at any time, making this mostly useless.

Setting and Viewing File Flags

Set flags with chflags(1). For example, to be sure that nothing replaces a server's kernel, you could do this:

```
# chflags schg /boot/kernel/kernel
```

You'd need to remove this flag to perform system updates.

You can recursively change the flags on an entire directory tree with the `-R` flag. For example, to make all of */bin* directory immutable, run this command:

```
# chflags -R schg /bin
```

And boom! Your basic system binaries can't be changed.

To see what flags are set on a file, use `ls -lo`.

```
# ls -lo log
-rw-r--r--  1 mwlucas  mwlucas  sappnd 0 Nov 12 12:37 log
```

The `sappnd` entry tells us that the system append-only flag is set on this log. For comparison, if a file has no flags set, it looks like this:

```
# ls -lo log
-rw-r--r--  1 mwlucas  mwlucas  - 0 Nov 12 12:37 log
```

The hyphen in place of the flag name tells us that no flag has been set.

An out-of-the-box FreeBSD install doesn't have many files marked with flags, but you can flag anything you want. On one system that I fully expected to be hacked, I went berserk with `chflags -R schg` in various system directories to prevent anyone from replacing system binaries with trojaned versions. It might not stop an attacker from getting in, but imagining their frustration improved my mood.

To remove a file flag, use `chflags` and a `no` in front of the flag name. For example, to unset the `schg` flag on your kernel, enter this command:

```
# chflags noschg /boot/kernel/kernel
```

That said, you must be running at securelevel –1 to unset many flags. So, without further ado, let's discuss securelevels and what they mean to you.

Securelevels

Securelevels are kernel settings that change basic system behavior to disallow certain actions. The kernel behaves slightly differently as you raise the securelevel. For example, at low securelevels, file flags can be removed. A file might be flagged immutable—but you can remove the flag, edit the file, and reflag it. When you increase the securelevel, the file flag can't be removed. Similar changes take place in other parts of the system. Taken as a whole, the behavior changes that result from increased securelevels either frustrate or stop an intruder. Enable securelevels at boot with the *rc.conf* option `kern_securelevel_enable="YES"`.

Securelevels complicate system maintenance by imposing restrictions on your behavior. After all, many system administration tasks are also things intruders might do to cover their tracks. For example, at certain securelevels, you can't format or mount new hard drives while the system is running. On the other hand, securelevels hamper intruders even more than they hamper you.

Securelevel Definitions

Securelevels come in 5 degrees: –1, 0, 1, 2, and 3, with –1 being the lowest and 3 the highest. Once you enable securelevels with the `kern_securelevel_enable` *rc.conf* option, you can set the securelevel at boot with the `kern_securelevel` *rc.conf* variable. You can raise the securelevel at any time, not just at boot, but you can't lower it without rebooting into single-user mode. After all, if you could lower the securelevel at any time, so could your intruder!

The effects of each securelevel vary depending on your FreeBSD release. To get the latest information, read security(7).

Securelevel –1

The default provides no additional kernel security whatsoever. If you're learning FreeBSD and are frequently changing your configuration, remain at securelevel –1 and use the built-in file permissions and other Unix safeguards for security. Flags like `sappnd` and `schg` will work, but chflags(1) can easily remove the flags.

Securelevel 0

Securelevel 0 is used only during booting and offers no special features over securelevel –1. When the system reaches multiuser mode, however, the securelevel is automatically raised to 1. Setting `kern_securelevel=0` in */etc/rc.conf* is effectively the same as setting `kern_securelevel=1`. Securelevel 0 is helpful if you have startup scripts that perform actions prohibited by securelevel 1.

Securelevel 1

At securelevel 1, the basic secure mode, things become interesting:

- System-level file flags may not be turned off.
- You can't load or unload kernel modules (see Chapter 6).
- Programs can't write directly to system memory via either */dev/mem* or */dev/kmem*.
- Nothing can access */dev/io*.
- You can't enter the kernel debugger with the `debug.kdb.enter` sysctl.
- You can't panic the system with the `debug.kdb.panic` sysctl.
- Mounted disks can't be written to directly. (You can write files to disk; you just can't address the raw disk devices.)

The most obvious effect of securelevel 1 for ordinary users is that the BSD-specific filesystem flags can't be altered. If a file is marked system-level immutable, and you want to replace it, too bad.

Securelevel 2

Securelevel 2 has all the behaviors of securelevel 1, with two additions:

- Disks can't be opened for writing, whether mounted or not.
- You can't alter system time by more than one second.

Both of these seem irrelevant to new sysadmins, but they provide important security protections. Although Unix provides handy tools, like text editors to write files, it's also possible to bypass both those tools and the actual filesystem to access the underlying ones and zeros on the hard drive. Poking at the hard drive lets you change any file regardless of the file permissions. The only time this commonly happens is when you install a new hard drive and must create a filesystem on it. Normally, only the root user can write directly to the disk in this manner. At securelevel 2, even root can't use newfs(8), zpool(8), and so on.

Similarly, another old hacker trick is to change the system time, edit a file, and change the time back. That way, when the administrator looks for files that might be causing trouble, the tampered file appears to have been untouched for months or years and hence doesn't seem an obvious source of concern.

Securelevel 3

Securelevel 3 is the *network secure mode*. In addition to the settings of securelevels 1 and 2, you can't adjust packet filter rules. The firewall on your host is immutable. If you have a system with packet filtering or bandwidth management enabled and those rules are well tuned and unlikely to change, you can use securelevel 3.

Which Securelevel Do You Need?

The securelevel appropriate for your environment depends entirely upon your situation. If you've just put a FreeBSD machine into production and are still fine-tuning it, leave the securelevel at –1. Once your system is tuned, however, you can raise the securelevel. Most production systems run just fine at securelevel 2.

If you use one of FreeBSD's packet filtering or firewall packages, securelevel 3 might look tempting. Be very sure of your firewall rules before you enable this, however! Securelevel 3 makes it impossible to change your firewall without disrupting your connection. Are you 100 percent certain that none of your customers will ever call in to say, "Here's a check. Now give me more servers!"?

What Won't Securelevels and File Flags Accomplish?

Consider a case where someone compromises a CGI script on your web server, uses that to bootstrap into a shell, and then uses the shell to bootstrap himself into root access.

If you've set the securelevel accordingly, perhaps this attacker will become frustrated because not only can't she replace your kernel with her specially compiled one, she also can't even load a kernel module. No problem—she can still replace assorted system programs with trojaned versions so that the next time you log in, your new version of login(1) sends your password to an anonymous web-based mailbox or to an internet newsgroup.

So, to protect your key files, you run around doing `chflags schg -R /bin/*`, `chflags schg -R /usr/lib`, and so on. Fine. If you forget one file—say, something obscure like */etc/rc.bsdextended* —your intruder can edit that file to include `chflags -R noschg /`. She can then reboot your system late at night when you might not notice. How often do you sit down and exhaustively audit your

/etc/rc files?

You think that your system is safe, with every file completely protected. But what about */usr/local/etc/rc.d*, the local program startup directory? The system boot process tries to execute any executable file in this directory that contains a line starting with `#PROVIDE:` (see Chapter 17 for why). Your intruder could therefore do a lot of damage by placing a simple shell script there. After all, */etc/rc* raises the securelevel at the end of the boot process. What if she were to create a shell script that kills the running */etc/rc* before it could raise the securelevel and then she turned around and ran his own */var/.hidden/rc.rootkit* to finish bringing the system up?

Of course, these are only a couple of possibilities. There are others, limited only by your intruder's creativity. Remember that system security is a thorny problem with no easy solution. Once intruders have a command prompt, it's you against them. And if they're any good, you won't even notice the penetration until it's too late. By following good computing practices and keeping your system up to date, you can stop them from intruding in the first place. Do not allow securelevels to make you lazy!

Living with Securelevels

If you've been liberal with the `schg` flag, you'll soon find that you can't upgrade or patch your system conveniently. The fact is, the same conditions that make intruders' lives difficult can make yours a living hell if you don't know how to work with them.

If you've frozen */etc/rc.conf* with `schg`, you must lower the securelevel to change the programs running on your system. Of course, the securelevel setting is in that file, so in order to edit it, you must take control of the system before */etc/rc* runs. That means you must boot into single-user mode (as discussed in Chapter 4), mount your filesystems, run `chflags noschg` on the files in question, and continue booting. You can even entirely disable securelevels in */etc/rc.conf* and work normally while the system runs or add commands to */etc/rc.local* so they take effect before the securelevel is set. You'll restore service more quickly that way but lose the protections of the file flags.

After you've finished maintenance, you can raise (but not lower) the securelevel by changing the `kern.securelevel` sysctl to your desired securelevel.

```
# sysctl kern.securelevel=3
```

Now that you can control file changes, let's consider controlling access to your system from the network.

Network Targets

Intruders normally break into applications that listen to the network, not the operating system itself. An operating system may or may not help defend a piece of software against network attacks, but the intrusion itself starts with the application. One way to reduce the number of attacks that can be carried out against your server is to identify all of the programs that are listening to the network and disable any that aren't strictly necessary. FreeBSD provides sockstat(1) as an easy way to identify programs that are listening to the network.

We cover sockstat in detail in Chapter 8; running `sockstat -4` shows all open IPv4 TCP/IP ports. Every network port you have open is a potential weakness and a potential target. Shut down unnecessary network services and secure those you must offer.

It's a good idea to regularly review which ports are open on your systems because you might learn something that surprises you. You might find that some piece of software you've installed has a network component that you weren't aware of and that it's been quietly listening to the network.

Once you know what's running, how do you turn off what you don't need? The best way to close these ports is to not start the programs that run them. Network daemons generally start in one of two places: */etc/rc.conf* or a startup script in */etc/rc.d*. Programs that are integrated with the main FreeBSD system, such as sendmail(8), sshd(8), and rpcbind(8), have flags in *rc.conf* to enable or disable them, as do many add-on programs. See Chapter 4 for details on enabling and disabling programs at startup.

WORKSTATION VS. SERVER SECURITY

Many companies I've seen have tightly secured servers but pay little attention to workstation security. A prospective

intruder doesn't care whether a system is a server or a workstation, however. Many servers and firewalls have special rules for the sysadmin's workstation. An intruder will happily penetrate a workstation and try to leverage that into server access. While server security is key, don't neglect workstations—especially *your* workstation!

Network probes are strange in that you really don't know when someone pokes at your hosts. To see how much of this goes on, set `log_in_vain` to 1 in */etc/rc.conf* on one of your public servers. This tells the kernel to log all connection attempts to closed ports. When someone checks your host for a nonexistent telnet, Squid, or database listener, the kernel logs the attempt to */var/log/messages*. Watch that log only long enough to realize clear down to your marrow that the whole internet really is out to get you—and then disable `log_in_vain`.

Putting It All Together

Once you have only the necessary network ports open and you know which programs are using those ports, you know which programs you must be most concerned about securing. If the FreeBSD security team sends out an announcement of a problem with a service you don't run, you can safely delay implementing a fix until your next maintenance window. If, however, the security team announces a hole in programs you're using, you know you have to implement a fix as soon as possible. If they announce a serious security problem with a piece of network software you're using, you know you must act quickly. Simply being able to respond intelligently and quickly to real risks helps protect you against most intruders. Tools such as file flags and securelevels minimize the damage successful intruders can do. Finally, using groups to restrict your own system administrators to particular sections of the system can protect your computers from both accidental and deliberate damage.

Let's shift gears now and talk storage.

10
DISKS, PARTITIONING, AND GEOM

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

First, let's discuss the most important thing to remember about storage devices.

Disks Lie

Once upon a time, a sysadmin could make decisions about a disk based on the information it provided. You could plug in a hard drive and query it for the number of platters, cylinders, sectors, and more. Those days are long, long past. Yes, you can perform the same query and get an answer, but those answers don't reflect any reality. Today, a disk is a magic box that regurgitates data on request. Some of those magic boxes contain spinning platters. Others lack moving parts. The magic boxes provide numbered sectors for storing bits and bytes. The relationship between those numbers and the contents of the box? That's magic: inscrutable and unknowable.

In previous books, including earlier editions of this one, I've discussed the importance of proper data placement on the disk, but all of that knowledge is completely obsolete. If you still retain any of that knowledge, discard it in favor of something more useful, like the complete biographies of all the

actors who appeared in any role in classic *Doctor Who*.

As far as disk design goes, the only thing you need to know about is *logical block addressing (LBA)*. Each sector on a disk is assigned a number. Filesystems call disk sectors by number. That's it. Anything beneath LBA is pure guesswork on your part.

Unfortunately, disks now have a new category of lies they tell: sector size.

Up through the 1990s, disk sector sizes varied from 128 bytes to 2KB. Even the original IBM PC could understand different sector sizes on floppy disks.

In the early 2000s, though, manufacturers settled on 512-byte sectors. Today's hard drives are much larger, and the files are similarly larger. In the last few years, the 512-byte sectors have mostly been replaced with 4,096-byte sectors, called *4K drives*. This sector size makes more sense for the type of data we store today.

The problem is, operating systems like Windows XP know that a disk sector always has been, and always will be, 512 bytes. These operating systems won't tolerate hard drives that reported having 4KB sectors because everybody knows there's no such thing. If you manufacture 4K drives, what do you do?

The same thing you always do.

You teach the hard drive to lie.

Best of all, different 4K drives lie in different ways. If the OS asks a drive its sector size, most drives state that they have 512-byte sectors. Drives that claim to have both 512-byte and 4KB sectors are probably 4K drives, struggling to tell the truth. Very few admit to having solely 4KB sectors. To complicate matters even more, some solid state drives have sectors as large as 8KB or 16KB, or they support multiple sector sizes.

Both of FreeBSD's main filesystems must know the sector size of the underlying disk and the logical block address of that sector. If you use the wrong sector size on your disk, performance suffers. I could go into long detailed discussions of how this happens, but to keep it simple, always align partitions on even megabyte boundaries. You might waste a few bytes here and there, but that's trivial compared to the truly appalling performance you'll get from having a filesystem misaligned with the disk.

Device Nodes

We touched briefly on device nodes in Chapter 4, but let's consider them in more detail here. Device nodes are special files that represent a hardware device or an operating system feature. They're used as logical interfaces to provide features to user programs. By using a command on a device node, sending information to a device node, or reading data from a device node, you're telling the kernel to perform an action. If the device node represents a physical device, you're acting on that device. These actions can be very different for different devices—writing data to disk is very different than writing data to a sound card. While you can expose device nodes anywhere, the standard device nodes exist in */dev*.

Before you can work with a disk or disk partition, you must know its device name. FreeBSD disk device nodes come from the names of the device driver for that type of hardware. Device driver names, in turn, often come from the type of device and not the device's role or function.

Table 10-1 shows the most common disk device nodes.

Table 10-1: Storage Device Nodes and Types

Device node	Man page	Description
*/dev/ada**	ada(4)	ATA-style direct access disks (SATA, IDE, etc.)
*/dev/cd**	cd(4)	Optical media drives (CD, Blu-Ray, etc.)
*/dev/da**	da(4)	SCSI-style direct access disks (USB storage, SAS, etc.)
*/dev/md**	md(4)	Memory disks
*/dev/mmcsd**	mmcsd(4)	MMC and SD memory cards
*/dev/nvd**	nvd(4)	NVM express drives
*/dev/vtbd**	virtio_blk(4)	Virtio-based virtual machine disk
*/dev/xbd**	xen(4)	Xen virtual disks

Many RAID controllers present their RAID containers as SCSI devices, so they show up as */dev/da* device nodes. Others present their disks as "SCSI plus special vendor topping," so they get special device node names such as

/dev/raid (ATA RAID), */dev/mfid* (certain LSI MegaRAID cards), and so on. Check the man page for your RAID controller to see the device node it presents.

The Common Access Method

The *Common Access Method (CAM)* is a standardized device driver architecture originally written to support the complex command set of 20th-century SCSI-2 disks. The idea was that standardizing based on this architecture would simplify writing device drivers. Only FreeBSD and DEC OSF/1 actually shipped with CAM, however, and each filled in the specification's gaps differently.

FreeBSD 9 and later consolidates management of all physical disks that support CAM in the CAM interface. Use camcontrol(8) to gather information from disks and issue commands to them. The camcontrol(8) command has a variety of subcommands that let you issue instructions to hard drives.

What Disks Do You Have?

To identify a host's storage devices, you can trawl */var/run/dmesg.boot* looking for disk device nodes or see which filesystems are mounted and backtrack from there. But the easiest way to identify your storage is to have camcontrol(8) ask the CAM system what disks it sees. Let's look at one of my test systems:

```
# camcontrol devlist
<ATA WDC WD1003FBYZ-0 1V03>        at scbus0 target 0 lun 0 (pass0,da0)
<ATA WDC WD1003FBYZ-0 1V03>        at scbus0 target 1 lun 0 (pass1,da1)
<ATA WDC WD1003FBYZ-0 1V03>        at scbus0 target 2 lun 0 (pass2,da2)
<ATA WDC WD1003FBYZ-0 1V03>        at scbus0 target 3 lun 0 (pass3,da3)
```

This output is broken up into three fields. The first gives the name of the device, as reported by the device itself. This is usually a vendor and the vendor's model number.

The second section gives SCSI connection information. These drives aren't actually SCSI drives—they're SATA connections managed via CAM. But you now know which disk devices are plugged into which port on the

SATA controller.

Finally, in parentheses, we have the SCSI device and what we probably want, the storage device node. This host has four disks, named da0, da1, da2, and da3.

Non-CAM Devices

Generally speaking, everything except proprietary RAID controllers and virtual disks support CAM.

RAID controllers have usually embraced and extended the CAM protocol, for what the manufacturer thought was a good reason at the time. A protocol written in the early 1990s wasn't sufficient for a 2010 RAID controller. These controllers usually have their own control programs. The RAID containers show up in `devlist` and some other camcontrol(8) subcommands.

Similarly, virtual disks don't respond to CAM commands. There's no disk to command there—you're just writing blocks to a file. You can view the disk with `camcontrol devlist`, but that's about it.

For most applications, I recommend using FreeBSD's RAIDZ or GEOM RAID, rather than a hardware RAID controller.

The GEOM Storage Architecture

FreeBSD has an incredibly flexible storage infrastructure system called *GEOM* (short for "disk geometry"). GEOM lives between device driver nodes and the underlying hardware, handling data exchanged between them. From this position, GEOM can arbitrarily transform input/output requests.

DEVICE CONTROL PROGRAMS

Some storage devices have special features that aren't addressed in the generic CAM framework. RAID controllers often have vendor-specific features, and FreeBSD includes many small programs to individually manage these controllers. You'll find programs like mfiutil(8) and mptutil(8)

for older LSI controllers, mpsutil(8) for newer LSI controllers, and so on. If you have a nonvolatile memory express (NVMe) drive, check out nvmecontrol(8).

GEOM is built out of kernel modules, called *GEOM classes*, that let you perform specific types of transformation or management. Disks have a GEOM class that lets the kernel put data on the disk. But if you want to encrypt your disks, that's a GEOM class. Software-based RAID? A GEOM class. FreeBSD implements all storage modifications as GEOM classes.

GEOM classes are *stackable*. They use the output of one class as the input for another. You want to encrypt your hard drive and then mirror it to another hard drive? Sure! Stack an encryption module on top of your hard drive and then stack the drive-mirroring module on top of that. You want to mirror that drive across the network? Add that GEOM class to the stack. This flexible modularity makes GEOM one of FreeBSD's most powerful features.

GEOM Autoconfiguration

When FreeBSD finds a new storage device, either at boot or when you plug a new drive in, the GEOM subsystem checks the device for known formats, like a master boot record, a BSD disklabel, or other metadata. GEOM also checks for physical identifiers, such as the disk's serial number. This is called *tasting*.

When GEOM finds identifying information, it configures the device as that metadata dictates. If a disk's metadata says, "I'm part of a mirror called *garbage*, along with two other disks," GEOM looks for the other disks and assembles the mirror. If GEOM can identify a storage device by format, label, or other information, it starts the device, fires up an instance of the GEOM class, makes the appropriate device nodes, and performs any other configuration it understands.

If GEOM can't identify any other metadata on the disk, such as on an unformatted and unpartitioned disk, GEOM creates the device node for the storage device and leaves it alone.

An instance of a GEOM class is called a *geom*. The gmirror(8) class makes

disks mirror each other, but the specific pair of mirrored disks named *garbage* is a geom. Each disk in that mirror is also a geom.

GEOM vs. Volume Managers

Traditional volume managers expect you to do things their way, whether that makes sense for your environment and hardware or not. If the volume manager says that you create an encrypted disk mirror by encrypting the individual drives and then mirroring on top of them, that's what you do. It might make more sense in your environment to mirror the drives and then encrypt them, but if that's not what the volume manager does, too bad. Worse, some volume managers make poor choices and then implement fixes sideways to minimize the consequences of those decisions.

GEOM differs from volume managers in that it assumes you know what you're doing. It gives you flexibility to arrange your storage in the manner that best fits your hardware and benefits your use case. GEOM classes let you easily insert new data transformations into your storage. You can't, say, add an encryption layer into your commercial volume manager.

Volume managers cover the most common cases for hardware that existed at the time they were conceived. As time passes, though, that most common case becomes increasingly uncommon. People continue to use volume managers long after the hardware they were designed for becomes obsolete. GEOM lets you evolve your designs with your hardware, environment, and application.

FreeBSD includes two software suites that look much like volume managers: gvinum(8) and ZFS. Vinum was the FreeBSD volume manager in the 1990s, and while gvinum(8) reimplements it as a GEOM class, its use is strongly discouraged. ZFS is very powerful, as we saw in Chapter 5, but it does have the "do it our way" ethos of a volume manager.

While you can theoretically stack GEOM modules forever, you must consider your hardware resources. Mirroring a busy disk across a network can require a dedicated network interface and an otherwise empty cross-connect cable. Encrypting and decrypting data eats processor time and memory. GEOM doesn't prevent you from thrashing your disks; it merely gives you new and interesting opportunities for doing so.

Providers, Consumers, and Slicers

Individual geoms are either consumers, providers, or both.

A *provider* offers services to another geom. If you're mirroring two hard drives, the geoms for the hard drive provide the disk to the mirror. A provider usually has a device node, such as */dev/ada1p1*.

A *consumer* uses the provider's services. A disk-mirror geom consumes the underlying disk drives. The consumer part of a geom doesn't need a device node.

A geom can be both a provider and a consumer—indeed, every geom in the middle of a stack must be both. A disk-mirror geom consumes the underlying physical storage media, but it provides a mirrored disk for the filesystem to live on.

FreeBSD treats all providers and consumers identically. A physical hard drive is just another provider, exactly like a mirror or encryption layer or import from the network. This characteristic lets you arbitrarily stack GEOM classes.

A GEOM class that subdivides a class is called a *slicer* and is usually responsible for managing partitions. The GEOM class that handles master boot record (MBR) partitions is a slicer, as is the GUID Partition Table (GPT) class. We discussed both of these partitioning methods in Chapter 2, and we'll go deeper into both in this chapter. Slicers must make sure that disk partitions don't overlap and that the partitions conform to the rules of the partitioning scheme.

GEOM Control Programs

Many GEOM classes have a control program that lets you administer the module or interrogate the device. Some widely used classes use geom(8), while other classes use programs like gmirror(8) or geli(8). The *disk* GEOM class talks to the physical storage media and provides consumers for upper layers. That's a really commonly used class. Here, I interrogate a host to see what geoms of type *disk* it has and print out the information the disk offers the operating system.

```
$ geom disk list
❶ Geom name: da0
```

```
Providers:
1. Name: da0
 ❷ Mediasize: 1000204886016 (932G)
 ❸ Sectorsize: 512
 ❹ Mode: r2w2e3
 ❺ descr: ATA WDC WD1003FBYZ-0
 ❻ lunname: ATA   WDC WD1003FBYZ-010FB0           WD-WCAW36478143
 ❼ lunid: 50014ee25e60dab5
 ❽ ident: WD-WCAW36478143
 ❾ rotationrate: 7200
 ❿ fwsectors: 63
    fwheads: 255
```

This hard drive provides a disk device called da0 ❶. The *mediasize* field gives its size in bytes and converts it to a more convenient 932GB ❷.

This disk claims to have a *sector size* of 512 bytes ❸. Many disks lie about their sector size. Check the drive manufacturer's documentation to determine the actual sector size. Drives might offer a *Stripesize* value of 4,096 to indicate that they're actually 4K drives.

A GEOM class's *mode* looks an awful lot like file permissions ❹, but it's really the number of GEOM classes reading from (r2) and writing to (w2) the device, plus the number of devices that have requested exclusive access to the device (e3).

The *descr* field ❺ offers the drive's model number.

The *lunname* field ❻ gives the model number plus the serial number. Yes, it's a combination of the descr and ident fields. The hard drive really, really wants you to believe this is its name and identifier.

The *lunid* ❼ gives the logical-unit-number (LUN) identifier, which describes how this drive attaches to this host.

The disk's *ident* ❽ is the drive's serial number.

The *rotationrate* ❾ tells us how fast this drive spins. It's a 7,200 RPM disk. Nonspinning disks, like SSDs, have a rotationrate of 0.

The *fwsectors* and *fwheads* fields ❿ give us the drive geometry. These are examples of the lies mentioned in the beginning of this chapter. Even SSDs offer these values.

Some drives offer less information. Virtual disks offer almost no information, and anything they do say, you can't trust. (While the VM system might say this disk offers 32,212,254,720 512-byte sectors, who knows what the actual disk beneath the virtual disk has?)

GEOM Device Nodes and Stacks

Many sysadmin tools expect to run on a disk or disk partition. Unix-like systems offer disks and partitions as device nodes. GEOM offers device nodes so that these tools remain compatible.

Most active GEOM modules have their own directory in */dev*. Device nodes within that directory represent the current providers of that module. The directory is often, but not always, named after the GEOM module using it. For example, the gmirror(8) class uses */dev/mirror*.

The directory name might be changed to avoid ambiguity or overlaps. The *glabel (GEOM label)* class uses */dev/label*. The */dev/gpt* directory contains the labels stored on GPT partitions, where */dev/gptid* contains the numerical identifiers integral to GPT partitions.

Some classes don't create a directory and instead piggyback on existing devices. The gnop(8) class creates a new node right next to the node it's attached to but appends *.nop* to the end of the device name.

Hard Disks, Partitions, and Schemes

While we discussed partitioning in Chapter 9, consider partitions from a disk drive perspective. The first possible SATA disk on our first SATA controller is called */dev/ada0*. Subsequent disks are */dev/ada1*, */dev/ada2*, and so on. If you also have SAS disks, they'll start their numbering over at 0.

Disks get further divided into *partitions*. Even average consumer-grade systems running Microsoft operating systems ship with multiple partitions on the hard drive. Sysadmins chop huge disk arrays into smaller, more manageable units with dedicated purposes—or perhaps they go the other way and merge multiple disks into one monster partition.

A *partitioning scheme* is the system for organizing partitions on a disk. The traditional master boot record (MBR) is one partitioning scheme. Old Apple and SPARC hardware have their own schemes. Today, the scheme used by

most hardware and operating systems is *GUID Partition Tables (GPT)*. Each scheme has its own requirements for boot blocks, hardware architecture, and partitions. This book discusses the MBR and GPT schemes, but you should be aware that other schemes exist.

Each disk partition gets its own device node, created by adding something to the end of the underlying device node name. Here, I look at the device node for a default FreeBSD install using UFS on a virtual disk:

```
# ls /dev/vtbd0*
/dev/vtbd0      /dev/vtbd0p1    /dev/vtbd0p2    /dev/vtbd0p3
```

We have a device node for the disk itself and then three others ending in p1, p2, and p3. What are those subdivisions? The *p* indicates that they're GPT partitions. In a default install, p1 is the boot partition, p2 is the swap space, and p3 is the main filesystem.

Each partitioning scheme has its own device node extensions. We'll read about those later this chapter.

The Filesystem Table: /etc/fstab

FreeBSD, like most Unix-like operating systems, uses the file system table */etc/fstab* to map on-disk partitions to filesystems and swap space. While ZFS doesn't use */etc/fstab*, every other FreeBSD filesystem can appear therein. Each partition in use appears on a separate line, along with mounting and management instructions.

```
/dev/gpt/rootfs  /       ufs     rw  2  1
/dev/gpt/swapfs  none    swap    sw  0  0
proc             /proc   procfs  rw  0  0
```

The first field gives the GEOM provider name. This might be a physical disk partition such as */dev/ada0p1* or perhaps a partition of a GEOM device node. The first two lines here offer device nodes under */dev/gpt*. They're GPT labels, which we'll see later this chapter. Our third entry lists the word *proc* rather than a device node: it's the procfs(5) virtual filesystem, which we'll examine in Chapter 13.

The second field gives the directory where the filesystem is available, called the *mount point*. Every partition you can read or write files on is

attached to a mount point, such as */usr*, */var*, and so on. A few special partitions, such as swap space (line 2 here), have a mount point of none. You can't read or write usable files to the swap space because they're not attached to the directory tree and because the system would overwrite those files when it swapped.

Next, we have the filesystem type. The first line shows a type of *ufs*, or Unix File System. The second line is defined as *swap* space, while the third is type *procfs*. Other types include *cd9660* (CD disks or images), *nfs* (Network File System mounts), and *ext4fs* (Linux filesystems). The filesystem table tells FreeBSD how to mount this partition. Chapter 13 discusses alternate filesystems.

The fourth field shows the mount(8) options used for this particular partition. Each filesystem has its own mount options, but here are a few that multiple filesystems use and that frequently appear in */etc/fstab*:

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

rw The filesystem is mounted read-write.

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

The fifth field is used to tell dump(8) what backup level is needed to back up this filesystem. Dump is largely obsolete these days; people perform file-level backup with tar(1) or use more advanced backup software, like Bacula (*http://www.bacula.org/*) or Tarsnap (*https://www.tarsnap.com/*).

The last field tells the FreeBSD boot process when to check filesystem integrity. All the partitions with the same number get checked in parallel with fsck(8). The root filesystem gets marked with a 1, meaning it's checked first. Only the root filesystem should get a 1. Any other partitions should get a 2 or higher, meaning they get checked later. Swap, read-only media, and logical filesystems don't require integrity checking, so they get set to 0.

FreeBSD configures all filesystems found in */etc/fstab* at boot. As the system runs, though, the sysadmin can mount other filesystems. And she can unmount ones listed there. That leads to our next question . . .

What's Mounted Now?

If not all filesystems are mounted automatically at boot, and if the sysadmin can add and remove mounted filesystems, how can you determine what's mounted right now? Use mount(8) without any options to see all mounted filesystems.

```
# mount
/dev/gpt/rootfs on / (ufs, local, journaled soft-updates)
devfs on /dev (devfs, local, multilabel)
```

This is a small UFS-based host. It has one disk partition and an instance of devfs(5) (see Chapter 13). The word *local* means that the partition is on a hard drive attached to this machine. The journaled soft-updates option is a UFS feature we'll discuss in Chapter 11. If you're using NFS or SMB to mount partitions, they'll appear here.

More complicated hosts give larger results:

```
# mount
base/ROOT/default on / (zfs, local, noatime, nfsv4acls)
base/tmp on /tmp (zfs, local, noatime, nosuid, nfsv4acls)
base/usr/home on /usr/home (zfs, local, noatime, nfsv4acls)
base/usr/ports on /usr/ports (zfs, local, noatime, nosuid, nfsv4acls)
procfs on /proc (procfs, local)
devfs on /dev (devfs, local, multilabel)
--snip--
```

This host uses many ZFS datasets, each with its own mount point. The mount(8) output shows selected ZFS options, such as noatime and nfsv4acls.

At the end of this output, we have a procfs(5) entry and one for a devfs(5) mount. A working FreeBSD system needs devfs mounted at */dev* or it won't work very well or for very long.

Disk Labeling

At the lowest level, operating systems identify disks by their physical attachment to the system. Traditionally, the filesystem table says something like, "Use the disk attached at ATA port 3 as the */var/log* filesystem." This worked fine with less flexible hardware, but as hardware technology improved, such connections became much more flexible. If you assign drive roles based on the physical attachment, sometimes that attachment changes.

I've had more than one mainboard explode at an inconvenient hour, forcing a desperate emergency replacement. Tracking which cable goes to which connecter under such circumstances never goes well. In older versions of FreeBSD, you needed to "wire down" devices so that a specific disk always showed up as a specific device node. This is no longer needed.

Today, a sysadmin uses on-disk *labels* to refer to the disk by something other than the physical attachment. A label identifies an instance of a geom. Rather than telling FreeBSD that */var/www* is on the disk attached to SATA port 2, you declare that */var/www* is on the disk labeled *website*. While the former easily goes wrong, the latter is mostly immune to sleepy hardware techs. One disk can have several labels simultaneously, if they're different types of label. FreeBSD automatically derives many labels from inherent disk characteristics; the sysadmin can define others.

Most label types have a dedicated device node directory. Each GPT partition has a *globally unique identifier (GUID)*, and the autocreated labels for those partitions live in */dev/gptid*. Disks get unique disk IDs based on their serial number, which gets entries in */dev/diskid*. Manually created GPT labels appear in */dev/gpt*.

Use these labels as you would any other device name. If you label the disk ada5 as *stuff1*, you can partition the disk stuff into *stuff1p1* and *stuff1p2*, use those partitions in configuration files, and more.

Not all labels come from GEOM. ZFS uses its own internal labeling method for filesystems and pools. You can also add labels to UFS filesystems.

Don't let swapped SATA cables ruin your weekend. Label everything.

Viewing Labels

View labels with glabel(8), a shortcut for `geom label`. Here are parts of a label from a small virtual machine. The labels on real hardware can quickly become very complex.

```
  $ glabel list
❶ Geom name: ada0p1
  Providers:
❷ 1. Name: gptid/b9c0c7c5-5b66-11e7-8aec-080027739ff6
     Mediasize: 524288 (512K)
     Sectorsize: 512
```

```
   --snip--
❸ Consumers:
   1. Name: ada0p1
      Mediasize: 524288 (512K)
      Sectorsize: 512
      Stripesize: 0
      Stripeoffset: 20480
      Mode: r0w0e0
```

This host has a single geom ❶ on the disk partition */dev/ada0p1*. It provides an appallingly long label based on the GPT partition ID ❷. We'll see a bunch of information on the underlying disk, such as the number of sectors on the disk, the sector size, and other information you might see in `geom disk list` output. This information comes from the partition, however. The physical drive information is passed up from the underlying disk.[1]

This drive has a single consumer ❸, the actual underlying partition. We're at the very bottom of this simple GEOM stack, right up against the disk, so it's consuming itself. If you add cryptographic layers or software RAID, you'll see what other device this geom consumes.

Sample Labels

Here are some examples of the kinds of labels you'll see on most FreeBSD systems.

Disk ID Labels

A physical machine offers labels not available on virtual machines.

```
Geom name: ada3
Providers:
1. Name: diskid/DISK-WD-WCAW36477141
--snip--
```

The drive ada3 provides a geom called *diskid/DISK-WD-WCAW36477141*. The *diskid* geom is named after the hard drive's serial number, based on information provided by the drive. You can remove the disk from this machine and attach it to a completely different FreeBSD host, and that new host will generate the exact same disk ID label. Using the diskid label in your configurations guarantees that FreeBSD will use the exact disk you intend. Here's how you might list partition 3 on this disk in

/etc/fstab:

```
/dev/diskid/DISK-WD-WCAW36477141p3 /usr/local ufs rw 2 2
```

This disk could attach to the host as */dev/ada3* or */dev/ada300*, and FreeBSD would still mount this partition as */usr/local*.

The problem with disk ID labels is that they're painful to read and more painful to type. I'm describing them because they can appear by default, but I'd encourage you to choose a different label. Eliminate these labels from your host by setting the tunable `kern.geom.label.disk_ident.enable` to 0 in */boot/loader.conf*.

GPT GUID Labels

Every GPT partition includes a GUID. FreeBSD can treat the GUID as a label. Here, we see a GPT ID label for partition 1 on the disk attached as ada0:

```
   Geom name: ada0p1
   Providers:
❶  1. Name: gptid/075e7b89-30ed-11e7-a386-002590dbd594
   --snip--
```

This disk partition is conveniently available as */dev/gptid/075e7b89-30ed-11e7-a386-002590dbd594* ❶. Much like disk serial numbers, GUIDs are integral to the partition. You can move the disk to another host and still get the same GPT ID. By using the GPT ID label in configurations like */etc/fstab*, you guarantee that FreeBSD uses this particular partition, rather than partition 1, on whatever device happens to get assigned ada0 at system boot.

Using a GPT ID label makes sense when you have many automatically configured disks, such as large storage arrays. On smaller systems, though, the 128-bit GUID is annoyingly long. If you decide not to use these labels, remove them from your system by setting the tunable `kern.geom.label.gptid.enable` to 0 in */boot/loader.conf*.

For most hosts, I recommend assigning GPT labels.

GPT Labels

GPT partitions let you manually assign a label name within the partition

table. I highly recommend doing so whenever possible. Here's a partition that I assigned a name:

```
   Geom name: ada2p1
   Providers:
❶ 1. Name: gpt/swap2
   --snip--
```

I've assigned the label *swap2* ❶ to partition 1 on disk ada2. This label is physically stored on the disk partition. I can use this label in my configurations just like any other device name. Using manually assigned labels is much more manageable for small systems, as this */etc/fstab* shows:

```
/dev/gpt/swap2  none  swap  sw  0  0
```

An assigned label is much more human-friendly than a long serial number or GUID. If you have the choice, I encourage you to label GPT partitions. We'll assign labels when we partition disks.

GEOM Labels

In addition to spilling the standard labels on your system, the glabel(8) command lets you configure GEOM labels. A GEOM label is specific to FreeBSD's GEOM infrastructure and appears in */dev/label*. Use GEOM labels with the `glabel label` command. Here, I apply the GEOM label *root* to the GPT partition da0p1:

```
# glabel label da0p1 root
```

There's also a `glabel create` command, but those labels disappear at system reboot.

GEOM Withering

A provider can have multiple labels. One partition might have a label based on the disk ID of the underlying storage device (*/dev/diskid/somethinglong*), a GPT ID (*/dev/gptid/somethingevenlonger*), a manually assigned label (*/dev/gpt/swap0*), and a device node based on the underlying device's attachment point (*/dev/ada0p1*). While any number of processes can look at a disk device simultaneously, many disk operations—such as mounting a

partition—require exclusive, dedicated control of the device.

To prevent accessing geoms by multiple names, when you access a device by one label, the kernel removes the unused labels. This is called *withering*. If I, say, mount a swap partition using the GPT label */dev/gpt/swap0*, all the other labels for that partition disappear from */dev*. Anyone who tries to access the corresponding */dev/gptid* partition will find that the device node is missing.

Once all exclusive locks on a device are removed, the kernel de-withers the other device labels. If I deactivate that swap space, the GPT ID and raw device name reappear.

The gpart(8) Command

Like many operating systems, FreeBSD once had specific partitioning tools for each partitioning scheme. Today, all disk partitioning functions, for MBR and GPT alike, are included in the gpart(8) program. Embedded devices with specialized storage might occasionally need older tools like fdisk(8) and bsdlabel(8), but gpart(8) works perfectly well for servers and desktops.

This common tool means you perform many functions the same way no matter which partitioning scheme you're using. For example, no matter whether you're working with the MBR or GPT scheme, you'll need a way to indicate a particular partition. Both schemes let you indicate a partition with `-i` and the partition number.

Viewing and deleting partitions are great examples of common functions.

Viewing Partitions

Use `gpart show` to see a brief summary of all GPT and MBR partitions on a geom. Give the name of a geom as an argument to see only the partitions on that geom. The output from `gpart show` doesn't look that different from fdisk(8) and other more traditional disk management tools. Here, I look at a storage device by its traditional device node, but I could use diskid or gptid or any other label:

```
$ gpart show ada0
```

```
❶ =>        40  1953525088  ada0  GPT  (932G)
❷           40        1024     1  freebsd-boot  (512K)
❸         1064         984        - free -  (492K)
❹         2048     4194304     2  freebsd-swap  (2.0G)
❺      4196352  1949327360     3  freebsd-zfs  (930G)
❻   1953523712        1416        - free -  (708K)
```

The first column gives the first block in the partition; the second, the partition size in blocks. The third gives the partition number, while the fourth gives the partition type. (We'll discuss partition types later this chapter: for the moment, just go with the flow.) At the end, we have the disk size.

Our first partition begins on the disk's sector number 40 and fills almost two billion sectors ❶. The third field shows that this isn't a partition on the disk, but rather an entry for the entire disk. The fourth field gives the partitioning scheme used. This is a GPT disk. The entire disk is about 932GB.

The second entry also starts on sector 40, and it fills 1,024 sectors ❷. This is partition 1, and it's of type *freebsd-boot*. If we want to boot off this disk, we need a boot loader on this partition.

The third entry begins on sector 1,064 and fills 984 sectors ❸. Why 1,064? The first partition started on sector 40 and filled 1,024 sectors, so the first (1,024 + 40) 1,064 sectors are filled with other partitions. But this partition doesn't have a partition number, and its type is `- free -`. This partition is aligned for disks with 4K sectors.

The fourth entry is swap space, according to the partition type ❹. It begins on sector 2,048, is 4,194,304 sectors long, and is partition 2. You'll often see swap space near the beginning of a disk, a hangover from the days when partition placement on the disk impacted performance. If you're using a virtual machine, however, putting the swap near the beginning of the disk leaves you room to expand a partition at the end of the disk.

The fifth entry is a FreeBSD ZFS filesystem, starting in sector 4,196,352 and going on for about 1.9 billion sectors ❺. This freebsd-zfs partition has our data.

The very end of the disk has 1,416 free sectors ❻. There's not quite

enough space to add space to the partition while still aligning the partition to the 1MB boundaries.

A MBR disk looks much like a GPT disk.

Other Views

Add command line flags to modify the output of `gpart show`.

You can assemble each partition's device node from the underlying device name and the partition number. If you want to see the device node rather than the partition number, add the `-p` flag.

To replace the partition type with the partition label, use `-l`.

Here, I show both the device node and the labels on this disk:

```
$ gpart show -pl ada0
=>        40  1953525088    ada0  GPT  (932G)
          40        1024  ada0p1  gptboot0  (512K)
        1064         984          - free -  (492K)
        2048     4194304  ada0p2  swap0  (2.0G)
     4196352  1949327360  ada0p3  zfs0  (930G)
  1953523712        1416          - free -  (708K)
```

The partition number now contains complete device names, like *ada0p3*. Rather than the GPT partition type, you get the label applied to the GPT partition, such as *swap0* and *zfs0*.

To see the human-hostile GPT partition type rather than the name FreeBSD presents, use `-r`. I mostly use this when examining disks from other operating systems. It's possible that FreeBSD will label multiple partition types as being type *ntfs*; while that's good enough for most uses, if I'm doing digital forensics, the precise partitioning scheme might be extremely important.

To see a more detailed description of your GPT partitions, use `gpart list`. This creates output much like `glabel list` or other GEOM class commands.

Removing Partitions

Maybe you screw up when creating your partitions and need to remove one. No, you haven't created partitions yet, in either MBR or GPT, but the process you follow is the same either way. Delete partitions by number.

Take a look at the partition table in the previous section. We have

partitions for boot, swap, and ZFS. Maybe you don't want swap space on your boot drive. Remove that partition with the gpart delete command. Use the `-i` flag and the number of the partition you want to remove. The gpart show command said the swap space was partition 2. Let's remove it.

```
# gpart delete -i 2 ada0
ada0p2 deleted
```

You can now resize your ZFS partition to use that space. How you resize a partition varies with the partitioning scheme.

Scheming Disks

No, not the sort of scheming where the disk deliberately lies to you. We're talking about the disk's partitioning scheme. Destruction is easier than creation, in both meatspace and with storage. Before you can partition a disk, you need to assign it a partitioning scheme.

Removing the Disk Partitioning Scheme

You could go through and painstakingly delete every partition on the disk and then obliterate the partitioning scheme. That's a bunch of work, though. It's much simpler to just trash the entire disk partition table.

You can't erase a disk with mounted partitions. Unmount those partitions first, and remove them from any ZFS pools. Once the disk is truly unused, erase any existing partitioning table with `gpart destroy`.

```
# gpart destroy da3
da3 destroyed
```

If the command returns immediately, the disk had no partitions. It might have had a partition scheme, but no partitions. If you get a "device busy" error, either the disk is still in use or the disk has partitions. You could methodically delete all existing partitions with `gpart delete` and then destroy the partitioning scheme, but it's easier to burn the existing scheme to the ground by adding `-F`.

```
# gpart destroy -F da3
```

This forcibly erases all partitions and the partitioning scheme. Running `gpart show da3` will show that there's no partition table. You can now create new disk partitions.

Assigning the Partitioning Scheme

Before you can create disk partitions, you need to mark the disk with the type of partitioning scheme you'll be using. Use `gpart create` with the `-s` flag and the scheme, such as `gpt` or `mbr`. Here, I mark a disk as using the GPT scheme:

```
# gpart create -s gpt da3
```

Use `gpart show` to verify that the disk now has a GPT partition table. You can now add GPT partitions or recreate the partition table with MBR and add those partitions. But we'll start by diving deep into GPT.

The GPT Partitioning Scheme

The GUID Partition Table, or GPT, is the modern standard for hard drive partitioning. This is the recommended standard for new installations. Always use the GPT partitioning scheme unless you have a deeply compelling reason not to, such as a lack of hardware support.

GPT supports disks up to 9.4ZB. One zettabyte is one billion terabytes. While our technology will eventually outgrow 9.4ZB, I expect GPT will last the rest of my career.

FreeBSD's GPT implementation currently supports 128 partitions. Each partition gets assigned a GUID, which is a 128-bit number displayed as 32 hexadecimal characters. While GUIDs aren't guaranteed to be truly unique across all of civilization, they're certainly going to be unique within your organization.

Most modern operating systems support GPT and its predecessor, the master boot record (MBR). MBR-based systems put partition records in the first sector on the disk. If a host supports only MBR, but the first sector of a disk contains something that isn't an MBR, the system gets confused and might refuse to boot. The GPT scheme puts a *protective master boot record (PMBR)* in the first sector of every disk. The PMBR indicates that the disk

contains one MBR partition of type GPT. The second sector contains the actual GUID Partition Table. GPT also puts a backup copy of the partition table on the last sector of the disk so you can more easily recover from damage.

GPT requires allocating a partition for bootstrap code. The PMBR boot code searches the disk for a FreeBSD boot partition. This boot partition must be larger than the boot code, smaller than 545KB, and reserved for the FreeBSD boot loader. FreeBSD has two GPT boot loaders, gptboot(8) and gptzfsboot(8). You must install one of these on the boot partition.

Use gptboot(8) to start UFS-based systems. At system boot, gptboot searches for a FreeBSD partition marked with the *bootme* or *bootonce* attributes.

Use gptzfsboot(8) on systems running ZFS.

Use gpart(8) and its many subcommands to view, create, edit, and destroy GPT partitions.

GPT Device Nodes

Each disk partition has a device node. GPT partition device nodes are an extension of the geom they're built on, indicated by the letter *p* and the partition number. If you've created GPT partitions directly on the disk ada0, the first partition will be */dev/ada0p1*, the second */dev/ada0p2*, and so on.

Many systems put their partitions on an upper-layer geom. One of my systems uses SATA RAID and offers the disk as */dev/raid/r0*. The partitions on this drive are */dev/raid/r0p1*, */dev/raid/r0p2*, and so on. You might also put partitions on a device by its GUID or disk ID, giving you partitions like */dev/diskid/DISK-WD-WCAW36477062p1*.

GPT Partition Types

When you create a GPT partition, you must mark it with a *partition type*. The type indicates the partition's intended use. FreeBSD makes decisions based on the partition types, so assign them correctly.

Strictly speaking, a partition type is another 128-bit GUID. FreeBSD marks GUIDs used as partition types with a leading exclamation point, such as `!516e7cb5-6ecf-11d6-8ff8-00022d09712b`. These partition types are common

across all operating system, but most OSs provide human-friendly names for these human-hostile GUIDs. This book uses the human-friendly names; check gpart(8) for the human-hostile ones.

The most common partition types you'll see on a FreeBSD system include the following:

freebsd-boot FreeBSD boot loader

freebsd-ufs FreeBSD UFS filesystem

freebsd-zfs FreeBSD ZFS filesystem

freebsd-swap FreeBSD swap partition

efi An EFI system partition, used to boot from EFI

You might also see these GPT partition types. Don't use them in modern FreeBSD, but know that their presence might help you identify just what that weird disk is and how to crack it open.

freebsd A GPT partition that's divided into bsdlabel(8) partitions

freebsd-vinum A partition controlled by gvinum(8)

mbr A partition subdivided into MBR partitions

ntfs A partition containing a Microsoft NTFS filesystem

fat16, fat32 Partitions containing FAT

For a complete listing of recognized partition types, see gpart(8).

Creating GPT Partitions

Partitioning disks is easy: figure out which partitions you want, create them, and go. The tricky part is living with your partitioning. Before creating partitions, decide what you're going to do with this disk. How much space do you have? How do you want to divide it? Before you start creating partitions, write down exactly what you want to achieve.

Here, I'm manually partitioning a 1TB disk for a UFS FreeBSD install. It'll need a 512KB boot partition (type *freebsd-boot*) and 8GB for swap (type *freebsd-swap*). The other partitions will be type *freebsd-ufs*: 5GB for root, 5GB for */tmp*, 100GB for */var*, and the rest for */usr*. I'll label each partition for its intended role.

Create partitions with gpart(8). Use the `-t` flag to specify the partition type, `-s` to give the size, and `-l` to assign a GPT label to the new partition. I'll start with the boot partition.

```
# gpart add -t freebsd-boot -l boot -s 512K da3
da3p1 added
```

Use `gpart show` to check your work. Add the `-l` flag to see the GPT label.

```
# gpart show -l da3
=>        40  1953525088  da3  GPT  (932G)
          40        1024    1  boot  (512K)
        1064  1953524064       - free -  (932G)
```

This disk has one partition, a 512K partition labeled *boot*. The command succeeded. Now add the swap space.

```
# gpart add -a 1m -t freebsd-swap -s 8g -l swap da3
da3p2 added
```

This command is much like the one to add the boot partition: we give the partition type, size, and label.

Hang on, though—what's this `-a 1m` thing? The `-a` flag lets you set a partition alignment, enabling you to set where partitions can begin and end relative to the beginning of the disk. Remember back at the beginning of this chapter when I discussed that misaligning a filesystem with the physical sectors on a 4K disk could cause problems? The `-a 1m` tells gpart to create partition on an even multiple of 1MB from the beginning of the disk. You'll have some empty space between partitions 1 and 2, as we saw in "Viewing Partitions" on page 215 in this chapter, but that's okay. That gives you room to change that partition to support UEFI if necessary (see "Unified Extensible Firmware Interface and GPT" on page 222 later this chapter).

Retain that 1MB alignment as you create the 5GB root and */tmp* partitions and the 100GB */var* partition.

```
# gpart add -a 1m -t freebsd-ufs -s 5g -l root da3
da3p3 added
# gpart add -a 1m -t freebsd-ufs -s 5g -l tmp da3
da3p4 added
# gpart add -a 1m -t freebsd-ufs -s 100g -l var da3
da3p5 added
```

When you create the last partition, don't give a size. This tells gpart to

make the partition as large as possible.

```
# gpart add -a 1m -t freebsd-ufs -l usr da3
da3p6 added
```

You have partitioned the disk, and it's ready for your install.

Resizing GPT Partitions

On second thought, perhaps having a huge */usr* partition isn't wise. A */usr* partition of 100GB or so would have all the room you might desire for operating system files, while leaving several hundred gigabytes for an isolated */home* partition. I trust most of my users, but a few[2] are just the sort to dump */dev/random* into a file until they absorb all available space. Here, I'll resize */usr* to create space for */home*.

Use `gpart resize` to change the size of a partition. You must know the target partition's partition number. Running `gpart show da3` tells us that */usr* is partition 6. Use the `-i` flag and the partition number to resize a partition.

```
# gpart resize -i 6 -s 100g -a 1m da3
da3p6 resized
```

Run `gpart show` to see the new disk size.

```
# gpart show da3
--snip--
   247465984   209715200    6  freebsd-ufs  (100G)
   457181184  1496343944       - free -  (714G)
```

This disk has 714GB free at the end. We can now create a spacious */home* for all our troublesome users.

Each partition is assigned specific sectors on the disk. You can't increase the size of a partition if there's no free space on either side of the partition. While this sample disk has a bunch of free space after partition 6, you can't use it to increase the size of partitions 1 through 5. You must delete and recreate partitions.

Changing the size of a partition doesn't change the size of the filesystem on that partition. Shrinking a partition with a filesystem will chop off part of the filesystem. Increasing the partition size won't expand the filesystem. Both UFS and ZFS have tools to handle increased partition sizes, but you

must handle that as a separate process.

Changing Labels and Types

You can modify a GPT partition's type or GPT label with the `gpart modify` command. Give the partition number with `-i`. Use `-l` to give the new label. Here, I change the GPT label on partition 2 of disk vtbd0:

```
# gpart modify -i 2 -l rootfs vtbd0
```

Similarly, change the type of partition with `-t`:

```
# gpart modify -i 2 -t freebsd-zfs vtbd0
```

The disk's GPT table now declares that partition 2 is labeled `rootfs` and is of type *freebsd-zfs*.

Booting on Legacy Hardware

Older hardware expects to see a master boot record at the start of the disk and won't recognize a GPT partition table. FreeBSD uses a protective MBR (PMBR) to give legacy hardware a recognizable partition table and help that hardware boot a GPT-partitioned disk. A bootable disk formatted with GPT needs both a protective MBR and a GPT boot loader.

Install a PMBR with the `gpart bootcode` command and the `-b` flag. FreeBSD provides a PMBR as */boot/pmbr*.

```
# gpart bootcode -b /boot/pmbr da3
bootcode written to da3
```

This disk will no longer confuse hosts that look for an MBR.

You also need a boot loader. UFS hosts need the *gptboot* boot loader, while ZFS hosts need *gptzfsboot*. For convenience, FreeBSD provides a copy of each in the */boot* directory. These copies are not the on-disk boot loader, only the version of the bootloaders needed for that version of FreeBSD. Install the selected boot loader with the `-p` flag to `gpart bootcode`. Use the `-i` option to tell gpart(8) which partition to copy the boot loader to. The sample disk we used in the last section had partition 1 as type *freebsd-boot*, so we'll use that.

```
# gpart bootcode -p /boot/gptboot -i 1 da3
partcode written to da3p1
```

You can combine `-p` and `-b` into a single command.

Unified Extensible Firmware Interface and GPT

The *Unified Extensible Firmware Interface (UEFI)* is a newer standard for booting amd64 hardware without using BIOS emulation. FreeBSD 10 and later have early support for UEFI booting to UFS, while FreeBSD 11 can boot ZFS off of UEFI.

UEFI uses a partition of type *efi*, which must be 800KB or larger. Create an efi partition on a new disk with `gpart create`.

```
# gpart create -s gpt da0
# gpart add -t efi -s 800K da0
```

FreeBSD provides an efi partition as */boot/boot1.efifat*. Copy that to the new boot partition with dd(1).

```
# dd if=/boot/boot1.efifat of=/dev/da0p1
```

Partition the rest of the disk as you desire.

An efi partition is actually a FAT filesystem with a very specific directory hierarchy. Feel free to mount the file *boot1.efifat* and explore it.

Expanding GPT Disks

We've seen how to expand a partition, but what about a disk? Expanding disks often happens with virtual hosts. Expand a virtual disk, and gpart(8) will complain that the disk's GPT is invalid. GPT and GEOM store information in the first and last sectors of the disk. Expanding a virtual disk means adding sectors. The new last sector will be empty. Create a new metadata block for the last sector with `gpart recover`.

```
# gpart recover vtbd0
```

You can now create or expand partitions on the expanded virtual disk.

Now that you have a handle on GPT partitions, let's look at MBR and

see why GPT seemed like such an improvement.

The MBR Partitioning Scheme

Old hardware, or new but small hardware, might need master boot record partitioning on its disks. Intel-style hardware has used MBR partitions for decades, and millions of devices running a plethora of operating systems use it. The MBR scheme works only on disks of 2TB or smaller. Larger disks must use GPT partitioning.

What Is the Master Boot Record?

The *master boot record (MBR)* is a file that takes up the first 512 bytes of a traditional disk, also known as *Sector 0*. The MBR contains partition information and a boot loader to allow the BIOS to find the operating system. The term *MBR* might refer to the actual first sector on the disk or the partition scheme used by that format.

A master boot record describes four *primary partitions*, called *slices* in the BSD community. Each slice description includes the disk sectors included in the partition and the type of filesystem expected on that slice. If a disk has only one slice on it, the MBR still lists four slices, but three of those slices have no sectors assigned to them. While the MBR format supports a linked list of up to 20 extended partitions, FreeBSD doesn't need them thanks to BSD labels.

One of the four primary slices is considered active. When the system powers on, the bootstrap code looks for the active slice and tries to boot it.

The MBR sector also contains bootstrap code. You don't need to allocate space specifically for a boot loader. In FreeBSD, the bootstrap code finds and executes the kernel. FreeBSD includes two different boot loaders, *mbr* and *boot0*. The mbr loader is for a host with a single operating system. If you have multiple operating systems installed on your hardware, use the boot0 loader—or, better still, dedicate your host to FreeBSD and virtualize the other operating systems.

The main function of a slice is to contain a bsdlabel(8) partition.

BSD Labels

BSD existed before either the MBR or the IBM PC. BSD used its own disk partition format, called a *disklabel*. Now that labeling disks is much more common, disklabels are also called *BSD labels* or *bsdlabels*. (If you want to start a spirited discussion, ask a room of FreeBSD developers which is more correct.) BSD systems had several partitions including at least / (root), */usr*, */var*, */tmp*, and swap space, plus separate partitions for whatever actual work the system did.

When BSD was ported to the i386 platform, they could have switched disks to using MBR partitions. With extended MBR partitions, one disk could have had up to 24 partitions. Disklabel partitions were embedded throughout the kernel, however, often in icky places that nobody dared touch. The porting group decided to treat an MBR slice as a BSD disk and to partition each slice with a BSD disklabel. Sysadmins needed to create MBR partitions and then nest disklabel partitions inside those MBR partitions.[3]

This worked but also made the word *partition* ambiguous. Does *partition* mean an MBR partition or a disklabel partition? FreeBSD dusted off the word *slices* for MBR partitions. Each MBR slice will have its own disklabel, listing the BSD partitions contained within the slice. If you come from a Linux or Microsoft Windows background, the MBR partitions you're familiar with are called *slices* over here.

You can't label slices or disklabel partitions. These formats have no space for labels. Instead, label the ZFS or UFS filesystem on the partition.

It's possible to skip slicing a disk, instead installing a disklabel directly on the hard drive. Some hardware refused to boot from such disks, so they're called *dangerously dedicated*. With the advent of GPT, dangerously dedicated disks aren't really used any more.

MBR Device Nodes

Every disk, slice, and partition has a device node. The slice device node is an extension of the underlying disk, and the partition device node is an extension of the device's node. Here are the device nodes on disk ada0 of an MBR-based system:

```
/dev/ada0       /dev/ada0s1a    /dev/ada0s1d
/dev/ada0s1     /dev/ada0s1b    /dev/ada0s1e
```

The first subdivision of the disk is the slice. Device nodes indicate a slice with the letter *s* and a number from 1 to 4. The first slice is s1, the second is s2, and so on. Unused MBR partitions don't get device nodes. Here, */dev/ada0s1* is slice 1 on the disk.

The second layer of subdivision is the disklabel partition inside the slice. Each partition has a unique device node name created by adding a letter to the slice's device node. Here, we have four disklabel partitions, */dev/ada0s1a* through */dev/ada0s1e*. Traditionally, the node ending in *a* (*/dev/ada0s1a*) is the root partition, while the node ending in *b* (*/dev/ada0s1b*) is swap space.

Note that the list of device nodes doesn't use the letter *c*. The c partition represents the entire slice. These days, you run disk partitioning tools on the slice entry rather than the disklabel for the slice.

Assign partitions d through h any way you like. A default disklabel can have up to seven usable partitions. With up to four slices on each drive, you can have up to 28 partitions on a drive. A disklabel can support up to 20 partitions, but you must indicate you want extra partitions when first creating the label.

MBR and Disklabel Alignment

Slices have their own disk sector and filesystem block alignment issues. Traditionally, MBR partitions end on a cylinder boundary. Cylinder boundaries don't mean anything on modern hardware, but even newer drives provide them as a comforting lie for older or less capable hardware. If you create MBR partitions that don't end on a cylinder boundary, and you put that disk in a machine that requires respecting cylinder boundaries, the machine will have some sort of nervous breakdown. A disk you slice today could theoretically find its way into an older system. FreeBSD therefore arranges slices so that they end on cylinder boundaries. Cylinder boundaries not only can but probably do conflict with 4K disk sector sizes. If nothing else, the MBR itself takes up the first cylinder, or sixty-three 512-byte sectors!

Fortunately you rarely write to slice tables, and the performance of writing slice tables is rarely an issue. If you align your disklabel partitions

within a slice to 1MB boundaries, you'll lose a few sectors between the slice partition table and the disklabel partition, but you'll have proper performance.

So: align disklabel partitions. Don't align slices.

Creating Slices

Use gpart(8) to manage MBR slices. To create a slice, you need a partition type and a size. FreeBSD slices use type *freebsd*. If you don't specify a size, gpart(8) uses all available space. On an empty disk, this dedicates the whole disk to a single slice.

Here, I erase the existing partitioning, tell the disk to use the MBR scheme, and create a single FreeBSD slice:

```
# gpart destroy -F ada3
# gpart create -s mbr ada3
# gpart add -t freebsd ada3
ada3s1 added
```

Run `gpart show` and you'll see that this disk now has a single slice. Add the `-p` flag to see the slice's device node.

```
# gpart show -p ada3
=>        63  1953525105    ada3  MBR  (932G)
          63  1953525105  ada3s1  freebsd  (932G)
```

Our slice ada3s1 is now ready for disklabel partitions.

To create multiple slices, specify a size with `-s`. A common configuration for small embedded systems is to put three slices on a disk. Two smaller slices contain different versions of the operating system, while the third contains any data. Here, I divide this 1TB disk into two 150GB slices and give the rest to a third slice:

```
# gpart add -s 150g -t freebsd ada3
ada3s1 added
# gpart add -s 150g -t freebsd ada3
ada3s2 added
# gpart add -t freebsd ada3
ada3s3 added
```

Removing Slices

Use `gpart delete` to remove unwanted slices. Give the slice number with `-i`. Here, I remove the third, larger slice from our multislice disk created in the last section:

```
# gpart delete -i 3 ada3
ada3s3 deleted
```

Activating Slices

The active slice is the one that the BIOS tries to boot. Set the active slice with the `-a` active flag. Use `-i` to give the number of the active slice.

```
# gpart set -a active -i 1 ada3
```

Change which slice gets booted by setting a different active slice.

The boot disk also needs a boot loader. While the MBR boot loader is different from the GPT or UEFI boot loaders, it uses the same gpart(8) `-b` flag. FreeBSD provides a copy of the MBR boot loader as */boot/mbr*.

```
# gpart bootcode -b /boot/mbr ada3
```

Slice 1 on disk ada3 is now bootable. Now that you've sliced your disk, you can create BSD labels inside the slices.

BSD Labels

Creating BSD label (or disklabel) partitions inside a slice is much like creating slices or GPT partitions. You must tell the storage device the scheme to be used, create and remove partitions until you're satisfied with them, and install a boot loader.

Creating a BSD Label

Where GPT and MBR specifically provide space for partition tables, you must create a BSD label and write it to the beginning of the slice. As with any scheme, use `-s` and the name of the scheme. Install this scheme on the slice, not on the disk.

Suppose you want to create a BSD label on the slice ada3s1. Use the BSD

scheme.

```
# gpart create -s bsd ada3s1
ada3s1 created
```

This is a default disklabel, with room for 8 disklabel partitions. You can increase the number of partitions, up to 20, by using the `-n` flag. Here, I create a whole bunch of partitions on ada3s3, the large partition.

```
# gpart create -n 20 -s bsd ada3s3
ada3s3 created
```

There are no actual disklabel partitions on this slice; there's merely a label that can contain disklabel partitions. Now that the label exists, you can create those partitions.

Creating BSD Label Partitions

Before blindly entering partitioning commands, plan how to partition the disk. Figuring things out on paper beforehand is much easier than figuring them out at the command line. I'm going to partition the first 150GB slice on this disk for UFS filesystems. This slice will get 5GB partitions for `/` (root), swap, and */tmp*. The rest will go to */usr*. Why no */var*? I'll dedicate the big slice, ada3s3, to */var*. I don't need to add a boot partition because MBR disks don't need one.

To create a disklabel partition, you must specify the type with `-t` and the size with `-s`—exactly as you would for GPT partitions. FreeBSD UFS filesystems are of type *freebsd-ufs*. Let's start with the root partition.

```
# gpart add -t freebsd-ufs -s 5g -a 1m ada3s1
ada3s1 added
```

To view this partition, you must give `gpart show` the slice device, not the disk device. Using the disk device displays the slices.

```
# gpart show ada3s1
=>        0  314572800  ada3s1  BSD  (150G)
          0       1985          - free -  (993K)
       1985   10485760       1  freebsd-ufs  (5.0G)
   10487745  304085055          - free -  (145G)
```

The third line of output shows our 5GB partition.

At the very beginning of this slice, we have 1,985 free blocks, or 993KB. I requested that the partition be aligned to 1MB boundaries, so gpart wasted a bit of space to meet that request. I'll happily lose that 993KB, rather than halve the system's performance.

Now create the swap partition of type *freebsd-swap*.

```
# gpart add -t freebsd-swap -s 5g -a1m ada3s1
ada3s1b added
```

The 5GB */tmp* comes next. Then, I dump the rest of the space into a partition for */usr* by omitting the size.

```
# gpart add -t freebsd-ufs -s 5g -a1m ada3s1
ada3s1d added
# gpart add -t freebsd-ufs -a1m ada3s1
ada3s1e added
```

A `gpart show` reveals our disklabel partitions have wasted 63 blocks, or 32KB, at the end of the disk. Watch me not care.

These partitions are now ready to receive filesystems. We discuss UFS in Chapter 11.

Assigning Specific Partition Letters

On a traditional BSD label, the *a* partition is for the root filesystem, while *b* is for swap. The *c* partition represents the entire slice. This isn't mandatory, but I recommend not using any of these letters for any other purpose.

Why is this important? I once added a hard drive to a server so that we had more space for a database. We moved the database software to partition *a* and the actual data to partition *b*.[4] When I went on vacation a few months later, the system ran short on virtual memory. I got a call from a sysadmin who had found and activated the unconfigured swap space on the new drive —but now the database data was missing. Yes, the company lost several customers and many thousands of dollars of revenue, which is sad—but more importantly, it ruined one day of my vacation and cast a shadow over the rest. This was unacceptable.

Don't bother fighting these traditions, especially on a decreasingly common disk format. Don't use the letters *a*, *b*, or *c* for partitions other than those decreed by the Berkeley elders.

The gpart program is designed to work with partition numbers, not letters. When you're creating disklabels, however, `gpart add` maps index numbers onto letters. Partition 1 is *a*, partition 2 is *b*, and so on. By specifying a partition index when you create the partition, you assign the letter to the partition.

If you don't specify a partition number, `gpart add` assigns partition letters starting with *a*. You might assign your first partition number 18, but if you don't specify a number for the next partition, it'll wind up getting partition *a*. To avoid using *a*, *b*, or *c*, use a number for every partition you create. You can use letters only up to the number of disklabel slots the partition has. A standard disklabel can use only letters *a* through *h*, while a 20-partition label can use *a* through *t*.

On my three-slice system, I want to put */var* on ada3s3. I want to use a letter other than *a*, *b*, or *c*, so I randomly pick index 18. It's almost exactly the same as the partition for */usr*, but we're adding it to a different slice.

```
# gpart add -t freebsd-ufs -a 1m -i 18 ada3s3
ada3s3r added
```

To see that disklabel partition, you'll need to run `gpart show ada3s3`. Add `-p` to see the device name.

```
# gpart show -p ada3s3
=>         0  1324379505   ada3s3  BSD  (632G)
           0        1985           - free -  (993K)
        1985  1324376064  ada3s3r  freebsd-ufs  (632G)
  1324378049        1456           - free -  (728K)
```

What do you know? The 18th letter of our alphabet is *R*.

With partitions, we can start to look at filesystems.

11
THE UNIX FILE SYSTEM

FreeBSD's filesystem, the Unix File System (UFS), is a direct descendant of the filesystem shipped with BSD 4.4. One of the original UFS authors still develops the FreeBSD filesystem and has added many nifty features in recent years. FreeBSD is not the only operating system to still use the 4.4 BSD filesystem or a descendant thereof. A Unix vendor that doesn't specifically tout its "improved and advanced" filesystem is probably running a UFS derivative.

UFS's place as the primordial filesystem has given it leave to extend tendrils throughout FreeBSD. Many UFS concepts underlie FreeBSD's support for other filesystems, from ZFS to optical disks. Even if you have no intention of ever using UFS, you must understand the basics of UFS to understand how FreeBSD manages filesystems.

Like the rest of Unix, UFS is designed to handle the most common situations effectively while reliably supporting unusual configurations. FreeBSD ships with UFS configured to be as widely useful as possible on relatively modern hardware, but you can choose to optimize a particular filesystem for trillions of small files or a half-dozen 1TB files if you must.

What we call UFS today is actually UFS version 2, or *UFS2*. Primordial UFS can't handle modern disk sizes.

UFS is best suited for smaller systems, or applications that can't handle the overhead of ZFS. Many people prefer UFS for virtual machines. I discuss choosing a filesystem in Chapter 2.

UFS Components

UFS is built of two layers, one called the *Unix File System* and the other the *Fast File System (FFS)*. UFS handles items like filenames, attaching files to directories, permissions, and all of those petty details users care about. FFS does the real work in getting files written to disk and arranging them for quick access. The two work together to provide data storage.

The Fast File System

FFS is built of superblocks, blocks, fragments, and inodes.

A *superblock* records the filesystem's characteristics. It contains a magic number that identifies the filesystem as UFS, as well as filesystem geometry information the kernel uses to optimize writing and reading files. A UFS filesystem keeps many backup copies of the superblock, in case the primary gets damaged.

Blocks are segments of disk that contain data. FreeBSD defaults to 32KB blocks. FFS maps blocks onto specific sectors on the underlying disk or GEOM provider. Every stored file gets broken up into 32KB chunks, and each chunk is stored in its own block.

Not all files are even multiples of 32KB, so FFS stores the leftovers in *fragments*. The standard is one-eighth of the block size, or 4KB. For example, a 39KB file would fill one block and two fragments. One of those fragments has only 3KB in it, so fragments do waste disk space—but they waste far less space than using full blocks everywhere.

How UFS Uses FFS

UFS allocates certain FFS blocks as *inodes*, or *index nodes*, to map blocks and fragments to files. An inode contains each file's size, permissions, and the list of blocks and fragments containing each file. Collectively, the data in an inode is known as *metadata*, or data about data.

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

many inodes remain free on your filesystem.

Theoretically, it was possible to run UFS on a storage layer other than FFS. That's how many log-based or extent-based filesystems work. Over decades of development, though, UFS features like journaling and soft updates have so greatly entangled FreeBSD's UFS and FFS that separating the two is no longer realistic or even vaguely plausible.

Vnodes

Inodes and blocks worked perfectly if the only filesystem you used was UFS and all your hard drives were permanently attached. These days, we routinely swap disks between different machines and even different operating systems. You probably need to read optical media and flash disks on your desktop, and servers might even need to accept hard drives formatted for a different operating system.

FreeBSD uses a storage abstraction layer—the *virtual node*, or *vnode*—to mediate between filesystems and the kernel. You'll never directly manipulate a vnode, but the FreeBSD documentation frequently refers to them. Vnodes are a translation layer between the kernel and whatever filesystem you've mounted. If you're an object-oriented programmer, think of a vnode like a base class that all storage classes inherit. When you write a file to a UFS filesystem, the kernel addresses the data to a vnode that, in turn, is mapped to a UFS inode and FFS blocks. When you write to a FAT32 filesystem, the kernel addresses data to a vnode that's mapped to a specific part of the FAT32 filesystem. Use inodes only when dealing with UFS filesystems, but use vnodes when dealing with any filesystem.

Mounting and Unmounting Filesystems

The mount(8) program's main function is attaching filesystems to a host's filesystem tree. While FreeBSD mounts every filesystem listed in */etc/fstab* at boot time, you must understand how mount(8) works. If you've never played with mounting before, boot your FreeBSD test machine into the single-user mode (see Chapter 4) and follow along.

In single-user mode, FreeBSD has mounted the root partition read-only. On a traditional Unix-like system, the root partition contains just enough of

the system to perform basic setup, get core services running, and find the rest of the filesystems. Other filesystems aren't mounted, so their content is inaccessible. The current FreeBSD installer puts everything in the root partition, so you'd get the basic operating system, but any special filesystems, network mounts, and so on would be empty. You might need to mount other filesystems to perform your system maintenance.

Mounting Standard Filesystems

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

```
# mount /media
```

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

```
# mount -a
```

When you mount all filesystems, filesystems that are already mounted don't get remounted.

Special Mounts

You might need to mount a filesystem at an unusual location or mount something temporarily. I most commonly mount disks manually when installing a new disk. Use the device node and the desired mount point. If my */var/db* partition is */dev/gpt/db* and I want to mount it on */mnt*, I would run:

```
# mount /dev/gpt/db /mnt
```

Unmounting a Partition

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

```
# umount /usr
```

You cannot unmount filesystems that are in use by any program. If you cannot unmount a partition, you're probably accessing it somehow. Even a command prompt in the mounted directory prevents you from unmounting the underlying partition. Running `fstat | grep /usr` (or whatever the partition is) can expose the blocking program.

UFS Mount Options

FreeBSD supports several mount options that change filesystem behavior. When you manually mount a partition, you can specify any mount option with `-o`.

```
# mount -o ro /dev/gpt/home /home
```

You can also specify mount options in `/etc/fstab` (see Chapter 10). Here, I use the `ro` option on the */home* filesystem, just as in the preceding command line.

```
/dev/gpt/home /home ufs ro 2 2
```

The mount(8) man page lists all of the UFS mount options, but here are the most commonly used ones.

Read-Only Mounts

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

Many system administrators want to mount the root partition, and perhaps even */usr*, as read-only to minimize potential system damage from an intruder or malicious software. This maximizes system stability but vastly complicates maintenance. If you use an automatic deployment system, such as Ansible or Puppet, and habitually redeploy your servers from scratch rather than upgrading them, read-only mounts might be a good fit for you.

Read-only mounts are especially valuable on a damaged computer. While FreeBSD won't let you perform a standard read-write mount on a damaged

or dirty filesystem, it will perform a read-only mount if the filesystem isn't too badly fubar. This gives you a chance to recover data from a dying disk.

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

Synchronous Mounts

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

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

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

Asynchronous Mounts

While *asynchronous mounts* are pretty much supplanted by soft updates (see "Soft Updates" on page 237), you'll still hear about them. For faster data access at higher risk, mount your partitions asynchronously. When a disk is asynchronously mounted, the kernel writes data to the disk and tells the writing program that the write succeeded without waiting for the disk to confirm that the data was actually written.

Asynchronous mounting is fine on disposable filesystems, such as memory file systems that disappear at shutdown, but don't use it with important data. The performance difference between asynchronous mounts and noasync with soft updates is minuscule. (I'll cover noasync in the next section.)

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

Combining Sync and Async

FreeBSD's default UFS mount option combines sync and async mounts as *noasync*. With noasync, data that affects inodes is written to the disk

synchronously, while actual data is handled asynchronously. Combined with soft updates (see later in this chapter), a noasync mount creates a very robust filesystem.

As noasync mounts are the default, you don't need to specify it when mounting, but when someone else does, don't let it confuse you.

Disable Atime

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

Disable Execution

Your policy might say that certain filesystems shouldn't have executable programs. The `noexec` mount option prevents the system from executing any programs on the filesystem. Mounting */home* noexec can help prevent users from running their own programs, but for it to be effective, also mount */tmp*, */var/tmp*, and anywhere else users can write their own files noexec as well.

A noexec mount doesn't prevent a user from running a shell script or an interpreted script in Perl or Python or whatever. While the script might be on a noexec filesystem, the interpreter usually isn't.

Another common use for a noexec mount is when you have a filesystem that contains binaries for a different operating system or a different hardware architecture and you don't want anyone to execute them.

Disable Suid

Setuid programs allow users to run programs as if they're another user. For example, programs such as login(1) must perform actions as root but must be run by regular users. Setuid programs obviously must be written carefully so that intruders can't exploit them to get unauthorized access to your system. Many system administrators habitually disable all unneeded setuid programs.

The `nosuid` option disables setuid access from all programs on a filesystem. As with noexec, script wrappers can easily evade nosuid restrictions.

Disable Clustering

FFS optimizes reads and writes on the physical media by clustering. Rather than scattering a file all over the hard drive, it writes out the whole thing in large chunks. Similarly, it makes sense to read files in larger chunks. You can disable this feature with the mount options `noclusterr` (for read clustering) and `noclusterw` (for write clustering).

Disable Symlinks

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

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

UFS Resiliency

UFS dates from the age when a power loss meant data loss. After decades of use and debugging, UFS almost never loses data, especially when compared with other open source filesystems. UFS achieves this resiliency by careful integrity checking, especially after an unexpected shutdown like a power failure.

The point of resiliency isn't to verify the data on disk—UFS is pretty good at that. It's to speed integrity verification and filesystem recovery after that unexpected shutdown. The size of modern disks means that verification can take a long time without additional resiliency. An integrity check of a 100MB filesystem is much faster than the same integrity check of a multiterabyte filesystem! Adding resiliency improves recovery times.

UFS offers several ways to improve the resilience of a UFS filesystem, such as soft updates and journaling. Before creating a filesystem, choose one that fits your needs.

Soft Updates

Soft updates is a technology used to organize and arrange disk writes so that

filesystem metadata remains consistent at all times, giving nearly the performance of an async mount with the reliability of a sync mount. That doesn't mean that all data will be safely written to disk—a power failure at the wrong moment can still lose data. The file being written to disk at the exact millisecond the power dies can't get to the disk no matter what the operating system does. But what's actually on the disk will be internally consistent. Soft updates lets UFS quickly recover from failure.

You can enable and disable soft updates when mounting or creating the filesystem.

As filesystems grow, soft updates show their limits. Multiterabyte filesystems still need quite a while to recover from an unplanned shutdown. The original soft updates journaling paper (*http://www.mckusick.com/softdep/suj.pdf*) mentions that a 92 percent full 14-drive array with a deliberately damaged filesystem needed 10 hours for integrity checking. You'll need a journal well before then.

Soft Updates Journaling

A journaling filesystem records any changes outside the actual filesystem. Changes get quickly dumped to storage and then inserted into the filesystem at a more leisurely pace. If the system dies unexpectedly, the filesystem automatically recovers any changes from the journal. This vastly reduces the requirement for rebuilding filesystem integrity at startup. When you install FreeBSD, it defaults to creating UFS partitions with soft update journaling.

Rather than recording all transactions, the soft updates journal records all metadata updates so that the filesystem can always be restored to an internally consistent state. Benchmarks show that journaling adds only a tiny amount of load to soft updates. It does add I/O overhead, however, as the system must dump all changes to the journal and then replay them into the filesystem. It vastly reduces recovery time, however. That 14-drive array that needed 10 hours for integrity checking? It needed less than one minute to recover from the same damage using the journal.

Soft updates with journaling is very powerful. Why wouldn't you always use journaling? Soft updates journaling disables UFS snapshots. If you need UFS snapshots, you can't journal. If you need snapshots, though, you're probably better off using ZFS anyway. FreeBSD's version of dump(8) uses

UFS snapshots to back up live filesystems. Only us old Unix hands use dump any more, and that's mostly because we already know it, but if your organization mandates using dump(8), you need another resiliency option.

GEOM Journaling

FreeBSD can also journal at the GEOM level with gjournal(8). Like any other filesystem journal, gjournal records filesystem transactions. At boot, FreeBSD checks the journal file for any changes not yet written to the filesystem and makes those changes, ensuring a consistent filesystem. Gjournal predates soft updates journaling.

While soft updates journals only metadata, gjournal journals all filesystem transactions. You're less likely to lose data in a system failure, but everything gets written twice, which impacts performance. If you're using gjournal, though, don't use any type of soft updates. You should also mount the filesystem async. You can use snapshots on a gjournaled filesystem.

Gjournal uses 1GB of disk per filesystem. You can't just turn it on and off—you must have space for the journal. You can use a separate partition for the journal or include the gigabyte in the partition if you leave space for it. If you decide to add gjournal to an existing partition, you need to find the space somewhere.

Should you use gjournal or soft updates journaling? I recommend using soft updates journaling if at all possible. If that isn't an option, use plain soft updates. Use GEOM journaling if you need UFS snapshots, including dump(8) on snapshots. Personally, I no longer use gjournal.

Creating and Tuning UFS Filesystems

In the last chapter, we partitioned and labeled your disks. Now let's put a filesystem on those partitions. Create UFS filesystems with newfs(8), using a device node as the last argument. Here, I create a filesystem on the device */dev/gpt/var*:

```
# newfs /dev/gpt/var
/dev/gpt/var: ❶51200.0MB (104857600 sectors) ❷block size 32768, ❸fragment size 4096
        using ❹82 cylinder groups of 626.09MB, 20035 blks, 80256 inodes.
super-block backups (for fsck_ffs -b #) at:
```

```
❺192, 1282432, 2564672, 3846912, 5129152, 6411392, 7693632, 8975872,
--snip--
```

The first line repeats the device node and prints the partition's size ❶, along with the block ❷ and fragment sizes ❸. You'll get filesystem geometry information ❹, a relic of the days when disk geometry bore some relationship to the hardware. Finally, newfs(8) prints a list of super-block backups ❺. The larger your filesystem, the more backup superblocks you get.

If you want to use soft updates journaling, add the `-j` flag. To use soft updates without journaling, add the `-U` flag. After you've created the filesystem, you can enable and disable soft updates journaling, and plain soft updates, with tunefs(8).

UFS Labeling

Device nodes can change, but labels remain constant. Best practice is to label GPT partitions, but you can't label MBR partitions. UFS filesystems on an MBR can use a UFS label with the `-L` flag.

```
# newfs -L var /dev/ada3s1d
```

The labels appear in */dev/ufs*. Use them in */etc/fstab* and other configuration files to avoid disk renaming mayhem. You can't apply UFS labels to non-UFS filesystems.

If you're using UFS on GPT partitions, choose either GPT or UFS labels. Thanks to withering, you'll see only one label at a time and probably confuse yourself.

Block and Fragment Size

UFS's efficiency is proportional to the number of blocks and fragments read or written. Generally, FreeBSD can read a 10-block file in half the time it needs to read a 20-block file. The FreeBSD developers chose the default block and fragment sizes to accommodate the widest variety of files.

If you have a special-purpose filesystem that overwhelmingly contains either large or small files, you might consider changing the block size when

creating the filesystem. While you can change the block size of an existing filesystem, it's a terrible idea. Block sizes must be a power of 2. The assumption that a fragment is one-eighth the size of a block is hardcoded in many places, so let newfs(8) compute the fragment size from the block size.

Suppose I have a filesystem dedicated to large files, and I want to increase the block size. The default block size is 32KB, so the next larger block size would be 64KB. Specify the new block size with `-b`.

```
# newfs -b 64K -L home /dev/da0s1d
```

If you're going to have many small files, you might consider using a smaller block size. One thing to watch out for is a fragment size smaller than the underlying disk's physical sector size. FreeBSD defaults to 4KB fragments. If your disk has 4KB sectors, don't use a smaller fragment size. If you're absolutely certain that your disk has 512-byte physical sectors, you can consider creating a filesystem with a 16KB (or even 8KB) block size and the corresponding 2KB or 1KB fragment size.

In my sysadmin career, I have needed[1] a custom block size only twice. Don't use one until you experience a performance issue.

Using GEOM Journaling

Before using gjournal(8), decide where you're putting the 1GB journal. If possible, I'd recommend including that gigabyte in the filesystem partition. That means if you want a 50GB filesystem, put it in a 51GB partition. Otherwise, use a separate partition.

Load the geom_journal kernel module with `gjournal load` or in */boot/loader.conf* before performing any gjournal operations.

To create a gjournal provider while including the partition in the journal, use the `gjournal label` command.

```
# gjournal label da3p5
```

If you want to have a separate provider be the journal, add that provider as a second argument.

```
# gjournal label da3p5 da3p7
```

These commands run silently if successful. They create a new device node with the same name as your journaled device, but with *.journal* added to the end. Running `gjournal label da3p5` creates */dev/da3p5.journal*. From this point on, do all work on the journaled device node.

Create your new UFS filesystem on the journaled device. Use the `-J` flag to tell UFS it's running on top of gjournal. Do not enable any sort of soft updates, including soft updates journaling. It seems to work for a time . . . then it doesn't.

Mount your gjournal filesystems async. The normal warnings that apply to async mounts don't apply to gjournal, however. The gjournal GEOM module handles the verification and integrity checking normally managed by the filesystem.

```
/dev/da3p5.journal /var/log ufs rw,async 2 2
```

The documentation says that you can convert an existing partition to use gjournal, provided that you have a separate partition for the journal and that the last sector of the existing filesystem is empty. In practice, I find that the last sector of the existing filesystem is always full, but if you want to, try to read gjournal(8) for the details.

Tuning UFS

You can view and change the settings on each UFS filesystem by using tunefs(8). This lets you enable and disable features; plus, you can adjust how UFS writes files, manages free space, and uses filesystem labels.

View Current Settings

View a filesystem's current settings with the `-p` flag and the partition's current mount point or underlying provider.

```
   # tunefs -p /dev/gpt/var
   tunefs: POSIX.1e ACLs: (-a)                                  disabled
❶  tunefs: NFSv4 ACLs: (-N)                                     disabled
❷  tunefs: MAC multilabel: (-l)                                 disabled
❸  tunefs: soft updates: (-n)                                   enabled
❹  tunefs: soft update journaling: (-j)                         enabled
❺  tunefs: gjournal: (-J)                                       disabled
   tunefs: trim: (-t)                                           disabled
```

```
   tunefs: maximum blocks per file in a cylinder group: (-e)  4096
   tunefs: average file size: (-f)                            16384
   tunefs: average number of files in a directory: (-s)       64
❻ tunefs: minimum percentage of free space: (-m)              8%
   tunefs: space to hold for metadata blocks: (-k)            6408
   tunefs: optimization preference: (-o)                      time
❼ tunefs: volume label: (-L)
```

Many of the available settings relate to specific security functionality we don't cover. Topics like MAC restrictions ❷ and all the different types of ACL ❶ fill entire books. But we can see that this filesystem uses soft updates ❸ and soft updates journaling ❹, though it doesn't use gjournal ❺. We get the minimum amount of free space ❻. At the end, we have the nonexistent UFS label ❼. We get a bunch of information on filesystem geometry and block size.

Use tunefs(8) to change any of these settings on an unmounted filesystem. Conveniently, tunefs(8) shows the command line flag to address each. I normally boot into single-user mode before changing a filesystem's settings.

You might notice that you can adjust all sorts of filesystem internals, such as block arrangements and filesystem geometry. Don't. In over two decades of FreeBSD use, I have never seen anyone improve their situation by twiddling these knobs. I have repeatedly seen people twiddle these knobs and ruin their day.

But let's look at the settings you might actually need to enable and disable.

Soft Updates and Journaling

Use the `-j` flag to enable or disable soft updates journaling on a filesystem. This automatically enables soft updates.

```
# tunefs -j enable /dev/gpt/var
Using inode 5 in cg 0 for 33554432 byte journal
tunefs: soft updates journaling set
```

To disable soft updates journaling, use the `disable` keyword.

```
# tunefs -j disable /dev/gpt/var
Clearing journal flags from inode 5
tunefs: soft updates journaling cleared but soft updates still set.
```

```
tunefs: remove .sujournal to reclaim space
```

A soft updates journal on a nonjournaled filesystem can only confuse matters. Mount the filesystem and remove the *.sujournal* file in the filesystem's root directory. Note that turning off journaling leaves soft updates still in place. Use `-n enable` and `-n disable` to turn soft updates (without journaling) on and off.

Minimum Free Space

UFS holds back 8 percent of each partition so that it has space to rearrange files for better performance. I discuss this further in "UFS Space Reservations" on page 249. If you want to change this percentage, use the `-m` flag. Here, I tell the filesystem to reserve only 5 percent of the disk.

```
# tunefs -m 5 /dev/gpt/var
tunefs: minimum percentage of free space changes from 8% to 5%
tunefs: should optimize for space with minfree < 8%
```

You should now have more usable disk space. Also, UFS will run more slowly because it always packs the filesystem as tightly as possible.

SSD TRIM

Solid-state disks use wear-leveling to extend their lifespan. Wear-leveling works best if the filesystem notifies the SSD when each block is no longer in use. The TRIM protocol handles this notification. Enable TRIM support on your SSD-backed filesystem with the `-t` flag.

```
# tunefs -t enable /dev/gpt/var
tunefs: issue TRIM to the disk set
```

For the best results, enable TRIM for every partition on a solid-state drive. Enable TRIM at filesystem creation with `newfs -E`.

Labeling UFS Filesystems

You can apply a UFS label to an existing filesystem with the `-L` flag.

```
# tunefs -L scratch /dev/ada3s1e
```

Don't mix UFS and GPT labels—you'll only confuse yourself.

Expanding UFS Filesystems

Your virtual machine runs out of space? Make the disk bigger, and expand the last partition to cover that space, as discussed in Chapter 10. But what about the filesystem on that partition? That's where growfs(8) comes in.

The growfs(8) command expands an existing UFS filesystem to fill the partition it's in. Give growfs one argument, the filesystem's device node. Use labels if you like.

```
# growfs /dev/gpt/var
It's strongly recommended to make a backup before growing the file system.
OK to grow filesystem on /dev/gpt/var from 50.0GB to 100GB? [Yes/No] ❶yes
super-block backups (for fsck_ffs -b #) at:
 19233792, 20516032, 21798272, 23080512, 24362752,
--snip--
```

When growfs(8) requests confirmation ❶, you must enter the full word `yes`. Any other answer, including a plain `y` like many other programs accept, cancels the operation. Confirm the operation and growfs(8) will add additional blocks, superblocks, and inodes as needed to fill the partition.

If you don't want the filesystem to fill the entire partition, you can specify a size with `-s`. Here, I expand this same partition to 80GB.

```
# growfs -s 80g /dev/gpt/var
```

I strongly encourage you to make filesystems the same size as the underlying partitions, unless you're looking to make your coworkers slap you.[2]

UFS Snapshots

You can take an image of a UFS filesystem at a moment in time; this is called a *snapshot*. You can snapshot a filesystem, erase and change some files, and then copy the unchanged files from the snapshot. Tools like dump(8) use snapshots to ensure consistent backups. UFS snapshots are not as powerful or flexible as ZFS snapshots, but they're a solid, reliable tool within their limits.

UFS snapshots require soft updates but are incompatible with soft updates journaling. Each filesystem can have up to 20 snapshots.

Snapshots let you get at the older version of an edited or removed file. Access the contents of a snapshot by mounting the file as a memory device. I'll discuss memory devices in Chapter 13.

Taking and Destroying Snapshots

Create snapshots with mksnap_ffs(8). This program assumes you want to make a snapshot of the filesystem your current working directory is in. Give the snapshot location as an argument. Snapshots traditionally go in the *.snap* directory at the filesystem root. If you're using a tool that automatically creates and removes snapshots, like dump(8), check there for your snapshot files. If you don't like that location, though, you can put them anywhere on the filesystem you're taking the snapshot of. Here, I took a snapshot of the */home* filesystem:

```
# cd /home
# mksnap_ffs .snap/beforeupgrade
```

Snapshots use disk space. You can't take a snapshot of a full filesystem.

A snapshot is just a file. Remove the file and you destroy the snapshot.

Finding Snapshots

Snapshots are files, and you can put them anywhere on the filesystem. This means it's easy to lose them. Use find(1) with the `-flags snapshot` option to find all snapshots on a filesystem.

```
# find /usr -flags snapshot
/usr/.snap/beforeupgrade
/usr/.snap/afterupgrade
/usr/local/testsnap
```

There's my stray snapshot!

Snapshot Disk Usage

A snapshot records the differences between the current filesystem and the filesystem as it existed when the snapshot was taken. Every filesystem change after taking a snapshot increases the size of the snapshot. If you remove a file, the snapshot retains a copy of that file so you can recover it later.

This means deleting data from a filesystem with snapshots doesn't actually free up space. If you have a snapshot of your */home* partition and you delete a file, the deleted file gets added to the snapshot.

Make sure that filesystems with snapshots always have plenty of free space. If you try to take a snapshot and mksnap_ffs(8) complains that it can't because there's no space, you might already have 20 snapshots of that filesystem.

UFS Recovery and Repair

Everything from faulty hardware to improper systems administration[3] can damage your filesystems. All of UFS's resilience technologies are designed to quickly restore data integrity, but nothing can completely guarantee integrity.

Let's discuss how FreeBSD keeps each UFS filesystem tidy.

System Shutdown: The Syncer

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

You'll see odd things from the syncer during shutdown. The syncer walks the list of vnodes that need synchronizing to disk, allowing it to support all filesystems, not just UFS. Thanks to soft updates, writing one vnode to disk can generate another dirty vnode that needs updating. You can see the number of buffers being written to disk rapidly drop from a high value to a low value and perhaps bounce between zero and a low number once or twice as the system really, truly synchronizes the hard drive.

If the syncer doesn't get a chance to finish, or if the syncer doesn't run at all thanks to your ham-fisted fumbling, you get a dirty filesystem.

Dirty Filesystems

No, disks don't get muddy with use (although dust on a platter will quickly damage it, and adding water won't help). A dirty UFS partition is in a kind

of limbo; the operating system has asked for information to be written to the disk, but the data is not yet completely on the physical media. Part of the data blocks might have been written, the inode might have been edited but the data not written out, or any combination of the two. Live filesystems are almost always dirty.

If a host with dirty filesystems fails—say, due to a panic or Bert tripping over the power cable, the filesystem is still dirty when the system boots again. The kernel refuses to mount a dirty filesystem.

Cleaning the filesystem restores data integrity but doesn't necessarily mean that all your data is on the disk. If a file was half-written to disk when the system died, the file is lost. Nothing can restore the missing half of the file, and the half on disk is essentially useless.

Journaled filesystems should automatically recover when FreeBSD tries to mount them. If the filesystem can't recover, or if you don't have a journal, you'll need to use the legendary fsck(8).

File System Checking: fsck(8)

The fsck(8) program examines a UFS filesystem and tries to verify that every file is attached to the proper inodes and in the correct directory. It's like verifying a database's referential integrity. If the filesystem suffered only minor damage, fsck(8) can automatically restore integrity and put the filesystem back in service.

Repairing a damaged filesystem takes time and memory. A fsck(8) run requires about 700MB of RAM to analyze a 1TB filesystem. Most computer systems have fairly proportional memory and storage systems: very few hosts have 512MB RAM and petabytes of disk. But you should know it's possible to create a UFS filesystem so large that the system doesn't have enough memory to repair it.

Manual fscks Runs

Occasionally this automated fsck-on-reboot fails to work. When you check the console, you'll be looking at a single-user mode prompt and a request to run fsck(8) manually.

Start by *preening* the filesystem with `fsck -p`. This automatically corrects a bunch of less severe errors without asking for your approval. Preening causes

data loss only rarely. This is frequently successful, but if it doesn't work, it will ask you to run a "full fsck."

If you enter `fsck` at the command prompt, fsck(8) verifies every block and inode on the disk. It finds any blocks that have become disassociated from their inodes and guesses how they fit together and how they should be attached. However, fsck(8) might not be able to identify which directory these files belong in.

Then, fsck(8) asks whether you want to perform these reattachments. If you answer `n`, it deletes the damaged files. If you answer `y`, it adds the lost file to a *lost+found* directory in the root of the partition, with a number as a filename. For example, the *lost+found* directory on your */usr* partition is */usr/lost+found*. If there are only a few files, you can identify them manually; if you have many files and are looking for particular ones, tools such as file(1) and grep(1) can help you identify them by content.

If you answer `n`, those nuggets of unknown data remain detached from the filesystem. The filesystem remains dirty until you fix them by some other means.

Trusting fsck(8)

If fsck(8) can't figure out where a file goes . . . can you? If not, you really have no choice but to trust fsck(8) to recover your system or restore from backup.

A full fsck(8) run inspects every block, inode, and superblock, and identifies every inconsistency. It asks you to type `y` or `n` to approve or reject every single correction. Any change you reject you must fix yourself, through some other means. You might spend hours at the console typing `y`, `y`, `y`.

So I'll ask again: if fsck(8) can't fix a problem, can you?

If you can't, consider `fsck -y`. The `-y` flag tells fsck(8) to reassemble these files as best it can, without prompting you. It assumes you answer all its questions "yes," even the really dangerous ones. Using `-y` automatically triggers `-R`, which tells fsck(8) to retry cleaning each filesystem until it succeeds or it's had 10 consecutive failures. It's cure or kill. You *do* have backups, right?

DANGER!

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

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

Avoiding fsck -y

What options do you have if you don't want to use `fsck -y`? Well, fsdb(8) and clri(8) allow you to debug the filesystem and redirect files to their proper locations. You can restore files to their correct directories and names. This is difficult,[4] however, and is recommended only for Secret Ninja Filesystem Masters.

Background fsck

Background fsck gives UFS some of the benefits of a journaled filesystem without actually requiring journaling. You must be using soft updates without journaling to use background fsck. (Soft updates with journaling is far, far preferable to background fsck.) When FreeBSD sees that a background fsck is in process after a reboot, it mounts the dirty disk read-write. While the server is running, fsck(8) runs in the background, identifying loose bits of files and tidying them up behind the scenes.

A background fsck actually has two major stages. When FreeBSD finds dirty disks during the initial boot process, it runs a preliminary fsck(8) assessment of the disks. The fsck(8) program decides whether the damage can be repaired while the system is running or whether a full single-user

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

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

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

Forcing Read-Write Mounts on Dirty Disks

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

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

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

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

- If the filesystem is clean, it is mounted without fsck(8).
- If a journaled filesystem is dirty at boot, FreeBSD recovers the data

from the journal and continues the boot. A journaled filesystem rarely needs fsck(8).

- If a filesystem without soft updates is dirty at boot, FreeBSD runs fsck(8) on it. If the filesystem damage is severe, FreeBSD stops checking and requests your intervention. You can either run `fsck -y` or manually approve each correction.
- If a filesystem with soft updates is dirty at boot, FreeBSD performs a very basic fsck(8) check. If the damage is mild, FreeBSD can use a background fsck(8) in multiuser mode.
- If the damage is severe, or you don't want background fsck(8), FreeBSD interrupts the boot and requests a manual fsck(8).

Consider the recovery path when configuring your UFS filesystems.

UFS Space Reservations

A UFS filesystem is never quite as large as you think it should be. UFS holds back 8 percent of the filesystem space for on-the-fly optimization. Only root can write over that limit. That's why a filesystem can seem to use more than 100 percent of the available space. Why 8 percent? That number's the result of many years of experience and real-world testing. That 8 percent holdback isn't a big deal on average filesystems, but as the filesystem grows, it can be considerable. On a 1PB disk array, UFS holds 80TB in reserve.

UFS behaves differently depending on how full a filesystem gets. On an empty filesystem, it optimizes for speed. Once the filesystem hits 92 percent full (85 percent of the total size, including the 8 percent reserve), it switches to optimize space utilization. Most people do the same thing—once you mostly fill up the laundry hamper, you can jam more dirty clothes in, but it takes a little more time and effort. UFS fragments files to use space more effectively. Fragments reduce disk performance. As free space shrinks, UFS works harder and harder to improve space utilization. A full UFS filesystem runs at about one-third the normal speed.

You might want to use tunefs(8) to reduce the amount of disk space FreeBSD holds in reserve. It won't help as much as you think. Reducing the reserve to 5 percent or less tells UFS to always use space optimization and pack the filesystem as tightly as possible.

Increasing the reserved space percentage doesn't improve performance. If you increase the reserved space percentage so that your filesystem appears full, regular users won't be able to write files.[5]

The reserved space can confuse tools such as NFS. Some other operating systems that can mount UFS over NFS see that a filesystem is 100 percent full and tell the user they can't write files, despite local clients being able to write files. Remember this when troubleshooting.

The best thing to do is to keep your partition from filling up.

How Full Is a Partition?

To get an overview of how much space each UFS partition has left, use df(1). This lists the partitions on your system, the amount of space each uses, and where it's mounted. (Don't use df(1) with ZFS; we'll discuss why in the next chapter.)

$BLOCKSIZE

One annoying thing about FreeBSD's disk utilities, including df(1), is that they default to providing information in 512-byte blocks. Blocks were fine with tiny disks that used 512-byte physical blocks, but it's not a useful measurement today. The environment variable `$BLOCKSIZE` controls what unit df(1) provides output in. The default *.cshrc* and *.profile* set `$BLOCKSIZE` to 1KB, which makes df(1) show kilobytes instead of blocks.

The `-h` and `-H` flags tell df(1) to produce human-readable output rather than using blocks. The small `-h` uses base 2 to create a 1,024-byte megabyte, while the large `-H` uses base 10 for a 1,000-byte megabyte. Typically, network administrators and disk manufacturers use base 10, while system administrators use base 2. Either works so long as you know which you've chosen. I'm a network administrator, so you get to suffer through my prejudices in these examples, despite what my tech editor thinks.

```
   # df -H
❶ Filesystem      Size    Used   Avail Capacity  Mounted on
❷ /dev/gpt/root   1.0G    171M    785M      18%  /
  devfs           1.0k    1.0k      0B     100%  /dev
  /dev/gpt/var    1.0G     64M    892M       7%  /var
  /dev/gpt/tmp    1.0G    8.5M    948M       1%  /tmp
❸ /dev/gpt/usr     14G   13.8G    203M      98%  /usr
```

The first line shows us column headers ❶ for the provider name, the size of the partition, the amount of space used, the amount of space available, the percent of space used, and the mount point. We can see that the partition labeled */dev/gpt/root* ❷ is only 1GB in size but has only 171MB on it, leaving 785MB free. It's 18 percent full and mounted on /.

If your systems are like mine, disk usage somehow keeps growing for no apparent reason. Look at the */usr* partition ❸ here. It's 98 percent full. You can identify individual large files with `ls -l`, but recursively doing this on every directory in the system is impractical.

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

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

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

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

example, `-d0` goes one directory deep and gives a simple subtotal of the files in a directory.

```
# du -h -d0 $HOME
 14G    /home/mwlucas
```

I have 14 gigs of data in my home directory? Let's look a layer deeper and identify the biggest subdirectory.

```
# du -h -d1
 38K    ./bin
 56M    ./mibs
--snip--
 13G    ./startrekgifs
--snip--
```

Apparently I must look elsewhere for storage space, as the data in my home directory is too important to delete. Maybe I should just grow the virtual disk under this host.

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

Adding New UFS storage

No matter how much planning you do, eventually your hard drives will fill up. You'll need to add disks. Before you can use a new hard drive, you must partition the drive, create filesystems, mount those filesystems, and move data to them.

Give the design of your new disk partitioning and filesystems as much thought as you did the initial install. It's much easier to partition disks correctly at install than to go back and repartition disks with data on them.

BACK UP, BACK UP, BACK UP!

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

Partitioning the Disk

While you can partition the disk any way you like, I recommend that new disks use the same partitioning scheme as the rest of the host. Having one disk partitioned with MBR and one with GPT is annoying. I'll use GPT for this example.

Decide how you want to divide the disk. This is a 1TB disk. 100GB will go to an expanded */tmp*. I'll dedicate 500GB to my new database partition. The remaining space gets partitioned off but labeled *emergency*. I won't put a filesystem in that space; it's there in case I need to do a full memory dump or have to put some files somewhere. I'm putting it right next to the database partition so I can grow the database partition if needed. I could leave the emergency space unpartitioned, but I want it to have a GPT label so that my fellow sysadmins realize this free space isn't accidental.

Start by destroying any partitioning scheme on the disk and creating a GPT scheme.

```
# gpart destroy -F da3
da3 destroyed
# gpart create -s gpt da3
da3 created
```

Now create your 100GB */tmp* and 500GB data partitions, and dump the rest into the emergency partition.

```
# gpart add -t freebsd-ufs -l tmp -s 100g da3
da3p1 added
# gpart add -t freebsd-ufs -l postgres -s 500g da3
da3p2 added
# gpart add -t freebsd-ufs -l emergency da3
da3p3 added
```

Check your work with `gpart show`.

```
# gpart show -lp da3
=>        40  1953525088    da3  GPT  (932G)
          40   209715200  da3p1  tmp  (100G)
   209715240  1048576000  da3p2  postgres  (500G)
  1258291240   695233888  da3p3  emergency  (332G)
```

Create filesystems on each partition.

```
# newfs -j /dev/gpt/tmp
# newfs -j /dev/gpt/postgres
```

As */tmp* gets emptied at every boot, I would prefer not to use soft updates journaling on */tmp*. Instead, I'd mount */tmp* async and run `newfs /dev/gpt/tmp` at boot. Many times, newfs(8) is faster than rm(1).

Configuring /etc/fstab

Now tell */etc/fstab* about your filesystems. We discuss the format of */etc/fstab* in Chapter 10.

```
/dev/gpt/postgres  /usr/local/etc/postgres ufs  rw  0  2
/dev/gpt/tmp       /tmp                    ufs  rw  0  2
```

FreeBSD will recognize the filesystems at boot, or you can mount these new partitions at the command line. Don't reboot or mount the partitions just yet, though. First you'll want to move files to those filesystems.

Installing Existing Files onto New Disks

Chances are that you intend your new disk to replace or subdivide an existing partition. You'll need to mount your new partition on a temporary mount point, move files to the new disk, then remount the partition at the desired location. While */tmp* doesn't have any files, if we're installing a new database filesystem, we presumably have database files to put there.

Before moving files, shut down any process using them. You cannot successfully copy files that are being changed as you copy them. If you're moving your database files, shut down your database. If you're moving your mail spool, shut down all of your mail programs. This is a big part of why I recommend doing all new disk installations in single-user mode.

Now mount your new partition on a temporary mount point. That's exactly what */mnt* is for.

```
# mount /dev/gpt/postgres /mnt
```

Now you must move the files from their current location to the new disk

without changing their permissions. This is fairly simple with tar(1). You can simply tar up your existing data to a tape or a file and untar it in the new location, but that's kind of clumsy. Pipe one `tar` into another to avoid the middle step.

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

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

```
# tar cfC - /usr/local/etc/postgres . | tar xpfC - /mnt
```

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

```
# rm -rf /usr/local/etc/postgres
# umount /mnt
# mount /usr/local/etc/postgres
```

You can now resume normal operation. I recommend rebooting to verify that everything comes back exactly as you intended.

Stackable Mounts

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

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

Here's the problem: the old filesystem still has all its original data on it.

The new filesystem is mounted "above" the old filesystem, so you see only the new filesystem. The old filesystem has no more free space than before you moved the data. If you unmount the new filesystem and check the directory again, you'll see the data miraculously restored! The new filesystem obscured the lower filesystem.

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

Now that you can talk UFS, let's explore ZFS.

12
THE Z FILE SYSTEM

Most filesystems are, in computing terms, ancient. We discard 5-year-old hardware because it's painfully slow, but we format the replacement's hard drive with a 40-year-old filesystem. While we've improved those filesystems and made them more robust, they still use the same basic architecture. And every time a filesystem breaks, we curse and scramble to fix it while desperately wishing for something better.

ZFS is something better.

It's not that ZFS uses revolutionary technology. All the individual pieces of ZFS are well understood. There's no mystery to hashes or data trees or indexing. But ZFS combines all of these well-understood principles into a single cohesive, well-engineered whole. It's designed with the future in mind. Today's hashing algorithm won't suffice 15 years from now, but ZFS is designed so that new algorithms and techniques can be added to newer versions without losing backward compatibility.

This chapter won't cover all there is to know about ZFS. ZFS is almost an operating system on its own, or perhaps a special-purpose database. Entire books have been written about using and managing ZFS. You'll learn enough about how ZFS works to use it on a server, though, and understand its most important features.

While ZFS expects to be installed directly on a disk partition, you can use other GEOM providers as ZFS storage. The most common example is when you do an install with encrypted disks. FreeBSD puts a geli(8) geom on the disk and installs ZFS atop that geom. This chapter calls any storage provider a "disk," even though it could be a file or an encrypted provider or anything

else.

If you've never worked with ZFS before, install a ZFS-based FreeBSD system on a virtual machine and follow along. The installer automatically handles prerequisites, like setting `zfs_load=YES` in *loader.conf* and `zfs_enable=YES` in *rc.local*; all you need concern yourself with is the filesystem.

WHAT DOES ZFS STAND FOR?

The *Z File System*. Yes, seriously. Once upon a time, it meant *Zettabyte File System*, but that acronym has been retconned away.

ZFS blends a whole bunch of well-understood technologies into a combination volume manager and filesystem. It expects to handle everything from the permissions on a file down to tracking which blocks on which storage provider get which information. As the sysadmin, you tell ZFS which hardware you have and how you want it configured, and ZFS takes it from there.

ZFS has three main components: datasets, pools, and virtual devices.

Datasets

A *dataset* is defined as a named chunk of ZFS data. The most common dataset resembles a partitioned filesystem, but ZFS supports other types of datasets for other uses. A snapshot (see "Snapshots" on page 271) is a dataset. ZFS also includes block devices for virtualization and iSCSI targets, clones, and more; all of those are datasets. This book focuses on filesystem datasets. Traditional filesystems like UFS have a variety of small programs to manage filesystems, but you manage all ZFS datasets with zfs(8).

View your existing datasets with `zfs list`. The output looks a lot like mount(8).

```
  # zfs list
  NAME                    USED  AVAIL  REFER  MOUNTPOINT
❶ zroot                 4.71G   894G    88K  none
```

```
❷ zroot/ROOT              2.40G   894G    88K  none
❸ zroot/ROOT/2018-11-17      8K   894G  1.51G  /
❹ zroot/ROOT/default      2.40G   894G  1.57G  /
❺ zroot/usr               1.95G   894G    88K  /usr
❻ zroot/usr/home           520K   894G   520K  /usr/home
  --snip--
```

Each line starts with the dataset name, starting with the storage pool—or *zpool*—that the dataset is on. The first entry is called *zroot* ❶. This entry represents the pool's *root dataset*. The rest of the dataset tree dangles off this dataset.

The next two columns show the amount of space used and available. The pool *zroot* has used 4.71GB and has 894GB available. While the available space is certainly correct, the 4.71GB is more complicated than it looks. The amount of space a dataset shows under USED includes everything on that dataset *and* on all of its children. A root dataset's children include all the other datasets in that zpool.

The REFER column is special to ZFS. This column shows the amount of data accessible on this specific dataset, which isn't necessarily the same as the amount of space used. Some ZFS features, such as snapshots, share data between themselves. This dataset has used 4.71GB of data but refers to only 88KB. Without its children, this dataset has only 88KB of data on it.

At the end, we have the dataset's mount point. This root dataset doesn't have a mount point; it's not mounted.

Look at the next dataset, *zroot/ROOT* ❷. This is a dataset created for the root directory and associated files. That seems sensible, but if you look at the REFER column, you'll see it also has only 88KB of data inside it, and there's no mount point. Shouldn't the root directory exist?

The next two lines explain why . . . sort of. The dataset *zroot/ROOT/2018-11-17* ❸ has a mountpoint of /, so it's a real root directory. The next dataset, *zroot/ROOT/default* ❹, also has a mountpoint of /. No, ZFS doesn't let you mount multiple datasets at the same mount point. A ZFS dataset records a whole bunch of its settings within the dataset. The mount point is one of those settings.

Consider these four datasets for a moment. The *zroot/ROOT* dataset is a child of the zroot dataset. The *zroot/ROOT/2018-11-17* and

zroot/ROOT/default datasets are children of *zroot/ROOT*. Each dataset has its children's space usage billed against it.

Why do this? When you boot a FreeBSD ZFS host, you can easily choose between multiple root directories. Each bootable root directory is called a *boot environment*. Suppose you apply a patch and reboot the system, but the new system won't boot. By booting into an alternate boot environment, you can easily access the defective root directory and try to figure out the problem.

The next dataset, *zroot/usr* ❺, is a completely different child of *zroot*. It has its own child, *zroot/usr/home* ❻. The space used in *zroot/usr/home* gets charged against *zroot/usr*, and both get charged against its parent, but their allocation doesn't affect *zroot/ROOT*.

Dataset Properties

Beyond some accounting tricks, datasets so far look a lot like partitions. But a partition is a logical subdivision of a disk, filling very specific LBAs on a storage device. Partitions have no awareness of the data on the partition. Changing a partition means destroying the filesystem on it.

ZFS tightly integrates the filesystem and the lower storage layers. It can dynamically divide storage space between the various filesystems as needed. Where partitions control the number of available blocks to constrain disk usage, datasets can use quotas for the same effect. Without those quotas, though, if a pool has space, you can use it.

The amount of space a dataset can use is a ZFS *property*. ZFS supports dozens of properties, from the `quotas` property that controls how large a dataset can grow to the `mounted` property that shows whether a dataset is mounted.

Viewing and Changing Dataset Properties

Use `zfs set` to change properties.

```
# zfs set quota=2G zroot/usr/home
```

View a property with `zfs get`. You can either specify a particular property or use `all` to view all properties. You can list multiple properties by

separating them with commas. If you specify a dataset name, you affect only that dataset.

```
# zfs get mounted zroot/ROOT
NAME       PROPERTY  VALUE    SOURCE
zroot/ROOT  mounted     no    -
```

Here, we have the dataset's name, the property, the property value, and something called source. (We'll talk about that last one in "Property Inheritance" on page 261.)

My real question is, which dataset is mounted as the root directory? I could check the two datasets with a mount point of /, but when I get dozens of boot environments, that will drive me nuts. Check a property for a dataset and all of its children by adding the `-r` flag.

```
# zfs get -r mounted zroot/ROOT
NAME                                PROPERTY  VALUE    SOURCE
zroot/ROOT                           mounted     no    -
zroot/ROOT/2018-11-17                mounted     no    -
zroot/ROOT/default                   mounted   ❶yes    -
```

Of the three datasets, only *zroot/ROOT/default* ❶ is mounted. That's our active boot environment.

Property Inheritance

Many properties are inheritable. You set them on the parent dataset and they percolate down through the children. Inheritance doesn't make sense for properties like mount points, but it's right for certain more advanced features. While we'll look at what the `compression` property does in "Compression" on page 273, we'll use it as an example of inheritance here.

```
# zfs get compression
NAME                    PROPERTY     VALUE     SOURCE
zroot                   compression  lz4       local
zroot/ROOT              compression  lz4       inherited from zroot
zroot/ROOT/2018-11-17   compression  lz4       inherited from zroot
zroot/ROOT/default      compression  lz4       inherited from zroot
zroot/tmp               compression  lz4       inherited from zroot
--snip--
```

The root dataset, zroot, has the `compression` property set to lz4. The source is local, meaning that this property is set on this dataset. Now look at

zroot/ROOT. The `compression` property is also lz4, but the source is inherited from zroot. This dataset inherited this property setting from its parent.

Managing Datasets

ZFS uses datasets much as traditional filesystems use partitions. Manage datasets with zfs(8). You'll want to create, remove, and rename datasets.

Create Datasets

Create datasets with `zfs create`. Create a filesystem dataset by specifying the pool and the dataset name. Here, I create a new dataset for my packages. (Note that this breaks boot environments, as we'll see later this chapter.)

```
# zfs create zroot/usr/local
```

Each dataset must have a parent dataset. A default FreeBSD install has a *zroot/usr* dataset, so I can create a *zroot/usr/local*. I'd like to have a dataset for */var/db/pkg*, but while FreeBSD comes with a *zroot/var* dataset, there's no *zroot/var/db*. I'd need to create *zroot/var/db* and then *zroot/var/db/pkg*.

Note that datasets are stackable, just like UFS. If I have files in my */usr/local* directory and I create a dataset over that directory, ZFS will mount the dataset over the directory. I will lose access to those files. You must shuffle files around to duplicate existing directories.

Destroying and Renaming Datasets

That new *zroot/usr/local* dataset I created? It hid the contents of my */usr/local* directory. Get rid of it with `zfs destroy` and try again.

```
# zfs destroy zroot/usr/local
```

The contents of */usr/local* reappear. Or, I could rename that dataset instead, using `zfs rename`.

```
# zfs rename zroot/usr/local zroot/usr/new-local
```

I like boot environments, though, so I'm going to leave */usr/local* untouched. Sometimes you really need a */usr/local* dataset, though . . .

Unmounted Parent Datasets

As a Postgres user, I want a separate dataset for my Postgres data. FreeBSD's Postgres 9.6 package uses */var/db/pgsql/data96*. I can't create that dataset without having a dataset for */var/db*, and I can't have *that* without breaking boot environment support for packages. What to do?

The solution is to create a dataset for */var/db*, but not to use it, by setting the `canmount` dataset property. This property controls whether or not a dataset can be mounted. FreeBSD uses an unmounted dataset for */var* for exactly this reason. New datasets automatically set `canmount` to `on`, so you normally don't have to worry about it. Use the `-o` flag to set a property at dataset creation.

```
# zfs create -o canmount=off zroot/var/db
```

The dataset for */var/db* exists, but it can't be mounted. Check the contents of your */var/db* directory to verify everything's still there. You can now create a dataset for */var/db/postgres* and even */var/db/pgsql/data96*.

```
# zfs create zroot/var/db/postgres
# zfs create zroot/var/db/postgres/data96
# chown -R postgres:postgres /var/db/postgres
```

You have a dataset for your database, and you still have the files in */var/db* itself as part of the root dataset. Now initialize your new Postgres database and go!

As you explore ZFS, you'll find many situations where you might want to set properties at dataset creation or use unmounted parent datasets.

Moving Files to a New Dataset

If you need to create a new dataset for an existing directory, you'll need to copy the files over. I recommend you create a new dataset with a slightly different name, copy the files to that dataset, rename the directory, and then rename the dataset. Here, I want a dataset for */usr/local*, so I create it with a different name.

```
# zfs create zroot/usr/local/pgsql-new
```

Copy the files with tar(1), exactly as you would for a new UFS partition

(see Chapter 11).

```
# tar cfC - /usr/local/pgsql . | tar xpfC - /usr/local/pgsql-new
```

Once it finishes, move the old directory out of the way and rename the dataset.

```
# mv /usr/local/pgsql /usr/local/pgsql-old
# zfs rename zroot/usr/local/pgsql-new zroot/usr/local/pgsql
```

My Postgres data now lives on its own dataset.

ZFS Pools

ZFS organizes its underlying storage in pools, rather than by disk. A ZFS storage pool, or *zpool*, is an abstraction of the underlying storage devices, letting you separate the physical medium and the user-visible filesystem on top of it.

View and manage a host's ZFS pools with zpool(8). Here, I use `zpool list` to see the pools from one of my hosts.

```
# zpool list
NAME      SIZE  ALLOC   FREE  EXPANDSZ   FRAG    CAP  DEDUP  HEALTH  ALTROOT
zroot     928G  4.72G   923G         -     0%     0%  1.00x  ONLINE  -
jail      928G  2.70G   925G         -     0%     0%  1.00x  ONLINE  -
scratch   928G  5.94G   922G         -     0%     0%  1.00x  ONLINE  -
```

This host has three pools: *zroot*, *jail*, and *scratch*. Each has its own line.

The `SIZE` column shows us the total capacity of the pool. All of these pools can hold 928GB. The `ALLOC` column displays how much of each pool is in use, while `FREE` shows how much space remains. These disks are pretty much empty, which makes sense as I installed this host only about three hours ago.

The `EXPANDSZ` column shows whether the underlying storage providers have any free space. When a pool has virtual device redundancy (which we'll discuss in the next section), you can replace individual storage devices in the pool and make the pool larger. It's like swapping out the 5TB drives in your RAID array with 10TB drives to make it bigger.

The `FRAG` column shows how much fragmentation this pool has. You've heard over and over that fragmentation slows performance. ZFS minimizes

the impact of fragmentation, though.

The CAP column shows what percentage of the available space is used.

The DEDUP column shows whether this pool uses deduplication. While many people trumpet deduplication as a ZFS feature, it's not as useful as you might hope.

The HEALTH column displays whether the pool is working well or the underlying disks have a problem.

Pool Details

You can get more detail on pools, or on a single pool, by running `zpool status`. If you omit the pool name, you'll see this information for all of your pools. Here, I check the status of my *jail* pool.

```
# zpool status jail
  pool: jail
 state: ONLINE
  scan: none requested
config:

        NAME                STATE     READ WRITE CKSUM
        jail                ONLINE       0     0     0
          mirror-0          ONLINE       0     0     0
            gpt/da2-jail    ONLINE       0     0     0
            gpt/ada2-jail   ONLINE       0     0     0

errors: No known data errors
```

We start with the pool name. The state is much like the HEALTH column; it displays any problems with the pool. The scan field shows information on scrubs (see "Pool Integrity and Repair" on page 273).

We then have the pool configuration. The configuration shows the layout of the virtual devices in the pool. We'll dive into that when we create our pools.

Pool Properties

Much like datasets, zpools have properties that control and display the pool's settings. Some properties are inherently informational, such as the `free` property that expresses how much free space the pool has. You can change others.

Viewing Pool Properties

To view all of a pool's properties, use `zpool get`. Add the property `all` to view every property. You can add a pool name to include only that pool.

```
# zpool get all zroot
NAME   PROPERTY       VALUE                          SOURCE
zroot  size           928G                           -
zroot  capacity       0%                             -
zroot  health         ONLINE                         -
zroot  guid           7955546176707282768            default
--snip--
```

Some of this information gets pulled into commands like `zpool status` and `zpool list`. You can also query for individual properties across all pools by using the property name.

```
# zpool get readonly
NAME     PROPERTY  VALUE   SOURCE
zroot    readonly  off     -
jail     readonly  off     -
scratch  readonly  off     -
```

Unlike dataset properties, most pool properties are set when you create or import the pool.

Virtual Devices

A *virtual device (VDEV)* is a group of storage devices. You might think of a VDEV as a RAID container: a big RAID-5 presents itself to the operating system as a huge device, even though the sysadmin knows it's really a bunch of smaller disks. The virtual device is where ZFS's magic happens. You can arrange pools for different levels of redundancy or abandon redundancy and maximize space.

ZFS's automated error correction takes place at the VDEV level. Everything in ZFS, from znodes (index nodes) to data blocks, is checksummed to verify integrity. If your pool has sufficient redundancy, ZFS will notice that data is damaged and restore it from a good copy. If your pool lacks redundancy, ZFS will notify you that the data is damaged and you can restore from backup.

A zpool consists of one or more identical VDEVs. The pool stripes data

across all the VDEVs, with no redundancy. The loss of a VDEV means the loss of the pool. If you have a pool with a whole bunch of disks, make sure to use redundant VDEVs.

VDEV Types and Redundancy

ZFS supports several different types of VDEV, each differentiated by the degree and style of redundancy they offer. The common mirrored disk, where each disk copies what's on another disk, is one type of VDEV. Piles of disks with no redundancy is another type of VDEV. And ZFS includes three different varieties of sophisticated parity-based redundancy, called *RAID-Z*.

Using multiple VDEVs in a pool creates systems similar to advanced RAID arrays. A RAID-Z2 array looks an awful lot like RAID-6, but a ZFS pool with two RAID-Z2 VDEVs resembles RAID-60. Mirrored VDEVs work like RAID-1, but multiple mirrors in a pool behave like RAID-10. In both of these cases, ZFS stripes the data across the VDEV with no redundancy. The individual VDEVs provide the redundancy.

Choose your VDEV type carefully.

Striped VDEVs

A VDEV composed of a single disk is called a *stripe* and has no redundancy. Losing the disk means losing your data. While a pool can contain multiple striped VDEVs, each disk is its own VDEV. Much like RAID-0, losing one disk means losing the whole pool.

Mirror VDEVs

A mirror VDEV stores a complete copy of all the VDEV's data on every disk. You can lose all but one of the drives in the VDEV and still access your data. A mirror can contain any number of disks.

ZFS can read data from all of the mirrored disks simultaneously, so reading data is fast. When you write data, though, ZFS must write that data to all of the disks simultaneously. The write isn't complete until the slowest disk finishes. Write performance suffers.

RAID-Z

RAID-Z spreads data and parity information across all of the disks, much

like conventional RAID. If a disk in a RAID-Z dies or starts giving corrupt data, RAID-Z uses the parity information to recalculate the missing data. A RAID-Z VDEV must contain at least three disks and can withstand the loss of any single disk. RAID-Z is sometimes called *RAID-Z1*.

You can't add or remove disks in a RAID-Z. If you create a five-disk RAID-Z, it will remain a five-disk RAID-Z forever. Don't go thinking you can add an additional disk to a RAID-Z for more storage. You can't.

If you're using disks over 2TB, there's a nontrivial chance of a second drive failing as you repair the first drive. For large disks, you should probably consider RAID-Z2.

RAID-Z2

RAID-Z2 stripes parity and data across every disk in the VDEV, much like RAID-Z1, but doubles the amount of parity information. This means a RAID-Z2 can withstand the loss of up to two disks. You can't add or remove disks from a RAID-Z2. It is slightly slower than RAID-Z.

A RAID-Z2 must have four or more disks.

RAID-Z3

Triple parity is for the most important data or those sysadmins with a whole bunch of disks and no time to fanny about. You can lose up to three disks in your RAID-Z3 without losing data. As with any other RAID-Z, you can't add or remove disks from a RAID-Z3.

A RAID-Z3 must have five or more disks.

Log and Cache VDEVs

Pools can improve performance with special-purpose VDEVs. Only adjust or implement these if performance problems demand them; don't add them proactively.[1] Most people don't need them, so I won't go into details, but you should know they exist in case you get unlucky.

The *Separate Intent Log (SLOG* or *ZIL)* is ZFS's filesystem journal. Pending writes get dumped to the SLOG and then arranged more properly in the primary pool. Every pool dedicates a chunk of disk space for a SLOG, but you can use a separate device for the SLOG instead. You need faster writes? Install a really fast drive and dedicate it to the SLOG. The pool will

dump all its initial writes to the fast disk device and then migrate those writes to the slower media as time permits. A dedicated fast SLOG will also smooth out bursty I/O.

The *Level 2 Adaptive Replacement Cache (L2ARC)* is like the SLOG but for reads. ZFS keeps the most recently accessed and the most frequently accessed data in memory. By adding a really fast device as an L2ARC, you expand the amount of data ZFS can provide from cache instead of calling from slow disk. An L2ARC is slower than memory but faster than the slow disk.

RAID-Z and Pools

You can add VDEVs to a pool. You can't add disks to a RAID-Z VDEV. Think about your storage needs and your hardware before creating your pools.

Suppose you have a server that can hold 20 hard drives, but you have only 12 drives. You create a single RAID-Z2 VDEV out of those 12 drives, thinking that you'll add more drives to the pool later if you need them. You haven't even finished installing the server, and already you've failed.

You can add multiple identical VDEVs to a pool. If you create a pool with a 12-disk VDEV, and the host can hold only another 8 disks, there's no way to create a second identical VDEV. A 12-disk RAID-Z2 isn't identical to an 8-disk RAID-Z2. You can force ZFS to accept the different VDEVs, but performance will suffer. Adding a VDEV to a pool is irreversible.

Plan ahead. Look at your physical gear. Decide how you will expand your storage. This 20-drive server would be fine with two 10-disk RAID-Z2 VDEVs, or one 12-disk pool and a separate 8-disk pool. Don't sabotage yourself.

Once you know what sort of VDEV you want to use, you can create a pool.

Managing Pools

Now that you understand the different VDEV types and have indulged in planning your storage, let's create some different types of zpools. Start by setting your disk block size.

ZFS and Disk Block Size

Chapter 10 covered how modern disks have two different sector sizes, 512 bytes and 4KB. While a filesystem can safely assume a disk has 4KB sectors, if your filesystem assumes the disk has 512-byte sectors and the disk really has 4KB sectors, your performance will plunge. ZFS, of course, assumes that disks have 512-byte sectors. If your disk really has 512-byte sectors, you're good. If you're not sure what size the physical sectors are, though, err on the side of caution and tell ZFS to use 4KB sectors. Control ZFS's disk sector assumptions with the *ashift* property. An ashift of 9 tells ZFS to use 512-byte sectors, while an ashift of 12 indicates 4KB sectors. Control ashift with the sysctl `vfs.zfs.min_auto_ashift`.

```
# sysctl vfs.zfs.min_auto_ashift=12
```

Make this permanent by setting it in */etc/sysctl.conf*.

You *must* set ashift before creating a pool. Setting it after pool creation has no effect.

If you're not sure what size sectors your disks have, use an ashift of 12. That's what the FreeBSD installer does. You'll lose a small amount of performance, but using an ashift of 9 on 4KB disks will drain system performance.

Now create your pools.

Creating and Viewing Pools

Create a pool with the zpool `create` command.

```
# zpool create poolname vdevtype disks...
```

If the command succeeds, you get no output back.

Here, I create a pool named *db*, using a mirror VDEV and two GPT-labeled partitions:

```
# zpool create db mirror gpt/zfs3 gpt/zfs4
```

The structure we assign gets reflected in the pool status.

```
# zpool status db
```

```
--snip--
config:

        NAME          STATE     READ WRITE CKSUM
        db            ONLINE       0     0     0
        ❶ mirror-0    ONLINE       0     0     0
          ❷ gpt/zfs3  ONLINE       0     0     0
          ❸ gpt/zfs4  ONLINE       0     0     0
--snip--
```

The pool db contains a single VDEV, named *mirror-0* ❶. It includes two partitions with GPT labels, */dev/gpt/zfs3* ❷ and */dev/gpt/zfs* ❸. All of those partitions are online.

If you don't include a VDEV name, zpool(8) creates a striped pool with no redundancy. Here, I create a striped pool called *scratch*:

```
# zpool create scratch gpt/zfs3 gpt/zfs4
```

The pool status shows each VDEV, named after the underlying disk.

```
--snip--
        NAME        STATE     READ WRITE CKSUM
        garbage     ONLINE       0     0     0
          gpt/zfs3  ONLINE       0     0     0
          gpt/zfs4  ONLINE       0     0     0
--snip--
```

Creating any type of RAID-Z looks much like creating a mirror. Just use the correct VDEV type.

```
# zpool create db raidz gpt/zfs3 gpt/zfs4 gpt/zfs5
```

The pool status closely resembles that of a mirror, but with more disks in the VDEV.

Multi-VDEV Pools

When you're creating a pool, the keywords `mirror`, `raidz`, `raidz2`, and `raidz3` all tell zpool(8) to create a new VDEV. Any disks listed after one of those keywords goes into creating a new VDEV. To create a pool with multiple VDEVs, you'd do something like this:

```
# zpool create poolname vdevtype disks... vdevtype disks...
```

Here, I create a pool containing two RAID-Z VDEVs, each with three disks:

```
# zpool create db raidz gpt/zfs3 gpt/zfs4 gpt/zfs5 raidz gpt/zfs6 gpt/zfs7 gpt/zfs8
```

A zpool status on this new pool will look a little different.

```
--snip--
        NAME          STATE     READ WRITE CKSUM
        db            ONLINE       0     0     0
        ❶ raidz1-0    ONLINE       0     0     0
            gpt/zfs3  ONLINE       0     0     0
            gpt/zfs4  ONLINE       0     0     0
            gpt/zfs5  ONLINE       0     0     0
        ❷ raidz1-1    ONLINE       0     0     0
            gpt/zfs6  ONLINE       0     0     0
            gpt/zfs7  ONLINE       0     0     0
            gpt/zfs8  ONLINE       0     0     0
--snip--
```

This pool contains a VDEV called `raidz1-0` ❶ with three disks in it. There's a second VDEV, named `raidz1-1` ❷, with three disks in it. It's very clear that these are identical pools. Data gets striped across both VDEVs.

Destroying Pools

To destroy a pool, use `zpool destroy` and the pool name.

```
# zpool destroy db
```

Note that zpool doesn't ask whether you're really sure before destroying the pool. Being sure you want to destroy the pool is your problem, not zpool(8)'s.

Errors and -f

If you enter a command that doesn't make sense, zpool(8) will complain.

```
# zpool create db raidz gpt/zfs3 gpt/zfs4 gpt/zfs5 raidz gpt/zfs6 gpt/zfs7
invalid vdev specification
use '-f' to override the following errors:
mismatched replication level: both 3-way and 2-way raidz vdevs are present
```

The first thing you see when reading the error message is "use `-f` to override this error." Many sysadmins read this as "`-f` makes this problem go away." What ZFS is really saying, though, is "Your command line is a horrible mistake. Add `-f` to do something unfixable, harmful to system stability, and that you'll regret as long as this system lives."

Most zfs(8) and zpool(8) error messages are meaningful, but you have to read them carefully. If you don't understand the message, fall back on the troubleshooting instructions in Chapter 1. Often, reexamining what you typed will expose the problem.

In this example, I asked zpool(8) to create a pool with a RAID-Z VDEV containing three disks and a second RAID-Z VDEV containing only two disks. I screwed up this command line. Adding `-f` and proceeding to install my database to the new malformed *db* pool would only ensure that I have to recreate this pool and reinstall the database at a later date.[2] If you find yourself in this situation, investigate `zfs send` and `zfs recv`.

Copy-On-Write

In both ordinary filesystems and ZFS, files exist as blocks on the disk. When you edit a file in a traditional filesystem, the filesystem picks up the block, modifies it, and sets it back down in the same place on the disk. A system problem halfway through that write can cause a *shorn write*: a file that's 50 percent the old version, 50 percent the new version, and probably 100 percent unusable.

ZFS never overwrites the existing blocks in a file. When a file changes, ZFS identifies the blocks that must change and writes them to a new chunk of disk space. The old version is left intact. This is called *copy-on-write (COW)*. With copy-on-write, a short write might lose the newest changes to the file, but the previous version of the file will remain intact.

Never corrupting files is a great benefit to copy-on-write, but COW opens up other possibilities. The metadata blocks are also copy-on-write, all the way up to the *uberblocks* that form the root of the ZFS pool's data tree. ZFS creates snapshots by tracking the blocks that contain old versions of a file. While that sounds simple, the details are what will lead you astray.

Snapshots

A snapshot is a copy of a dataset as it existed at a specific instant. Snapshots are read-only and never change. You can access the contents of a snapshot to access older versions of files or even deleted files. While snapshots are read-only, you can roll the dataset back to the snapshot. Take a snapshot before upgrading a system, and if the upgrade goes horribly wrong, you can fall back to the snapshot. ZFS uses snapshots to provide many features, such as boot environments (see "Boot Environments" on page 276). Best of all, depending on your data, snapshots can take up only tiny amounts of space.

Every dataset has a bunch of metadata, all built as a tree from a top-level block. When you create a snapshot, ZFS duplicates that top-level block. One of those metadata blocks goes with the dataset, while the other goes with the snapshot. The dataset and the snapshot share the data blocks within the dataset.

Deleting, modifying, or overwriting a file on the live dataset means allocating new blocks for the new data and disconnecting blocks containing the old data. Snapshots need some of those old data blocks, however. Before discarding an old block, ZFS checks to see whether a snapshot still needs it. If a snapshot needs a block, but the dataset no longer does, ZFS keeps the block.

So, a snapshot is merely a list of which blocks the dataset used at the time the snapshot was taken. Creating a snapshot tells ZFS to preserve those blocks, even if the dataset no longer needs those blocks.

Creating Snapshots

Use the `zfs snapshot` command to create snapshots. Specify the dataset by its full path, then add `@` and a snapshot name. I habitually name my snapshots after the date and time I create the snapshot, for reasons that will become clear by the end of this chapter.

I'm about to do maintenance on user home directories, removing old stuff to free up space. I'm pretty sure that someone will whinge about me removing their files,[3] so I want to create a snapshot before cleaning up.

```
# zfs snapshot zroot/usr/home@2018-07-21-13:09:00
```

I don't get any feedback. Did anything happen? View all your snapshots with the `-t snapshot` argument to `zfs list`.

```
# zfs list -t snapshot
NAME                                  USED  AVAIL    REFER     MOUNTPOINT
zroot/usr/home@2018-07-21-13:09:00     ❶0     ❷-    ❸4.68G   ❹-
```

The snapshot exists. The USED column shows that it uses zero disk space ❶: it's identical to the dataset it came from. As snapshots are read-only, available space ❷ shown by AVAIL is just not relevant. The REFER column shows that this snapshot pulls in 4.68GB of disk space ❸. If you check, you'll see that's the size of *zroot/usr/home*. Finally, the MOUNTPOINT column shows that this snapshot isn't mounted ❹.

This is an active system, and other people are logged into it. I wait a moment and check my snapshots again.

```
# zfs list -t snapshot
NAME                                  USED  AVAIL  REFER  MOUNTPOINT
zroot/usr/home@2018-07-21-13:09:00    ❶96K      -  4.68G  -
```

The snapshot now uses 96KB ❶. A user changed something on the dataset, and the snapshot gets charged with the space needed to maintain the difference.

Now I go on my rampage, and get rid of the files I think are garbage.

```
# zfs list -t snapshot
NAME                                  USED  AVAIL  REFER  MOUNTPOINT
zroot/usr/home@2018-07-21-13:09:00    1.62G      -  4.68G  -
```

This snapshot now uses 1.62GB of space. Those are files that I've deleted but that are still available in the snapshot. I'll keep this snapshot for a little while to give the users a chance to complain.

Accessing Snapshots

Every ZFS dataset has a hidden *.zfs* directory in its root. It won't show up in ls(1); you have to know it exists. That directory has a snapshot directory, which contains a directory named after each snapshot. The contents of the snapshot are in that directory.

For our snapshot *zroot/usr/home@2018-07-21-13:09:00*, we'd go to */usr/home/.zfs/snapshot/2018-07-21-13:09:00*. While the *.zfs* directory doesn't show up in ls(1), once you're in it, ls(1) works normally. That directory contains every file as it existed when I created the snapshot, even if I've deleted or changed that file since creating that snapshot.

Recovering a file from the snapshot requires only copying the file from the snapshot to a read-write location.

Destroying Snapshots

A snapshot is a dataset, just like a filesystem-style dataset. Remove it with `zfs destroy`.

```
# zfs destroy zroot/usr/home@2017-07-21-13:09:00
```

The space used by the snapshot is now available for more junk files.

Compression

Snapshots aren't the only way ZFS can save space. ZFS uses on-the-fly compression, transparently inspecting the contents of each file and squeezing its size if possible. With ZFS, your programs don't need to compress their log files: the filesystem will do it for you in real time. While FreeBSD enables compression by default at install time, you'll use it more effectively if you understand how it works.

Compression changes system performance, but probably not in the way you think it would. You'll need CPU time to compress and decompress data as it goes to and from the disk. Most disk requests are smaller than usual, however. You essentially exchange processor time for disk I/O. Every server I manage, whether bare metal or virtual, has far, far more processor capacity than disk I/O, so that's a trade I'll gleefully make. The end result is that using ZFS compression most often *increases* performance.

Compression works differently on different datasets. Binary files are already pretty tightly compressed; compressing */usr/bin* doesn't save much space. Compressing */var/log*, though, often results in reducing file size by a factor of six or seven. Check the property `compressratio` to see how effectively compression shrinks your data. My hosts write to logs far more often than

they write binaries. I'll gleefully accept a sixfold performance increase for the most common task.

ZFS supports many compression algorithms, but the default is *lz4*. The lz4 algorithm is special in that in quickly recognizes incompressible files. When you write a binary to disk, lz4 looks at it and says, "Nope, I can't help you," and immediately quits trying. This eliminates pointless CPU load. It effectively compresses files that can be compressed, however.

Pool Integrity and Repair

Every piece of data in a ZFS pool has an associated cryptographic hash stored in its metadata to verify integrity. Every time you access a piece of data, ZFS recomputes the hash of every block in that data. When ZFS discovers corrupt data in a pool with redundancy, it transparently corrects that data and proceeds. If ZFS discovers corrupt data in a pool without redundancy, it gives a warning and refuses to serve the data. If your pool has identified any data errors, they'll show up in `zpool status`.

Integrity Verification

In addition to the on-the-fly verification, ZFS can explicitly walk the entire filesystem tree and verify every chunk of data in the pool. This is called a *scrub*. Unlike UFS's fsck(8), scrubs happen while the pool is online and in use. If you've previously run a scrub, that will also show up in the pool status.

```
  scan: scrub repaired 0 in 8h3m with 0 errors on Fri Jul 21 14:17:29 2017
```

To scrub a pool, run `zpool scrub` and give the pool name.

```
# zpool scrub zroot
```

You can watch the progress of the scrub with `zpool status`.

Scrubbing a pool reduces its performance. If your system is already pushing its limits, scrub pools only during off hours. You can cancel a scrub[4] with the `-s` option.

```
# zpool scrub -s zroot
```

Run another scrub once the load drops.

Repairing Pools

Disks fail. That's what they're for. The point of redundancy is that you can replace failing or flat-out busted disks with working disks and restore redundancy.

Mirror and RAID-Z virtual devices are specifically designed to reconstruct the data lost when a disk fails. They're much like RAID in that regard. If one disk in a ZFS mirror dies, you replace the dead disk, and ZFS copies the surviving mirror onto the new disk. If a disk in a RAID-Z VDEV fails, you replace the busted drive, and ZFS rebuilds the data on that disk from parity data.

In ZFS, this reconstruction is called *resilvering*. Like other ZFS integrity operations, resilvering takes place only on live filesystems. Resilvering isn't quite like rebuilding a RAID disk from parity, as ZFS leverages its knowledge of the filesystem to optimize repopulating the replacement device. Resilvering begins automatically when you replace a failed device. ZFS resilvers at a low priority so that it doesn't interfere with normal operations.

Pool Status

The `zpool status` command shows the health of the underlying storage hardware in the `STATE` field. We've seen a couple examples of healthy pools, so let's take a look at an unhealthy pool.

```
# zpool status db
  pool: db
 state: ❶DEGRADED
status: One or more devices could not be opened.  Sufficient replicas exist for
        the pool to continue functioning in a degraded state.
action: Attach the missing device and online it using 'zpool online'.
   see: http://illumos.org/msg/ZFS-8000-2Q
  scan: none requested
config:

        NAME                        STATE     READ WRITE CKSUM
        db                          DEGRADED     0     0     0
        ❷mirror-0                    DEGRADED     0     0     0
            gpt/zfs1                ONLINE       0     0     0
```

```
        ❸14398195156659397932   ❹UNAVAIL      0     0     0   ❺was /dev/gpt/zfs3

errors: No known data errors
```

The pool state is DEGRADED ❶. If you look further down the output, you'll see more DEGRADED entries and an UNAVAIL ❹. What exactly does that mean?

Errors in a pool percolate upward. The pool state is a summary of the health of the pool as a whole. The whole pool shows up as DEGRADED because the pool's virtual device *mirror-0* ❷ is DEGRADED. This error comes from an underlying disk being in the UNAVAIL state. We get the ZFS GUID ❸ for this disk, and the label used to create the pool ❺.

ZFS pools show an error when an underlying device has an error. When a pool has a state other than ONLINE, dig through the VDEV and disk listings until you find the real problem.

Pools, VDEVs, and disks can have six states:

ONLINE The device is functioning normally.

DEGRADED The pool or VDEV has at least one provider missing, offline, or generating errors more quickly than ZFS tolerates. Redundancy is handling the error, but you need to address this right now.

FAULTED A faulted disk is corrupt or generating errors more quickly than ZFS can tolerate. A faulted VDEV takes the last known good copy of the data. A two-disk mirror with two bad disks faults.

UNAVAIL ZFS can't open the disk. Maybe it's been removed, shut off, or that iffy cable finally failed. It's not there, so ZFS can't use it.

OFFLINE This device has been deliberately turned off.

REMOVED Some hardware detects when a drive is physically removed while the system is running, letting ZFS set the REMOVED flag. When you plug the drive back in, ZFS tries to reactivate the disk.

Our missing disk is in the UNAVAIL state. For whatever reason, ZFS can't access */dev/gpt/zfs3*, but the disk mirror is still serving data because it has a working disk. Here's where you get to run around to figure out where that disk went. How you manage ZFS depends on what you discover.

Reattaching and Detaching Drives

Unavailable drives might not be dead. They might be disconnected. If you wiggle a drive tray and suddenly get a green light, the disk is fine but the connection is faulty. You should address that hardware problem, yes, but in the meantime, you can reactivate the drive. You can also reactivate deliberately removed drives. Use the `zpool online` command with the pool name and the GUID of the missing disk as arguments. If the disk in my example pool were merely disconnected, I could reactivate it like so:

```
# zpool online db 14398195156659397932
```

ZFS resilvers the drive and resumes normal function.

If you want to remove a drive, you can tell ZFS to take it offline with zpool offline. Give the pool and disk names as arguments.

```
# zpool offline db gpt/zfs6
```

Bringing disks offline, physically moving them, bringing them back online, and allowing the pools to resilver will let you migrate large storage arrays from one SAS cage to another without downtime.

Replacing Drives

If the drive isn't merely loose but flat-out busted, you'll need to replace it with a new drive. ZFS lets you replace drives in several ways, but the most common is using `zpool replace`. Use the pool name, the failed provider, and the new provider as arguments. Here, I replace the db pool's */dev/gpt/zfs3* disk with */dev/gpt/zfs6*:

```
# zpool replace db gpt/zfs3 gpt/zfs6
```

The pool will resilver itself and resume normal operation.

In a large storage array, you can also use successive `zpool replace` operations to empty a disk shelf. Only do this if your organization's operation requirements don't allow you to offline and online disks.

Boot Environments

ZFS helps us cope with one of the most dangerous things sysadmins do. No, not our eating habits. No, not a lack of exercise. I'm talking about system upgrades. When an upgrade goes well, everybody's happy. When the upgrade goes poorly, it can ruin your day, your weekend, or your job. Nobody likes restoring from backup when the mission-critical software chokes on the new version of a shared library. Nobody likes to restore from backup.

Through the magic of *boot environments*, ZFS takes advantage of snapshots to let you fall back from a system upgrade with only a reboot. A boot environment is a clone of the root dataset. It includes the kernel, the base system userland, the add-on packages, and the core system databases. Before running an upgrade, create a boot environment. If the upgrade goes well, you're good. If the upgrade goes badly, though, you can reboot into the boot environment. This restores service while you investigate how the upgrade failed and what you can do to fix those problems.

Boot environments do not work when a host requires a separate boot pool. The installer handles boot pools for you. They appear when combining UEFI and GELI, or when using ZFS on an MBR-partitioned disk.

Using boot environments requires a boot environment manager. I recommend beadm(8), available as a package.

```
# pkg install beadm
```

You're now ready to use boot environments.

Viewing Boot Environments

Each boot environment is a dataset under *zroot/ROOT*. A system where you've just installed beadm should have only one boot environment. Use `beadm list` to view them all.

```
  # beadm list
  BE        Active    Mountpoint  Space   Created
❶default  ❷NR      ❸/          ❹2.4G   ❺2018-05-04 13:13
```

This host has one boot environment, named *default* ❶, after the dataset *zroot/ROOT/default*.

The Active column ❷ shows whether this boot environment is in use. An `N` means that the environment is now in use. An `R` means that this environment will be active after a reboot. They appear together when the default environment is running.

The Mountpoint column ❸ shows the location of this boot environment's mount point. Most boot environments aren't mounted unless they're in use, but you can use beadm(8) to mount an unused boot environment.

The Space column ❹ shows the amount of disk space this boot environment uses. It's built on a snapshot, so the dataset probably has more data than this amount in it.

The Created column ❺ shows the date this boot environment was created. In this case, it's the date the machine was installed.

Before changing the system, create a new boot environment.

Creating and Accessing Boot Environments

Each boot environment needs a name. I recommend names based on the current operating system version and patch level or the date. Names like "beforeupgrade" and "dangitall," while meaningful in the moment, will only confuse you later.

Use `beadm create` to make your new boot environment. Here, I check the current FreeBSD version, and use that to create the boot environment name:

```
# freebsd-version
11.0-RELEASE-p11
# beadm create 11.0-p11
Created successfully
```

I now have two identical boot environments.

```
# beadm list
BE  a       Active Mountpoint  Space Created
default    NR     /           12.3G 2015-04-28 11:53
11.0-p11   -      -          236.0K 2018-07-21 14:57
```

You might notice that the new boot environment already takes up 236KB. This is a live system. Between when I created the boot environment and when I listed those environments, the filesystem or its metadata changed.

The Active column shows that we're currently using the default boot environment and that we'll be using that on the next boot. If I change my installed packages or upgrade the base system, those changes will affect the default environment.

Each boot environment is available as a snapshot under *zroot/ROOT*. If you want to access a boot environment read-write, use `beadm mount` to temporarily mount the boot environment under */tmp*. Unmount those environments with `beadm umount`.

Activating Boot Environments

Suppose you upgrade your packages and the system goes belly-up. Fall back to an earlier operating system install by activating a boot environment and rebooting. Activate a boot environment with `beadm activate`.

```
# beadm activate 11.0-p11
Activated successfully
# beadm list
BE         Active Mountpoint  Space Created
default    N      /           12.4G 2015-04-28 11:53
11.0-p11   R      -          161.8M 2018-07-21 14:57
```

The default boot environment has its Active flag set to `N`, meaning it's now running. The 11.0-p11 environment has the `R` flag, so after a reboot it will be live.

Reboot the system and suddenly you've fallen back to the previous operating system install, without the changes that destabilized your system. That's much simpler than restoring from backup.

Removing Boot Environments

After a few upgrades, you'll find that you'll never fall back to some of the existing boot environments. Once I upgrade this host to, say, 12.2-RELEASE-p29, chances are I'll never ever reboot into 11.0-p11 again. Remove obsolete boot environments and free up their disk space with `beadm destroy`.

```
# beadm destroy 11.0-p11
Are you sure you want to destroy '11.0-p11'?
This action cannot be undone (y/[n]): y
Destroyed successfully
```

Answer y when prompted, and beadm will remove the boot environment.

Boot Environments at Boot

So you've truly hosed your operating system. Forget getting to multiuser mode, you can't even hit *single*-user mode without generating a spew of bizarre error messages. You can select a boot environment right at the loader prompt. This requires console access, but so would any other method of rescuing yourself.

The boot loader menu includes an option to select a boot environment. Choose that option. You'll get a new menu listing every boot environment on the host by name. Choose your new boot environment and hit ENTER. The system will boot into that environment, giving you a chance to figure out why everything went sideways.

Boot Environments and Applications

It's not enough that your upgrade failed. It might take your application data with it.

Most applications store their data somewhere in the root dataset. MySQL uses */var/db/mysql*, while Apache uses */usr/local/www*. This means that falling back to an earlier boot environment can revert your application data with the environment. Depending on your application, you might want that reversion —or not.

If an application uses data that shouldn't be included in the boot environment, you need to create a new dataset for that data. I provided an example in "Unmounted Parent Datasets" on page 262 earlier this chapter. Consider your application's need and separate out your data as appropriate.

While ZFS has many more features, this covers the topics every sysadmin *must* know. Many of you would find clones, delegations, or replication useful. You might find the books *FreeBSD Mastery: ZFS* (Tilted Windmill Press, 2015) and *FreeBSD Mastery: Advanced ZFS* (Tilted Windmill Press, 2016) by Allan Jude and yours truly helpful. You'll also find many resources on the internet documenting all of these topics.

Now let's consider some other filesystems FreeBSD administrators find useful.

13
FOREIGN FILESYSTEMS

FreeBSD supports a variety of filesystems other than ZFS and UFS. You'll need to be able to interoperate with other hosts by using optical media, flash drives, and the like. Additionally, FreeBSD uses the special-purpose filesystem devfs(5) to manage device nodes. Jail users might need the process filesystem procfs(5). For extremely fast storage that doesn't need to survive a reboot, you can use system RAM as a filesystem. You can mount filesystems over the network, using either the Unix-style Network File System or Microsoft's Common Internet File System (CIFS). And no matter how hard you try to avoid it, sometimes you're stuck mounting ISO images.

Using any of these requires a deeper understanding of mounting filesystems.

FreeBSD Mount Commands

We saw mount(8) earlier when discussing UFS filesystems, but you'll also use it to attach other filesystems to the directory tree. The mount(8) command assumes that any local partitions use UFS. If you try to mount a non-UFS filesystem, you'll get an error.

```
# mount /dev/cd0 /media
mount: /dev/cd0: Invalid argument
```

The device node */dev/cd0* represents an optical drive. I put a CD in the drive just for this test, so it should work. Trying to mount it gives an error, though. To mount a UFS filesystem, you need the device node and a mount

point. Mounting foreign filesystems means adding the filesystem type with `-t`. CDs use the ISO 9660 filesystem, which FreeBSD calls cd9660. Here, I specify the filesystem to mount that CD on */cdrom*:

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

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

Many filesystems have their own custom variant of the mount(8) command. Get a full list by running `apropos mount_`. Yes, you need the trailing underscore; all of the mount(8) variants use that as a separator. You'll find mount_cd9660(8), mount_msdosfs(8), mount_nfs(8), and more. Always use `mount -t` on filesystems without such a command.

Apply mount options with the `-o` flag. You'll need to check each mount command's man page to see what mount options the filesystem supports. Separate multiple mount options with commas. Here, I mount a FAT32 drive at device node */dev/da1* read-only, assigning the owner and group to user *bert*:

```
# mount -t msdosfs -o ro,-gbert,-ubert /dev/da1 /media
```

You can unmount any mounted filesystem with umount(8):

```
# umount /media
```

The umount(8) command doesn't care about the filesystem type. It just tries to disconnect the disk partition from the filesystem. It does care about whether someone is using the filesystem, however, and refuses to unmount it if even one process uses it. If you have an idle terminal with a shell prompt in the filesystem, umount(8) will refuse to unmount the filesystem.

If you're constantly connecting and disconnecting filesystems, investigate autofs(5) to handle these mounts automatically.

Supported Foreign Filesystems

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

FAT (MS-DOS)

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

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

ISO 9660

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

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

UDF

UDF, or Universal Disk Format, is a replacement for ISO 9660. You'll find UDF on some DVDs and Blu-Ray disks and on a few thumb drives larger than the 32GB supported under FAT32. As the capacity of removable media increases, you'll see more and more UDF filesystems. The mount command is `mount -t udf`.

EXT

The standard Linux filesystems—EXT2, EXT3, and EXT4—support many of the same features as UFS. FreeBSD can safely read from and write to EXT2 and EXT3 filesystems without any problems but can mount EXT4 filesystems only as read-only.

Mounting Linux filesystems is most useful for disaster recovery, dual-

boot systems, or system migrations. Despite the name, `mount -t ext2fs` supports mounting all versions of EXT.

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

Permissions and Foreign Filesystems

Permissions of a filesystem depend on the filesystem features and the person who mounts it. FreeBSD tries to support features that aren't too different from those in UFS or ZFS.

Consider the Linux filesystem, EXT. EXT stores permissions in the filesystem and lets the kernel map them to UIDs. Since EXT permissions behave much like UFS permissions and all the necessary permissions information is available within the filesystem, FreeBSD respects the permissions on these filesystems. EXT doesn't support BSD file flags, however, so you can't assign those flags to a file on EXT.

FAT has no permissions system. Even if you mount your FAT32 thumb drive in your FreeBSD host, you can't apply permissions to files.

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

```
# mount -t msdosfs -u xistence -g xistence /dev/da5 /mnt
```

The user *xistence* now owns the files on the device.

You might get sick of mounting media for your users, especially in a facility with dozens of machines. To let users mount filesystems, set the sysctl `vfs.usermount` to `1`. Users can then mount any device they have permission to access on any mount point they own. While *xistence* couldn't mount the removable device on */media*, he could mount it on */home/xistence/media*.

Using Removable Media

You must be able to manage any removable media that might wander in through the door of your data center. Here, we'll discuss optical disks and flash drives.

I recommend not plugging removable media willy-nilly into your production servers—for security reasons if nothing else. Who knows what's actually on that vendor's USB device? Worse, you can order "USB killer" devices that deliberately damage hardware. Mount suspicious devices on a disposable workstation, examine the contents, and then copy the desired data over to the FreeBSD machine. This isn't guaranteed to be safe, as the many USB interfaces can inject data into the hardware beneath the OS layer, but it's as safe as you'll get. Removable media is just too easy for certain applications, however, and of course the rules change when it's my personal USB device.

Using the device requires a filesystem type, a device node, and a mount point.

Figuring out a removable drive's filesystem can require a bit of trial and error. CDs use the ISO 9660 filesystem, while DVDs and Blu-Rays use either a UDF or a combination of ISO 9660 and UDF. When in doubt, try CD9660 first. USB devices and floppy disks are usually FAT32. While it was once expected that large USB devices would use UDF, most of them still use FAT32. Run fstyp(8) on a device node to help identify the filesystem on it, or try `gpart show` on the disk's device node.

Removable devices can have a different device node each time you plug them in. Optical drives are a little easier to identify in that most hosts have very few optical drives. If you have one optical drive, it's */dev/cd0*. USB devices appear as the next available unit of */dev/da*. When you insert a USB device, a message giving the device node and type appears on the console and in */var/log/messages*, or you could check `camcontrol devlist` for the new device.

FreeBSD provides a */media* mount point for general *removable media* mounts. You can create additional mount points as you like—they're just directories. For miscellaneous short-term mounts, FreeBSD offers */mnt*.

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

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

Occasionally, you'll find a thumb drive with a partition table. These devices will insist you mount */dev/da0s1* or */dev/da0p1* rather than */dev/da0*. The device's formatting dictates this, not anything in FreeBSD. The `gpart show` command can help you figure out which partitions are on a device and what filesystem is on each partition.

Ejecting Removable Media

To disconnect removable media from your FreeBSD system, first unmount the filesystem. Your optical drive won't open until you unmount the disk. You can pull a USB flash drive from its port, but doing so while the filesystem is mounted might damage data on the device. Use umount(8) just as you would for any other filesystem:

```
# umount /media
```

On many optical drives, `camcontrol eject` opens the drive tray.

Removable Media and /etc/fstab

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

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

Here's an */etc/fstab*'s entry for an optical drive:

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

While I'm sure you've already memorized the meaning of every column in */etc/fstab*, we'll remind you that this entry means, "Mount */dev/cd0* on */cdrom*, using the ISO 9660 filesystem. Mount it as read-only, and don't mount it automatically at boot."

Here's a similar entry for a thumb drive. I use the `large` option to support

filesystems larger than 128GB, as discussed in mount_msdosfs(8).

```
❶/dev/da0    /media    msdosfs  rw,noauto,large    0      0
```

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

Formatting FAT32 Media

Thumb drives use the FAT32 filesystem but always come preformatted. As thumb drives have a limited number of reads and writes directly proportional to their cheapness, do not reformat them capriciously.[1] Only reformat thumb drives when their filesystem becomes corrupt. Use newfs_msdos(8) to create a FAT32 filesystem.

```
# newfs_msdos /dev/da0
```

You'll get a couple lines of output, and you have a new filesystem.

Creating Optical Media

FreeBSD will let you bundle up a bunch of files into an image suitable for burning onto a CD, DVD, or Blu-Ray, using either CD 9660 or UDF formats. You can burn either image onto disk. FreeBSD supports creating ISOs natively, but you'll need programs from the `cdrtools` package to create UDF.

In either case, start by putting all of the files and directories you want to burn into a single directory. The image will contain these files and directories exactly as you arrange them. Remember, optical disk images are read-only. You can't update an image; you can only create a new image, so be sure you have everything exactly as you want it. Later this chapter, you'll learn to mount these images with mdconfig(8).

In both of these examples, we're creating an image from the files contained in */home/xistence/cdfiles*.

Creating ISOs

Use makefs(8) to create an ISO.

```
# makefs ❶-t cd9660 ❷-o allow-deep-trees,rockridge ❸image.iso ❹source-files
```

Start by using `-t` ❶ to specify the type of filesystem to create—in this case, CD 9660. The `-o` flag ❷ lets you specify filesystem-specific options. You can get a whole list of options from the makefs(8) man page, but the ones shown here suffice for most images. We then need the filename ❸ for the created image and the source directory ❹ for those files.

To make an image containing the files in */home/xistence/cdfiles* as *bert.iso*, run:

```
# makefs -t cd9660 -o allow-deep-trees,rockridge bert.iso /home/xistence/cdfiles
```

Bert can now wastefully burn his ISO to physical media.

Creating a UDF

Creating a UDF requires using mkisofs(1) from the `cdrtools` package. Give the destination image file with `-o`. Enable the Joliet and Rock Ridge extensions with `-J` and `-R`, respectively. (I'm not going to go into what each of these do, but if you want your ISO to behave like a disk from this millennium, you need them.) Add the `-udf` and `-iso-level 3` flags.[2]

```
# mkisofs -R -J -udf -iso-level 3 -o bert.udf /home/xistence/cdfiles
```

You now have a UDF image based on what's in */home/xistence/cdfiles*.

Whichever format you create, I encourage you to mount it and double-check your work before burning a physical disk. If you're lucky, you'll remember the stuff you forgot to include on the image.

Burning ISOs to Optical Media

Use cdrecord(1) from the `cdrtools` package to burn ISO images to the disk. Give the image file as an argument.

```
# cdrecord bert.iso
```

Depending on the drive speed and image size, this might take a while.

The cdrecord(1) program defaults to using */dev/cd0*. If you have additional optical drives, use the `-dev` flag to give an alternate device name.

```
# cdrecord -dev=cd9 bert.iso
```

You now have a flimsy plastic disk that you'll use twice before flinging it into the landfill. Congratulations!

Burning UDF to Optical Media

While you *can* use cdrecord(1) to burn UDF images to media, the growisofs(1) command from the `dvd+rw-tools` package is generally recommended. You'll need the `-dvd-compat` and `-Z` flags. Then, specify the device and the image file.

```
# growisofs -dvd-compat -Z /dev/burner=image.udf
```

Suppose I want to burn *bert.udf* to the Blu-Ray in */dev/cd0*.

```
# growisofs -dvd-compat -Z /dev/cd0=bert.udf
```

UDF files can be huge. Go make some tea. Eventually, you'll have a burned disk.

Writing Images to Thumb Drives

USB thumb drives have increasingly supplanted optical disks, thanks in part to their reusability. FreeBSD supports writing disk images to thumb drives with dd(1).

Be very certain which device node is your thumb drive and which is your system hard drive. Thumb drives show up as */dev/da* devices, exactly like many hard drives. Overwriting the wrong hard drive is embarrassing.[3]

The dd(1) command looks confusing at first glance.

```
# dd ❶if=inputfile ❷of=outputdevice ❸bs=1M ❹conv=sync
```

The `if=` argument ❶ gives the file you want to copy. The `of=` argument ❷ is the device node to copy to. The `bs=` flag ❸ gives the amount to copy at one

time. Without this, dd(1) copies in 512-byte increments. The `conv=` argument ❹ gives dd(1) instructions about how to convert the incoming file. In this case, *sync* tells dd(1) to synchronize the size of the incoming and outgoing buffer. To burn *bertimage.udf* to thumb drive */dev/da9*, I would run:

```
# dd if=bert.udf of=/dev/da9 bs=1m conv=sync
```

Wait a bit, and you'll have an imaged thumb drive. Other uses of dd(1) might not need the `conv=` flag, but always use `bs`.

Now let's look at some other filesystems you might find useful.

Memory Filesystems

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

FreeBSD supports two different memory-backed 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 better suited for short-term use or mounting disk images.

tmpfs

The *tmp* in tmpfs(5) doesn't mean "temporary." It literally means *tmp*, as in */tmp*. Use tmpfs for a speedy memory-backed */tmp* and similar filesystems. Don't deploy tmpfs everywhere you see a path with *tmp* in it, though. While */tmp* is supposed to be cleared at every boot, */var/tmp* is supposed to survive a reboot. You might use tmpfs for application lock files and other ephemeral data where vastly increased speed would improve application performance. While tmpfs has a troubled history, as of FreeBSD 10, it's widely deployed and considered ready for production.

Create a tmpfs by mounting it.

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

If your system has the sysctl `vfs.usermount` set to 1, users can create and mount tmpfs filesystems.

tmpfs Options

A tmpfs defaults to the size of the system's available RAM plus the available swap space. Repeatedly copying a file to */tmp* could exhaust system memory. This would be bad. Set a maximum size for your tmpfs with the `size` option.

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

Control the ownership and permissions on a tmpfs with the `uid`, `gid`, and `mode` options. An actual */tmp* directory needs to 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, even an unprivileged user that runs only a single application, assign that user ownership of the tmpfs.

tmpfs at Boot

Now that you can set a maximum size and the proper permissions, it's okay to use */etc/fstab* to automatically create a tmpfs at boot.

```
tmpfs  /tmp  tmpfs  rw,mode=1777,size=1G  0  0
```

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 isn't 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 or some other backing store. No matter what, the memory disk disappears at system shutdown.

Memory Disk Types

Memory disks come in four types: malloc-backed, swap-backed, vnode-

backed, and null.

Malloc-backed memory disks are pure memory. Even if your system runs short on memory, FreeBSD won't swap out the malloc-backed disk. Much like tmpfs(5), 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 memory disks are mostly memory, but they also access the system swap partition. If the system runs out of memory, it moves the least recently used parts of memory to swap, as discussed in Chapter 21. Swap-backed disks are usually the best compromise between speed and performance.

Vnode-backed memory disks are files on disk. While you can use a file as backing for your memory disk, this is mostly useful for mounting disk images and testing.

A *null* memory disk discards everything sent to it. Any writes are successful, while any reads return zero. If I didn't mention null memory disks, someone would write to complain, but I'm not giving a disk guaranteed to lose all data any more coverage than this.

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

Creating and Mounting Memory Disks

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

Swap-backed memory disks are the default. Just tell mdmfs(8) the size of the disk and the mount point. Here, we create a 48MB swap-backed memory disk on */home/mwlucas/test*:

```
# mdmfs -t -s 48m md /home/mwlucas/test
```

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

The `-t` flag enables TRIM, which we'll discuss in the following section, "Memory Disk Headaches."

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

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

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

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

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

Memory Disk Headaches

Traditional swap-backed memory disks never returned used memory to the system. Once you wrote to a memory disk, that memory was used up. If you needed a larger memory disk, you had to permanently allocate memory for it. This was one reason FreeBSD included tmpfs(5).

If the filesystem on the memory disk supports *TRIM*, however, FreeBSD now returns unused memory to the system. TRIM is not an acronym but rather a protocol for telling a disk which sectors are no longer in use. UFS, the default memory disk format, supports TRIM. Enable TRIM in mdmfs with the `-t` flag. If you're using a different filesystem on a memory disk, though, be sure it's strictly temporary.

To free the memory from a memory disk, shut down the memory disk.

Memory Disk Shutdown

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

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

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

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

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

Memory Disks and /etc/fstab

If you list memory disks in */etc/fstab*, FreeBSD automatically creates them at boot time. These entries look more complicated than the other entries but aren't too bad if you understand the mdmfs(8) commands we've been using so far.

We're allowed to use *md* as a device name to indicate a memory disk. Choose the mountpoint just as for any other device, and use the filesystem type `mfs`. Under `Options`, list *rw* (for read-write) and the command line options used to create this device. If this is a long-term mount, add `-t` to enable TRIM. To create our 48MB filesystem mounted at */home/mwlucas/test*, use the following */etc/fstab* entry:

```
md      /home/mwlucas/test      mfs      rw,-s48m,-t      0   0
```

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

Mounting Disk Images

You can use mdmfs(8) to view UFS disk images, but most often you want to examine the contents of an ISO or UDF file without burning it to disk. (FreeBSD's tar(1) can access the contents of an ISO, but not a UDF.) Just attach a memory disk to a file with the mdconfig(8) command's `-a` flag. Here, I attach Bert's ISO to a memory device:

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

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

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

I can now verify that the ISO contains Bert's files, so he doesn't get to whine that the ISO is busted.

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

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

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

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

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

I have an ISO image mounted on this system? Wow. I should probably reboot some month. Nah, that's too much work; I'll just unmount the filesystem and destroy the memory device.

Filesystems in Files

One trick used in embedded systems is building complete filesystem images on a local file. In the previous section, we saw how we could use memory disks to mount and access CD disk images. You can use the same techniques to create, update, and access UFS disk images.

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

Creating an Empty Filesystem File

Use truncate(1) to create an empty file for a filesystem. These files are sparse files: they're labeled as having a certain size but don't actually take up any space until you put something in them. An empty sparse file takes up one filesystem block and grows when you put stuff in it. This means you can create an image for a disk of any size but use up only an amount of space equal to the stuff you put in the image.

Use the `-s` option and the file size to create an image file. Here, I create a 1GB file:

```
# truncate -s 1G filesystem.file
```

The resulting file claims to be pretty large.

```
# ls -l filesystem.file
-rw-r--r--  1 mwlucas  mwlucas  1073741824 Aug 11 11:31 filesystem.file
```

But if you check the disk usage, you'll see something different.

```
# du filesystem.file
1       filesystem.file
```

This 1GB file uses one block on the disk.

Sparse files never shrink. They can only grow. If you erase a bunch of files from your disk image, the image file still needs that space.

Also, not all filesystems support sparse files. UFS and ZFS do. If you're trying to create a sparse file on a FAT32 filesystem, you're probably solving the wrong problem.

Creating the Filesystem on the File

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

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

Now, let's make a filesystem on this device. This is much like creating a UFS filesystem on a thumb disk with the newfs(8) command. Soft updates journaling is exactly as useful on file-backed filesystems as on disk-backed ones, so enable them with `-j`.

```
# newfs -j /dev/md0
/dev/md0: 1024.0MB (2097152 sectors) block size 32768, fragment size 4096
        using 4 cylinder groups of 256.03MB, 8193 blks, 32896 inodes.
        with soft updates
super-block backups (for fsck_ffs -b #) at:
 192, 524544, 1048896, 1573248
Using inode 4 in cg 0 for 8388608 byte journal
newfs: soft updates journaling set
```

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

Now that you have a filesystem, mount it:

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

Congratulations! You now have a 1GB file-backed filesystem. Copy files to it, dump it to tape, or use it in any way you would use any other filesystem. But in addition to that, you can move it just like any other file.

File-Backed Filesystems and /etc/fstab

You can mount a file-backed filesystem automatically at boot with the proper entry in */etc/fstab*, much like you can automatically mount any other memory disk. You simply have to specify the name of the file with `-F` and use `-P` to tell the system not to create a new filesystem on this file but just to use the one already there. Here, we mount the file-backed filesystem we created on */mnt* automatically at boot time.

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

I told you we'd see */etc/fstab* entries uglier than the one for generic memory disks, didn't I?

devfs

devfs(5) is a dynamic filesystem for managing device nodes. Remember, in a Unix-like operating system, *everything* is a file. This includes physical hardware. Almost all devices on the system have a device node under */dev*. You've seen a bunch of device nodes for disks, but you'll also see keyboards (*/dev/ukbd0* or */dev/kbd0*), the console (*/dev/console*), sound mixers (*/dev/mixer0*), and more. You'll also find device nodes for logical devices, like the random number generator (*/dev/random*), terminal sessions (*/dev/ttyv0*), and so on.

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

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

/dev at Boot

When device nodes were permanent files on disk, the sysadmin could symlink to those nodes or change their permissions without worrying that his changes would vanish. With an automated, dynamic device filesystem, this assurance disappears. (Of course, you no longer have to worry about

occult mknod(8) commands either, so you're better off in the long run.) The device node changes could include, for example:

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

DEVICE MANAGEMENT AND SERVERS

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

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

devfs.conf

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

```
action      realdevice      desiredvalue
```

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

```
❶link       ❷cd0            ❸cdrom
```

We want a symbolic link ❶ to the device node */dev/cd0* ❷ (an optical

drive), and we want this link to be named */dev/cdrom* ❸. If we reboot with this entry in */etc/devfs.conf*, our optical drive */dev/cd0* also appears as */dev/cdrom*, as many desktop multimedia programs expect.

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

```
perm        cd0            666
```

Here, we set the permissions on */dev/cd0* (our CD device, again) so that any system user can read or write to the device. Remember, changing the permissions on the */dev/cdrom* link won't change the permissions on the device node, just the symlink.

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

```
own         da20            xistence:xistence
```

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

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

Global devfs Rules

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

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

devfs Ruleset Format

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

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

The first rule in *devfs.rules* is called `devfs_hide_all` ❶ and is ruleset number 1 ❷. This ruleset contains only one rule ❸.

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

```
devfs_system_rulesets="laptoprules"
```

Remember, devfs rules apply to the devices in the system at boot and the devices configured dynamically after startup.

Ruleset Content

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

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

This rule assigns the user *mwlucas* ownership of all device nodes with a node name beginning with `da`. This is probably a bad idea.

Devices specified by path use standard shell regular expressions. If you want to match a variety of devices, use an asterisk as a wildcard. For example, `path ada1s1` matches exactly the device */dev/ada1s1*, but `path ada*s*` matches every device node with a name beginning with `ada`, a character, the letter `s`, and possibly more characters. You could tell exactly what devices are

matched by a wildcard by using it at the command line.

```
# ls /dev/ada*s*
```

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

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

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

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

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

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

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

The `hide` keyword lets you make device nodes disappear, and `unhide` makes them reappear. Since no program can use a device node if the device is invisible, this is of limited utility except when the system uses jail(8). Hiding and unhiding makes the most sense when including rules in rules.

Including Rules in Rules

As in so many parts of systems administration, making devfs rules modular so they can be reused is a good way to reduce problems. The default jail rules show exactly how FreeBSD's devfs supports reuse, through the `include` keyword.

Here's the start of the default configuration:

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

```
❸ [devfsrules_unhide_basic=2]
❹ add path log unhide
❺ add path null unhide
❻ add path zero unhide
❼ add path crypto unhide
   --snip--
```

Rule number one, `devfsrules_hide_all` ❶, conceals all device nodes ❷.

Rule number two, `devfsrules_unhide_basic` ❸, contains only a series of `unhide` statements. This rule does nothing but unhide critical Unix device nodes, like */dev/log* ❹, */dev/null* ❺, */dev/zero* ❻, */dev/crypto* ❼, and so on. Most processes won't run without these devices. These device nodes are already exposed in a standard system, so why would you need a rule just to unhide them? Similarly, ruleset number three, `devfsrules_unhide_login`, does nothing but unhide device nodes for logged-in users.

The last ruleset leverages all of these.

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

This ruleset, `devfsrules_jail`, uses `include` statements to pull in the previous rulesets by reference. The last statement also unhides */dev/zfs*, allowing ZFS tools to work within jails.

If you want to make additional device nodes available within all of your jails, you could add that device node to the jails ruleset. Or you could define a new ruleset and use it for all your jails. Better still, you could define a ruleset for just the jails that absolutely need that device and assign that ruleset to those jails.

To finish up, let's look at dynamic devices.

Dynamic Device Management with devd(8)

Hot-swappable hardware is now routine. FreeBSD's devfs dynamically creates new device nodes when this hardware is plugged in and erases the nodes when the hardware is removed, making using these dynamic devices

much simpler. The devd(8) daemon takes this a step further by letting you run userland programs when hardware appears and disappears.

FreeBSD's default configuration, */etc/devd.conf*, handles most modern hardware just fine. If you need to customize devd(8), put your configuration files under */usr/local/etc/devd/* to simplify upgrades. You could also add different rules files for different types of devices if you find your devd(8) configuration becoming very complicated.

devd Configuration

You'll find four types of devd(8) rules: `attach`, `detach`, `nomatch`, and `notify`.

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

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

The `nomatch` rules are triggered when new hardware is installed but not attached to a device driver. These devices don't have device drivers in the current kernel.

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

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

```
❶notify ❷0 {
        match "system"          ❸"IFNET";
        match "subsystem"       ❹"!usbus[0-9]";
        match "type"            ❺"ATTACH";
        action ❻"/etc/pccard_ether $subsystem start";
};
```

This is a `notify` rule ❶, which means it activates when the kernel sends a

message to userland. As a priority 0 rule ❷, this rule can be triggered only if no rule of higher priority matches the criteria we specify. This rule is triggered only if the notification is on the network system `IFNET` ❸ (network) and only if the subsystem ❹ doesn't match the expression `usbus[0-9]`. It excludes USB network cards. The notification type is `ATTACH` ❺—in other words, this matches only when someone plugs in a network interface. If all three of these matches hit, devd(8) runs a command to configure the network interface ❻.

Read the devd(8) man page to see about all the options you can put in rules. If you want to automatically mount a particular USB flash disk on a certain mount point, you can do that by checking the serial number of every USB device you put in. If you want to configure Intel network cards differently than Atheros network cards, you can do that by checking the vendor. Whatever you need to write a rule for, it's probably in there somewhere.

Miscellaneous Filesystems

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

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

If you're using Linux mode (see Chapter 17), you might need the Linux process filesystem linprocfs(5). Much Linux software requires a process filesystem, and FreeBSD suggests installing linprocfs at */compat/linux/proc* when you install Linux mode. I'd recommend installing linprocfs only if a piece of software complains it's not there.

The file descriptor filesystem fdesc(5) offers a filesystem view of file descriptors for each process. Some software, notably Java and the popular Bash shell, requires fdescfs(5). It's less of a security risk than procfs, but still

undesirable. You'll get instructions on mounting fdescfs(5) when you install a package that requires it.

Now that we've talked about local filesystems, let's look at the network.

The Network File System

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

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

NFS wasn't 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 completely inadequate—you must prevent random hosts from poking at the host's remote procedure call (RPC) services. Restrict access to the host by IP address as well as port number.

Additionally, standard NFS isn't encrypted. Anyone with a packet sniffer and access to your wire can see all filesystem activity. Once you deploy Kerberos, you can encrypt NFS, but Kerberos requires its own book.

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

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

NFS INTEROPERABILITY

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

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

NFS is one of those topics that has entire books written about it. We're not going to go into the intimate details about NFS, but rather focus on getting basic NFS operations working. If you're deploying complicated NFS setups, you'll want to do further research. Even this basic setup lets you accomplish many complicated tasks.

NFS Versions

Modern NFS comes in three versions: NFSv2, NFSv3, and NFSv4. FreeBSD can transparently autodetect and interoperate with versions 2 and 3.

NFSv2 is rather minimal, dating from the time when people were delighted to get file sharing working at all.

NFSv3 contains many incremental improvements over and much better performance than NFSv2. Most of these improvements don't even require special configuration.

NFSv4 is an entirely different and highly complex protocol that breaks many of the long-standing rules of NFS. It was deliberately designed to resemble Microsoft's file sharing. Understanding NFSv4 requires understanding filesystem extended ACLs, synchronizing user IDs across the

network, and other headaches.

When people say "NFS" they almost always mean NFSv2 or NFSv3. Some folks call these protocols "traditional NFS." Someone who means NFSv4 usually says "NFSv4."

This book sticks with the commonly deployed NFSv2 and NFSv3. I devote a couple chapters to NFSv4 and related topics in *FreeBSD Mastery: Specialty Filesystems* (Tilted Windmill Press, 2016).

Configuring the NFS Server

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

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

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

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

The NFS server is designed to seamlessly interoperate a whole bunch of

different NFS implementations. While it should transparently autonegotiate connections, you might find that you need to tweak your NFS server nfsd(8) to best fit your clients. Tune nfsd(8) at startup with the *rc.conf* option `nfs_server_flags`.

NFS can run over TCP or UDP. UDP is the traditional NFS transport protocol. TCP works better over lossy networks and can better cope with irregular network speeds. FreeBSD offers both protocols but defaults to using TCP mounts. Some clients behave better with one protocol or the other. You can explicitly enable only TCP with `-t` and only UDP with `-u`.

The NFS server defaults to listening to all IP addresses on a machine. When a server has multiple IP addresses, replies to a UDP request can come from any of those addresses. This can confuse NFS clients. If your NFS server has multiple IP addresses and you have clients that prefer UDP, tell the NFS server to use only a single address with `-h` and the server IP.

While nfsd(8) works well, highly loaded servers might need additional nfsd(8) processes. While FreeBSD starts four nfsd(8) processes by default, you can start additional processes with the `-n` flag and the desired number of processes.

This *rc.conf* entry tells NFS to use only UDP, bind to the IP address 198.51.100.71, and run six instances of nfsd(8).

```
nfs_server_flags="-uh 198.51.100.71 -n 6"
```

Before you start tweaking server behavior, though, you really should have some exports.

Configuring NFS Exports

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

FreeBSD lets you configure exports through two different paths. The traditional method is the file */etc/exports*. A ZFS-based server can configure exports through each dataset's `sharenfs` property. The server will create the

ZFS exports file */etc/zfs/exports* based on these properties. Both exports files have the same format.

Choose one method of managing your NFS exports. Either edit */etc/exports*, or use zfs(8). Using both methods simultaneously might merely confuse you but will probably break everything. If you use the ZFS method, never edit */etc/zfs/exports* by hand. Stick with one method.

No matter which method you choose, though, */etc/exports* must exist. If you manage NFS through zfs(8), I recommend creating a one-line */etc/exports* that contains only a comment telling people to use zfs(8).

Exports Entries

So how do you configure an export? I'll start with the exports file */etc/exports*, but most everything also applies to using ZFS. I'll discuss the differences in "Managing NFS with zfs(8)" on page 308, but understanding those limitations requires understanding */etc/exports*.

Each exports entry has up to three parts:

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

Each combination of clients and a disk device can only have one line in the exports file. This means that if */usr/ports* and */usr/home* are on the same partition and you want to export both of them to a particular client, they must both appear in the same line. You can't export */usr/ports* and */usr/home* to one client with different permissions. You don't have to export the entire disk device, mind you; you can export a single directory within a partition. This directory cannot contain either symlinks or double or single dots.

NFS mounts don't cross partitions. If a host has separate UFS partitions for */usr* and */usr/src*, exporting */usr* doesn't automatically export */usr/src*.

Of the three parts of the */etc/exports* entry, only the directory is mandatory. An exports line cannot contain symlinks or periods. To export my home directory to every host on the internet, I could use an */etc/exports* line consisting entirely of:

```
/home/mwlucas
```

This has no options and no host restrictions. Such an export would be foolish, of course, but I could do it.[4]

After editing the *exports* file, tell mountd(8) to reread it:

```
# service mountd reload
```

Any problems with mountd(8) appear in */var/log/messages*. The log messages are generally enigmatic: while mountd(8) informs you that a line is bad, it usually doesn't say why. The most common errors I experience involve symlinks. Use pwd(1) in a directory to get a directory's actual path.

NFS and Users

NFSv2 and NFSv3 identify users by UID. (NFSv4 uses usernames because it assumes you've synchronized usernames across the entire network.) For example, on my laptop, the user *mwlucas* has the UID of 1001. On the NFS server, *mwlucas* also has the UID 1001. This makes my life easy, as I don't have to worry too much about file ownership; I have the same privileges on the server as on my laptop.

This can be a problem on a large network, where users have root on their own machines. The best way around this is to create a central repository of authorized users via Kerberos. On a small network or on a network with a limited number of NFS users, this usually isn't a problem; you can synchronize */etc/master.passwd* on your systems or just assign the same UID to each user on each system.

The root user is handled slightly differently, however. An NFS server doesn't trust root on other machines to execute commands as root on the server. After all, if an intruder breaks into an NFS client, you don't want the server to automatically go down with it. NFS defaults to mapping requests from a client's root account to the UID and GID of –2 on the server. This is where the highly unprivileged *nobody* account originated.

The authors of many other server programs thought the nobody account was a great idea, so they appropriated nobody for their own use. Multiple security entities simultaneously running as nobody creates security issues. FreeBSD's packages create unprivileged users for all applications that need one. I consider the nobody user tainted and suggest you don't permit its use.

You can map requests from root to any other username. For example, you

might say that all requests from root on a client will run as the *nfsroot* user on the server. With careful use of groups, you could allow this nfsroot user to have limited file access. Use the `maproot` option to map root to another user. Here, we map UID 0 (root) on the client to UID 5000 on the server:

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

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

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

NFS users can belong to no more than 16 groups. Some operating systems can break that limit, but they violate the NFS protocols in doing so. If a user can't access files with group-based access control, check the number of groups that they're in.

Remember to restart mountd(8) after editing the exports file.

Exporting Multiple Directories

A standard FreeBSD UFS install puts all the files on one partition. You might want to export multiple directories on that partition. List all directories on the same partition on the same line in */etc/exports*, right after the first exported directory, separated by spaces. Here's a sample */etc/exports* with multiple exports:

```
/usr/home/mwlucas /usr/src /var/log /usr/ports/distfiles -maproot=nfsroot
```

Clients can mount any of these directories, and requests from root get mapped to nfsroot.

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

Perhaps you want clients to be able to mount any directory on a partition. Allow this with the `-alldirs` option. I wouldn't do this on a host with a single

partition.

```
/home -alldirs
```

You can only specify a partition mount point with `-alldirs`.

Long Lines

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

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

Once your exports line gets long enough, this style suddenly gets more readable than the alternative.

Restricting Clients

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

```
/usr/home/mwlucas /usr/src /usr/obj /usr/ports/distfiles \
    -maproot=5000 203.0.113.200
```

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

```
/usr/home/mwlucas /usr/src /usr/obj /usr/ports/distfiles \
    -maproot=5000 -network 203.0.113 -mask 255.255.255.0
```

This lets any client with an IP address beginning in 203.0.113 access your NFS server. I use a setup much like this to upgrade clients quickly. I build a new world and kernel on the NFS server and then let the clients mount those partitions and install the binaries over NFS.

To export to an IPv6 network, include the slash in the address.

```
/usr/home/mwlucas -network 2001:db8:bad:c0de::/64
```

You can also list hostnames rather than IP addresses, but this creates a dependency on name resolution. If you lose DNS, you'd lose file sharing. Also, the NFS server looks up the IP address of each host when you start mountd. Changing a client's IP means reloading both DNS and mountd(8). If you must list hostnames, put them at the end of the line.

```
/usr/home/mwlucas www1 www2 www3
```

Assigning NFS on a per-host basis is more labor. Assign NFS permissions as broadly as possible without compromising security.

Combinations of Clients and Exports

Each line in */etc/exports* specifies exports from one partition to one network, address, or set of hosts. Different hosts require entirely different export statements. You can change the options for each if you wish.

```
/usr/home/mwlucas /usr/src /usr/obj /usr/ports/distfiles \
    -maproot=5000 203.0.113.200
/usr -maproot=0 203.0.113.201
```

Here, I've exported several subdirectories of */usr* to the NFS client at 203.0.113.200. The NFS client at 203.0.113.201 gets to mount the whole of */usr* and may even do so as root.

NFS and Firewalls

NFS is famous for not liking firewalls. The dynamic port assignment of services like mountd(8), rpc.lockd(8), and rpc.statd(8) makes packet filtering nearly impossible. You can use the `-p` flag to assign each of these services a specific TCP port. Here, I use *rc.conf* entries to nail mountd(8) to port 4046, rpc.lockd(8) to 4045, and rpc.statd(8) to 4047:

```
mountd_flags="-r -p 4046"
rpc_lockd_flags="-p 4045"
rpc_statd_flags="-p 4047"
```

I can use these ports in my packet filter rules, providing some protection to my NFS server.

Managing NFS with zfs(8)

Using zfs(8) to manage NFS has advantages and disadvantages. You can configure NFS on a per-dataset basis, and you don't need to manually restart mountd(8) after each change. Command line configuration is easier to automate, and many folks find it easier to type 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 in the world can mount this dataset or any of its children, with no restrictions and no options, unless you have other access control, such as a firewall.

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

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

You probably want some NFS options on an export, though. Set `sharenfs` to the desired options for the dataset. This example sets a maproot user and restricts clients to my local network. Put the options in quotes.

```
# 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 all permitted hosts get the same options. That is, if most of your hosts need to mount */home* with `-maproot=nfsroot` but you have one host that needs root to mount that dataset as root, you can't use ZFS properties. Similarly, you can define only one permitted network with ZFS properties.

Enabling the NFS Client

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

```
nfs_client_enable="YES"
```

You can reboot or run `service nfsclient start`. Either starts NFS client functions.

Show Available Mounts

One obvious question for an NFS client to ask would be, "What can I mount from that server?" The showmount(8) command lists all exports available to a client. Give the `-e` flag and the name of the NFS server. Here, I ask the

storm server what exports it offers:

```
# showmount -e storm
Exports list on storm:
/usr/home                          203.0.113.0
```

This client is allowed to mount */usr/home* under the rule that permits the network 203.0.113.0.

Running showmount(8) doesn't offer any server-side options, like `-maproot`. These details aren't readily available to clients, although touch(1) lets you easily test for read-only exports.

Mounting Exports

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

```
# mount storm:/usr/home/mwlucas /mnt
```

Afterward, test your mount with df(1).

```
# df -h
Filesystem            Size    Used   Avail Capacity  Mounted on
--snip--
storm:/usr/home       891G    2.7G    888G     0%    /mnt
```

The NFS-mounted directory shows up as a normal partition, and I can read and write files on it as I please.

NFS Mount Options

FreeBSD uses conservative NFS defaults so that it can interoperate with any other Unix-like operating system. You can use mount options to adjust how FreeBSD mounts NFS exports. Use these options at the command line with `-o` or add them to an */etc/fstab* entry.

If you need to access a UDP-only NFS server, use the mount option `udp` to use UDP rather than the default TCP.

Programs expect the filesystem not to disappear, but when you're using NFS, it's possible that the server will vanish from the network. This makes

programs on the client trying to access the NFS filesystem hang forever. By making your NFS mount *interruptible*, you'll be able to interrupt processes hung on unavailable NFS mount with CTRL-C. Set interruptibility with `intr`.

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

If you want a read-only mount, use the `ro` mount option.

Putting everything together, I might mount my home directory as an interruptible soft mount.[5]

```
# mount -o soft,intr storm:/usr/home/mwlucas /mnt
```

I could add this to */etc/fstab* as follows:

```
storm:/usr/home/mwlucas  /mnt  nfs  rw,soft,intr  0  0
```

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

Now, let's look at reading Windows shares.

The Common Internet File System

If you're on a typical office network, the standard network file sharing protocol is Microsoft's *Common Internet File System (CIFS)*. You might know CIFS as Server Message Block (SMB), "Network Neighborhood," or "Why can't I mount that drive?" While originally provided only by Microsoft Windows systems, this protocol has become something of a pseudostandard.

FreeBSD includes the smbutil(8) program to find, mount, and use CIFS shares as a CIFS client. FreeBSD doesn't include a CIFS server in the base system, but the open source CIFS server Samba (*https://www.samba.org/*) works well on FreeBSD.

Use FreeBSD's CIFS support to interoperate with existing Microsoft infrastructure. Don't deploy CIFS to support Unix-like systems.

Prerequisites

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

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

Kernel Support

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

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

You can load these automatically at boot time with a *boot/loader.conf* entry.

```
smbfs_load=YES
```

You can now configure CIFS.

Configuring CIFS

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

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

Information that applies to an entire server goes into a section named

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

Configuration entries use the values from the CIFS system—for example, Bert's Windows username is *bertjw*, but his FreeBSD username is *xistence*, so I use *bertjw* in *nsmb.conf*.

nsmb.conf Keywords

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

workgroup=string

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

```
workgroup=MegaCorp
```

addr=a.b.c.d

The `addr` keyword sets the IP address of a CIFS server. This keyword can appear only under a plain `[servername]` label. You shouldn't need this if you have working CIFS name resolution, but reality sometimes disagrees.

nbns=a.b.c.d

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

password=string

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

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

```
# smbutil crypt superSecretPassword
$$1624a53302a6d
```

If the server needs access to a CIFS share to do its routine job, don't use your account. Ask the Windows team for an account for your server so that problems with your account won't interrupt the server's functions.

Sample Configuration

Here, I build an *nsmb.conf* allowing Bert access to his files on the corporate CIFS fileserver.

```
[default]
nbns=203.0.113.12
workgroup=BigCorp
[FILESERVER:bertjw]
password=$$1624a53302a6d
```

With this configuration, Bert should be able to access whatever CIFS shares those tyrannical Windows admins permit.

CIFS Name Resolution

Before FreeBSD can mount a CIFS share, it needs to identify the host the share is on. While Microsoft has used DNS for decades now, typical Windows environments often support a whole panoply of legacy protocols. Verify that smbutil(1) can find CIFS servers with `smbutil lookup`.

```
# smbutil lookup fileserver1
Got response from 203.0.113.12
IP address of ntserv1: 203.0.113.4
```

If this works, you have basic CIFS functionality.

Other smbutil(1) Functions

You can view shares on a host at the command line. Start by logging into your host.

```
# smbutil login //unix@fileserver1
Password:
```

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

```
# smbutil view //unix@fileserver1
Password:
Share        Type       Comment
-------------------------------
IPC$         pipe       Remote IPC
ADMIN$       disk       Remote Admin
C$           disk       Default share
unix         disk
4 shares listed from 4 available
```

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

```
# smbutil logout //unix@fileserver
```

Mounting a Share

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

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

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

```
# mount_smbfs //unix@fileserver1/MP3 /home/mwlucas/smbmount
```

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

other filesystem. Use umount(8) to disconnect from the server.

Other mount_smbfs Options

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

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

nsmb.conf Options

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

Unique Password on a Standalone System

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

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

Accessing a Second Domain

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

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

CIFS File Ownership

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

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

Serving CIFS Shares

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

We've now finished our tour of FreeBSD filesystems. While I've spent a few chapters on the topic, FreeBSD has several additional filesystems options, an automounter, and even Filesystem in Userspace (FUSE) support for accessing NTFS, Linux's extfs, and more. It has special iSCSI support and special filesystems like nullfs(5) that make managing jails at scale very powerful. If I spend any more time on filesystems, though, you'll track me down and use a blunt instrument to express your displeasure, so let's proceed to some of FreeBSD's advanced security features.

14
EXPLORING /ETC

The */etc* directory contains the basic configuration information needed to boot a Unix-like system. Every time I get saddled with an unfamiliar system, one of the first things I do is scope out */etc*. The fastest way to go from a junior sysadmin to a midgrade one is to read */etc* and the associated man pages. Yes, all of it. Yes, this is a lot of reading. Understanding */etc* means that you understand how the system hangs together. As you progress as a sysadmin, you're going to pick up this information piecemeal anyway, so you might as well take the easier route and master this part of your toolkit at the beginning.

I discuss many */etc* files in chapters where they're most important, such as */etc/services* in Chapter 7 and */etc/fstab* in Chapter 10. Also, some files are of only historical interest or are gradually being removed. This chapter covers important */etc* files that don't quite fit anywhere else.

/etc Across Unix Species

Different Unix-like systems use different */etc* files. In many cases, these files are simply renamed or restructured files from primordial BSD. The first time I encountered an IBM AIX system, for example, I went looking for a BSD-style */etc/fstab*. It wasn't there. A little hunting led me to */etc/filesystems*, which is an IBM-specific */etc/fstab*. Apparently IBM felt that a file named for an abbreviation of *filesystem table* was confusing, so they renamed the file. Knowing this information existed somewhere in */etc*, and knowing which files it obviously wasn't in, greatly shortened my search.

Even radically different FreeBSD systems have almost identical */etc* directories. While some add-on programs insert their own files here, you can expect certain files to be on every FreeBSD system you encounter.

Remember that */etc* is the heart of FreeBSD and that changes to these files can damage or destroy your system. While having to manually recover a scrambled filesystem can turn an adequate sysadmin into a pretty good one, it's one of the least pleasant ways to get there.

/etc/adduser.conf

This file lets you configure the defaults for new users. See Chapter 9 for details.

/etc/aliases

This file lets you configure system-wide email forwarding. We cover it in Chapter 20.

/etc/amd.map

FreeBSD has the ability to automatically mount and unmount NFS filesystems upon demand through the automounter daemon, amd(8). The automounter daemon is very old, however, and has largely been replaced by autofs(5) and automountd(8). Automounting is mostly useful for workstations, so we won't go into it.

/etc/auto_master

The *auto_master* file configures FreeBSD's modern automounting service. It lets you configure mount options, determine how long an automounted filesystem should remain mounted, and so on. See auto_master(5), autofs(5), and automountd(8) for details.

/etc/blacklistd.conf

FreeBSD includes an automated blacklist daemon, blacklistd(8), that's comparable to fail2ban and similar programs. Programs that link against libblacklist(3) can direct blacklistd(8) to block intrusive hosts at the firewall. We configure blacklistd(8) in Chapter 19.

/etc/bluetooth, /etc/bluetooth.device.conf, and /etc/defaults/bluetooth.device.conf

FreeBSD supports Bluetooth, a standard for short-range wireless communication. Unlike 802.11, Bluetooth is designed for short-range but high-level services, such as voice communications. This book is about servers, so we won't cover Bluetooth, but you should know that your FreeBSD laptop can attach to your Bluetooth-equipped cellphone and connect to the internet if you desire.

/etc/casper

The Capsicum security system lets programmers add sandboxing and security capabilities to their software. The */etc/casper* directory contains sample capsicum(4) configurations.

/etc/crontab and /etc/cron.d

The cron(8) daemon lets users schedule tasks. See Chapter 20 for examples and details.

/etc/csh.*

The */etc/csh.** files contain system-wide defaults for csh and tcsh. When a user logs in with either of these shells, the shell executes any commands it finds in */etc/csh.login*. Similarly, when the user logs out, */etc/csh.logout* is executed. You can place general shell configuration information in */etc/csh.cshrc*.

Per-shell configuration is prone to errors, and you'll have to maintain

identical settings for any other shells. I recommend putting necessary environment variables in a login class (see "Restricting System Usage" on page 188).

/etc/ddb.conf

The kernel debugger configuration utility ddb(8) reads *ddb.conf* for instructions. We'll use this in Chapter 24 to prepare kernel crash dumps on small systems.

/etc/devd.conf

The device daemon devd(8) is best known for managing detachable hardware, such as USB, PCCard, and Cardbus devices. When you insert a USB network card into your laptop, devd(8) notices the arrival and fires up the appropriate system processes to configure the card as per */etc/rc.conf*. More generally, it's a state change daemon and can react on link up/down events, notify you about CPU overheating, process suspend/resume events, and more. We discuss devd(8) briefly in Chapter 13, but if you think you need to edit */etc/devd.conf* on a server, you're probably doing something wrong.

/etc/devfs.conf, /etc/devfs.rules, and /etc/defaults/devfs.rules

FreeBSD manages device nodes through devfs(5), a virtual filesystem that dynamically provides device nodes as hardware boots, appears, and disappears. See Chapter 13 for more information.

/etc/dhclient.conf

Many operating systems give you very basic DHCP client configuration with no way to fine-tune or customize it; you either use it or you don't. In most cases, an empty */etc/dhclient.conf* file gives you full DHCP client functionality, but it won't work correctly in all situations. Perhaps your

network is having trouble or you're at a conference where some script kiddie thinks it's fun to set up a second DHCP server and route everyone's traffic through his machine so he can capture passwords.

Your server better not be configured via DHCP (unless it's diskless), so we won't go into any depth on this. You should be aware that you can configure FreeBSD's DHCP client functionality, however.

/etc/disktab

Once upon a time, hard disks were rare and exotic creatures that came in only a few varieties. In */etc/disktab*, you'll find low-level descriptions of many different kinds of disks, from the 360KB floppy disk to a Panasonic 60MB laptop hard drive. (Yes, laptops came with 60MB hard drives, and we were durned happy to have them.)

Today, this file is mostly used for removable media, such as 1.44MB floppy disks and zip disks. While I described formatting flash drives in Chapter 13, this file contains the descriptions needed to format other removable media. If you want to put a filesystem on your LS 120 disk or zip drive, you'll find the necessary label here at the beginning of an entry.

Editing */etc/disktab* is useful only if you have multiple identical hard drives that you want to partition and format in exactly the same way. If you need to make your own entries, read disktab(5).

/etc/dma/

The Dragonfly Mail Agent (DMA) stores its configuration in */etc/dma/*. We discuss DMA in Chapter 20.

/etc/freebsd-update.conf

This file is used by freebsd-update(8) when getting binary updates for your server. See Chapter 18 for details.

/etc/fstab

See Chapter 10 for a discussion on the filesystem table, */etc/fstab*.

/etc/ftp.*

The FTP daemon ftpd(8) uses these files to determine who may access the system via FTP and what access they have upon a successful connection. Unless you're running a large FTP site, you should be using sftp(1) instead.

/etc/group

Assigning users to groups is covered in painful detail in Chapter 9.

/etc/hostid

Certain software expects every host to have a universally unique ID, or UUID. If you're running on real hardware, this UUID is burned into the mainboard and is accessible via kenv(8). Virtual hosts can generate a UUID from software. The */etc/hostid* file contains that UUID.

/etc/hosts

This file contains host-to-IP mappings, as discussed in Chapter 8.

/etc/hosts.allow

The */etc/hosts.allow* file controls who can access the daemons compiled with TCP Wrappers support. Learn about it in Chapter 19.

/etc/hosts.equiv

The */etc/hosts.equiv* file is used by the r-services (rlogin, rsh, etc.) to let trusted remote systems log in or run commands on the local system without providing a password or even logging in. Hosts listed in this file are assumed to have performed user authentication on a trusted system, so the local

system doesn't have to bother reauthenticating the user.

Such blatant trust is very convenient on friendly networks, much as leaving the doors of your Manhattan townhouse unlocked saves you the trouble of digging out your door keys every time you get home. There's no such thing as a friendly network.[1] A single disgruntled employee can largely destroy a corporate network with this service, and a machine using the r-services is pretty much dog meat for the first script kiddie who wanders by. In fact, */etc/hosts.equiv* and its related services have bitten even top-notch security experts who thought they could use it safely. I suggest leaving this file empty and perhaps even making it immutable (see Chapter 9).

/etc/hosts.lpd

The */etc/hosts.lpd* file is one of the simplest files in */etc*. Hosts listed here, each on its own line, may print to the printer(s) controlled by this machine. While you can use hostnames, that would allow DNS issues to choke printing, so use IP addresses instead.

Unlike most other configuration files, */etc/hosts.lpd* doesn't accept network numbers or netmasks; you must list individual hostnames or IP addresses.

We configure FreeBSD as a printer client in Chapter 20.

/etc/inetd.conf

inetd(8) handles incoming network connections for smaller daemons that don't run frequently. See the section on inetd in Chapter 20.

/etc/libmap.conf

FreeBSD's linker lets you substitute one shared library for another. We discuss this in Chapter 17.

/etc/localtime

This file contains local time zone data, as configured by tzsetup(8). It's a

binary file, and you can't edit it with normal tools. tzsetup(8) actually copies this file from a subdirectory of */usr/share/zoneinfo*. If your time zone changes, you'll need to upgrade FreeBSD to get the new time zone files and then rerun tzsetup(8) to configure time correctly.

We discuss time in Chapter 20.

/etc/locate.rc

locate(1) finds all files of a given name. For example, to find *locate.rc*, enter the following:

```
# locate locate.rc
/etc/locate.rc
/usr/share/examples/etc/locate.rc
/usr/src/usr.bin/locate/locate/locate.rc
/var/db/etcupdate/current/etc/locate.rc
```

You'll see that a file called *locate.rc* can be found in four places: in the main */etc* directory, the system examples directory, the system source code, and a copy retained by etcupdate(8).

As part of periodic(8)'s weekly run (see Chapter 21), your FreeBSD system scans its disks, builds a list of everything it finds, and stores that list in a database. The list-building program, locate.updatedb(8), takes its settings from */etc/locate.rc*. The following variables in this file all change how your locate.updatedb(8) builds your locate database:

- `TMPDIR` contains the temporary directory used by locate.updatedb(8), and defaults to */tmp*. If you're short on space in */tmp*, change this path to a place where you have more room.
- While you can change the location of the database itself with the `FCODES` variable, this affects other parts of FreeBSD that expect to find that database in its default location. Be prepared for odd results, especially if you leave an old locate database in the default location of */var/db/locate.database*.
- The `SEARCHPATHS` variable gives the directory where you want to start building your database. This defaults to /, the whole disk. To index only a portion of your disk, set that value here.
- `PRUNEPATHS` lists directories you don't want to index. This defaults to

excluding temporary directories that traditionally contain only short-lived files.

- The FILESYSTEMS variable lists the types of filesystems you want to index. By default, locate.updatedb(8) indexes only UFS (FreeBSD) and ext2fs (Linux) filesystems. Listing NFS (see Chapter 13) filesystems is a bad idea: all of your servers simultaneously indexing the fileserver will bottleneck either the network or the fileserver.

/etc/login.*

You can control who may log into your system—and what resources those users may access—by using */etc/login.access* and */etc/login.conf*. See Chapter 9 for instructions.

/etc/mail

Most of the contents of */etc/mail* are dedicated to Sendmail. The two exceptions are the aliases(5) file and mailer.conf(5). We discuss both in Chapter 20.

/etc/mail.rc

While FreeBSD uses most *.rc* files for system startup, the */etc/mail.rc* file is used to configure mail(1).

/etc/mail/mailer.conf

FreeBSD allows you to choose any mail server program you like via */etc/mail/mailer.conf*, as covered in Chapter 20.

/etc/make.conf

To make, or compile, a program is to build it from source code into machine language. We'll discuss building software in detail in Chapter 17. The

/etc/make.conf contains settings that control how the building process works, letting you set options that directly affect software builds. Remember, anything you add to *make.conf* affects all software built on the system, including system upgrades. This may cause upgrade failures.[2] Many of the options from *make.conf* are useful only to developers.

If you're interested in setting options that affect only system upgrades, use */etc/src.conf* instead.

Here are some common features set in *make.conf*. Any values set here require the same syntax used by make(1). If you insist on trying to optimize software builds, follow the examples in make.conf(5) or in the software's documentation. Best of all, though, don't muck with make at all.

CFLAGS

This option specifies optimization settings for building nonkernel programs. Many other Unix-like operating systems suggest compiling software with particular *compiler flags*, or CFLAGS. This practice is actively discouraged on FreeBSD. System components that require compiler flags already have that specified in the software configuration, and add-on software has that configuration set for it separately. While people might recommend other settings for CFLAGS, custom options aren't supported by the FreeBSD Project.

In general, FreeBSD code is expected to compile most correctly out of the box. The only thing that adding compiler options can do is impair your performance. If you build FreeBSD or ports with nonstandard flags and have problems, remove those flags and build it again.

COPTFLAGS

The COPTFLAGS optimizations are used only for building the kernel. Again, settings other than the defaults can build a nonworking kernel.

CXXFLAGS

CXXFLAGS tells the compiler what optimizations to use when building C++ code. Be sure to use the += syntax when using CXXFLAGS so that you add your instructions to those specified in the software. Everything that I said earlier about CFLAGS applies equally well to CXXFLAGS.

/etc/master.passwd

This file contains the confidential core information for all user accounts, as discussed in Chapter 9. Protect it.

/etc/motd

The *message of the day (motd)* file is displayed to users when they log in. You can place system notices in this file or other information you want shell users to see. The `welcome` option in */etc/login.conf* (see Chapter 9) can point users to different motd files, so you can have separate messages for each login class.

/etc/mtree

mtree(1) builds directory hierarchies with permissions set according to a predefined standard. The */etc/mtree* directory stores that standard for the FreeBSD base system. The FreeBSD upgrade process uses mtree records to install the system correctly. If you damage file or directory permissions in your base system, you can use mtree(1) to restore them to the defaults. While you don't generally need to edit these files, they can be useful if you muck too much with your system. Diskless systems use these files to build memory-based */var* filesystems. We'll use this information in Chapter 19 to inspect system security.

/etc/netconfig

If you're accustomed to SVR4-derived operating systems, you might expect to configure various parts of networking in */etc/netconfig*. FreeBSD uses this file only for RPC code, though. I mention it only to keep old Solaris hands from thinking changes here will help them.

/etc/netstart

This shell script is designed specifically for bringing up the network while in single-user mode. Having a network in single-user mode is terribly useful for

any number of reasons, from mounting NFS shares to connecting to remote machines in order to verify configurations. Just run */etc/netstart*. This script has no effect when in full multiuser mode.

/etc/network.subr

This shell script isn't intended for human use; rather, other network configuration scripts use the subroutines defined herein to support common functions.

/etc/newsyslog.conf

This file configures the rotation and deletion of log files. See Chapter 21 for more information.

/etc/nscd.conf

The nscd(8) service caches the results of name service lookups to optimize system performance. It's useful if you're running LDAP, but for hostname lookups, you're better off running a local caching resolver.

/etc/nsmb.conf

FreeBSD's Windows file-share mounting system uses */etc/nsmb.conf* to define access to Windows systems, as described in Chapter 13.

/etc/nsswitch.conf

Name Service Switching is covered in Chapters 8 and 20.

/etc/ntp/, /etc/ntp.conf

Keeping correct time on your host simplifies . . . well, everything. The time daemon ntpd(8) uses these files, as Chapter 20 illustrates.

/etc/opie*

One-time Passwords In Everything (OPIE) is a one-time password system derived from S/Key. While still used in a few places, it's no longer very popular. You can read opie(4) if you're interested. For the most part, OPIE has been largely replaced by systems like Kerberos, Google Authenticator, and other PAM plug-ins.

/etc/pam.d/*

Pluggable Authentication Modules (PAM) allow the sysadmin to use different authentication, authorization, and access control systems. If you're using Kerberos, LDAP, or some other centralized authentication system, you'll need to configure PAM. PAM, unfortunately, fills an entire book on its own. If you're trapped into going anywhere near PAM, permit me to recommend my book *PAM Mastery* (Tilted Windmill Press, 2016).

/etc/passwd

This file contains user-visible account information. We talk about the password files in Chapter 9.

/etc/pccard_ether

This script starts and stops removable network cards, such as Cardbus cards and USB Ethernet. Its name is just a leftover of history, when the only cards available were PC Cards. For the most part, devd(8) runs this script as needed, as discussed in Chapter 13.

/etc/periodic.conf and /etc/defaults/periodic.conf

The system's regular maintenance jobs that create those annoying mails to root are run by periodic(8), which just runs shell scripts stored in */etc/periodic* and */usr/local/etc/periodic*. Every one of these scripts can be enabled or disabled in */etc/periodic.conf*.

periodic(8) runs programs either daily, weekly, or monthly. Each set of programs has its own settings—for example, daily programs are configured separately from monthly programs. These settings are controlled by entries in */etc/periodic.conf*. While we show examples from only the daily scripts, you'll find very similar settings for the weekly and monthly scripts.

daily_output="root"

If you want the status email to go to a user other than root, list that user's name here. Unless you have a user whose job it is to specifically read periodic email, it's best to leave this at the default and forward root's email to an account you read. You could also give a full path to a file if you prefer and even have newsyslog(8) rotate the periodic log (see Chapter 19).

daily_show_success="YES"

With this set to YES, the daily message includes information on all successful checks.

daily_show_info="YES"

When set to YES, the daily message includes general information from the commands it runs.

daily_show_badconfig="NO"

When set to YES, the daily message includes information on periodic commands it tried to run but couldn't. These messages are generally harmless and involve subsystems that your system just doesn't support or include.

daily_local="/etc/daily.local"

You can define your own scripts to be run as part of the daily, weekly, and monthly periodic(8) jobs. These default to */etc/daily.local*, */etc/weekly.local*, and */etc/monthly.local*, but you can place them anywhere you like.

Each script in the daily, weekly, and monthly subdirectories of */etc/periodic*

has a brief description at the top of the file, and most have configuration options in */etc/defaults/periodic.conf*. Skim through these quickly, looking for things that are of interest to you. The defaults enabled are sensible for most circumstances, but there's extra functionality you can enable with a simple setting in */etc/periodic.conf*. For example, if you use GEOM-based disk features, you'll find the daily GEOM status messages useful. Since anything I could list here would be obsolete before I could deliver this manuscript, let alone before the book reaches you, I won't go into detail about the various scripts.

/etc/pf.conf, /etc/pf.os

We cover the basics of the PF packet filter in Chapter 19.

One less-known feature of PF is its ability to identify operating systems by the packets they send. The */etc/pf.os* file contains TCP fingerprints for different operating systems, allowing you to write firewall rules such as "Show FreeBSD users my real home page, but show Windows users a page suggesting that they get a real operating system." See pf.os(5) for more information. I encourage you to peruse this file, if only to drive home how all these network stacks behave so differently.

/etc/phones

Modem users can store phone numbers for remote modems in */etc/phones*, aliasing them so that they can just type `home` instead of the full phone number. Only tip(1) and cu(1) use this file, however, so it's not as useful as you might think.

/etc/portsnap.conf

Portsnap provides updates for the ports tree, as discussed in Chapter 18.

/etc/ppp/

FreeBSD supports outbound modems with ppp(8). Read the man page for

more information.

/etc/printcap

This file contains printer configuration information. Printing on Unix-like systems can be very complicated, especially with the vast variety of printers you can use. Making your FreeBSD machine send print jobs to a print server isn't hard at all, however. We cover the topic in Chapter 20.

/etc/profile

The */etc/profile* files contain the default account configuration information for the */bin/sh* shell, much like */etc/csh.** for csh and tcsh users. Whenever a */bin/sh* user logs in, he inherits what's in this file. Users can override */etc/profile* with their own *.profile*. Bash and other sh derivatives also use this file.

While tcsh is the standard FreeBSD shell, sh and derivatives (particularly bash) are quite popular. Keep settings in */etc/profile* and */etc/csh.login* synchronized to ease troubleshooting in the future—or, better still, set necessary environment variables in a login class (see Chapter 9) so that they affect any shell the user needs.

/etc/protocols

In Chapter 7, we discussed network protocols. The */etc/protocols* file lists the various network protocols you might encounter. Remember, a TCP or UDP port number isn't the same as a protocol number.

/etc/pwd.db

This is the database version of the */etc/passwd file*. It contains public information about user accounts, as discussed in Chapter 9.

/etc/rc*

Whenever your system boots to the point where it can execute userland commands, it runs the shell script */etc/rc*. This script mounts all filesystems, brings up the network interfaces, configures devfs(5), finds and catalogs shared libraries, and performs all the other tasks required to set up a system. We discussed the FreeBSD startup system in Chapter 4.

Different systems have radically different startup tasks. A terminal server with three 48-port serial cards works completely differently from a web server. Instead of a single monolithic */etc/rc* script that handles every task, FreeBSD segregates each startup process into a separate shell script that addresses a specific need.

Additionally, you'll find a few scripts directly under */etc*, such as */etc/rc.firewall* and */etc/rc.initdiskless*. These scripts were split out on their own years before the current startup system came along, and remain in their historical locations because there isn't anything to be gained by moving them.

/et/regdomain.xml

Wireless cards are subject to different regulatory rules depending on where in the world they're used. The cards read *regdomain.xml* to learn which frequencies they may use and how strongly they're allowed to transmit. Edit this at your own risk.

/etc/remote

This file contains machine-readable configurations for connecting to remote systems over serial lines. Today, this is of interest only if you use your system as a serial client—for example, if you want to connect to a serial console. We discuss serial consoles in Chapter 4.

/etc/resolv.conf

This file lets you set nameservers, domain search order, and more DNS client settings. See Chapter 8 for the details.

/etc/rpc

Remote Procedure Calls (RPC) is a method for executing commands on a remote computer. Much like TCP/IP, RPC has service and port numbers, and */etc/rpc* contains a list of these services and their port numbers. The most common RPC consumer is NFS, discussed in Chapter 13.

/etc/security/

This directory contains configuration information for the audit(8) security utility.

/etc/services

This file contains a list of network services and their associated TCP/IP ports. We discussed */etc/services* in Chapter 7.

/etc/shells

This file contains the list of all legitimate user shells, as discussed in Chapter 9.

/etc/skel/

In */etc/skel/*, you'll find shell dotfiles that get copied to the new user accounts.

/etc/snmpd.config

FreeBSD includes a basic SNMP implementation, which we discuss in Chapter 20.

/etc/spwd.db

This file contains the confidential database of the user password file

/etc/master.passwd. See Chapter 9 for all the inglorious detail.

/etc/src.conf

This file contains machine instructions for building FreeBSD from source. It's a parallel of *make.conf* for the source tree alone. Values set in */etc/make.conf* affect building FreeBSD from source as well, though; the difference is that */etc/src.conf* affects only building FreeBSD but not ports and packages. See Chapter 18 for all your upgrading needs.

/etc/ssh/

Configure the Secure Shell software suite (SSH) in */etc/ssh*. This includes the client ssh(1) and the server sshd(8). Chapter 20 touches on sshd(8).

/etc/ssl/

FreeBSD includes the OpenSSL cryptographic software. Chapter 19 discusses a few basic uses and configuration. The */etc/ssl* directory contains most OpenSSL information.

/etc/sysctl.conf

This file contains information on which kernel sysctls are set during the boot process. See Chapter 6.

/etc/syslog.conf, /etc/syslog.conf.d/

This file controls which data goes into your system logs and where those logs are stored. See Chapter 21.

/etc/termcap, /etc/termcap.small

This file contains the settings and capabilities of different terminal types. In

the age when terminals came in dozens of different types and vendors released new terminals on an almost daily basis, understanding this file was vital. Now that the world has largely converged on vt100 as a standard, however, the default configuration is suitable for almost everyone.

The termcap file is a symlink to */usr/share/misc/termcap*. This file might not be available in single-user mode. FreeBSD offers the */etc/termcap.small* file to provide terminal information in single-user mode.

/etc/ttys

This file contains all of the system terminal devices (the windows containing a command prompt). The name is a relic of the time when terminals were physical teletypes, but today most users use the virtual terminals generated by telnet or SSH.

We'll use this file to set up serial logins in Chapter 23.

/etc/unbound/

FreeBSD ships with the Unbound DNS client. The configuration information goes in */etc/unbound/*. Chapter 20 covers setting up Unbound as a local DNS resolver.

/etc/wall_cmos_clock

This isn't a vital file, but as I went to the trouble of digging up what it does, you get to learn about it. If this file exists, FreeBSD's time-keeping routines have determined that the hardware's CMOS clock keeps a time other than Coordinated Universal Time (UTC). If the file is missing, the CMOS clock is set to some other time. It's documented in adjkerntz(8).

/etc/zfs/

FreeBSD's ZFS uses this directory to store NFS integration information. We discuss NFS in Chapter 13.

If you can crawl through all of */etc*, you'll be better prepared than most

sysadmins. Now let's add some software to your server.

15
MAKING YOUR SYSTEM USEFUL

A basic FreeBSD install includes *exactly* enough to make the system run, plus a few extra bits that Unix systems traditionally include. You can decide whether to install additional programs or source code. While FreeBSD has grown over the years, a complete base install fills about a gigabyte—much less disk space than either a Windows or a commercial Linux install.

The advantage to this sparseness is that it includes only necessary system components. Debugging becomes much simpler when you know that no shared library you've never even heard of, and would never use, can be responsible for your problems. The downside is that you must decide what functions you do need and select software to provide those functions. FreeBSD simplifies add-on software installation through ports and packages.

Ports and Packages

FreeBSD supports two different ways to install add-on software. Everything starts with the Ports Collection, but most users will prefer preconfigured packages.

FreeBSD has a system for building add-on software called the *Ports Collection*, or just *ports*. Ports let you start with raw source code from the program vendor and build the software in exactly the way you need, enabling and disabling features as you need. Ports are fast and easy for the experienced user but require a certain amount of Unix expertise and can intimidate the new user.

Packages are the result of building ports, using the options the port maintainer thinks will be most useful to the widest variety of people, and bundling them up in a lump to make them easily installable. The FreeBSD Project has a whole farm of systems that do nothing but build all the ports, bundle them up, and make them available for users to download and install. Packages let you quickly install, uninstall, and upgrade add-on software.

INTERNET ADVICE STICKS AROUND FOREVER

Forums and mailing list archives contain many recommendations to skip packages and proceed directly to ports. This is no longer true; packages are preferable for both technical reasons and your own peace of mind. The older package system, pkg_tools, had serious limitations.

Ignore any recommendations that mention tools like pkg_add(8), pkg_delete(8), pkg_create(8), and so on. It's extremely obsolete.

FreeBSD's highly flexible packaging system is called *package*, pkg(8), or just plain pkg. Package information gets stored in SQLite databases, which lets you perform arbitrary queries on package data. During development, pkg was called *pkgNG*. That name's been gone for years now but lingers on in some old docs and third-party software. Don't let the name confuse you.[1]

We'll start by discussing managing systems with pkg(8) and then proceed to customizing software with ports.

Packages

Packages are precompiled software from the Ports Collection, bundled up for a particular version of FreeBSD. The FreeBSD Project offers several sets of packages in a public repository, updated every few days. Packages are the simplest way to install add-on software. Any software without legal restrictions on its distribution is probably available as a package.

Legal restrictions? Software can have any license terms, including some really odd ones. The license of some software prohibits distribution in any form other than source code. FreeBSD can't legally package that. Other software can legally be distributed only in precompiled form. FreeBSD usually packages such software, distributing it as the precompiled binary plus FreeBSD-specific packaging information.

Packages are built on the oldest supported release of each major FreeBSD version. Packages for all versions of FreeBSD 12 are built on the oldest supported release of FreeBSD 12, FreeBSD 13 packages are built on the oldest supported version of FreeBSD 13, and so on. This helps reduce, identify, and contain ABI incompatibilities.

If you need to build your own package repository, investigate the Ports Collection (see Chapter 16) and the add-on package *poudriere*.

Package Files

Ultimately, packages contain files. Those files might be binary programs, documentation, configuration files, or anything else the software might need. These files are considered part of the operating system. Don't edit them manually.

The one odd case is when a package contains a sample configuration file. If a program needs a configuration file, the package should include a sample. You'll need to edit the configuration to fit your needs—that's what configuration files are *for*.

FreeBSD reconciles this by installing the package's configuration files with the suffix *.sample*. Our web server configuration file appears as something like *httpd.conf.sample*.

If there's no production configuration file, the package installation also copies the sample file into place. That file is yours to edit.

If you upgrade a package, pkg(8) compares the current production file to the old sample file. If the sample is identical to the production version, the upgrade replaces the production file. If the files differ in any way, pkg updates only the sample file. It's your job to merge any desirable changes into your production configuration. Note that the package upgrade always replaces the sample configuration, so if an old sample is important, you need to make a point to hang on to it.

Introducing pkg(8)

Unlike the older packaging system, pkg(8) is a single program with a whole flock of subcommands. You'll use the same program to install, uninstall, and investigate packages. All changes to installed packages must be run as root. Here's how you'd install a vital program desired by all right-thinking sysadmins:

```
# pkg install emacs
```

Those of you clinging to irrational biases against superior text processors probably want to remove it.

```
# pkg delete emacs
```

All package operations use the pkg(8) command.

While the pkg(8) man page documents the base pkg functions, each subcommand has its own man page, named *pkg-* and the subcommand. Examples include pkg-install(8) and pkg-delete(8). You can also use the `pkg help` command and the name of the subcommand to get assistance—for example, `pkg help install`.

FreeBSD doesn't ship with pkg(8) installed. You need to install it . . . as a package. No, wait, don't scream—it's *much* better than it sounds.

Installing pkg(8)

FreeBSD ships with a very simple package manager in */usr/sbin/pkg*, pkg(7). It has barely enough brains to find FreeBSD's current package manager. It installs that new package manager and surrenders all responsibility for package management to it. This gives FreeBSD the flexibility to update the package manager with the packages.

The first time you try to install a package, pkg(8) prompts you to install the package manager. I found I needed the dmidecode package on a new server, so I can get an RMA on a bad power supply from the manufacturer. (Don't worry about how I *found* the dmidecode package—just go with me for the moment.)

```
# pkg install dmidecode
```

FreeBSD runs pkg(8) and finds that no package management is installed yet.

```
The package management tool is not yet installed on your system.
Do you want to fetch and install it now? [y/N]: y
```

The default answer appears in capital letters. If I hit `n` and ENTER, pkg terminates. If I hit `y` and ENTER, FreeBSD bootstraps the system.

```
❶ Bootstrapping pkg from pkg+http://pkg.FreeBSD.org/FreeBSD:12:amd64/quarterly,
please wait...
❷ Verifying signature with trusted certificate pkg.freebsd.org.2013102301... done
❸ Installing pkg-1.10.0_2...
   Extracting pkg-1.10.0_2: 100%
❹ Updating FreeBSD repository catalogue...
   meta.txz                                : 100%  944 B     0.9kB/s    00:01
   packagesite.txz                         : 100%    6 MiB   2.0MB/s    00:03
   Processing entries: 100%
   FreeBSD repository update completed. 26059 packages processed.
   All repositories are up to date.
   Updating database digests format: 100%
```

The installation starts by downloading the current pkg tools from a FreeBSD mirror ❶. It then checks the digital signature on the downloaded file ❷. The tools are extracted and installed ❸. Pkg then downloads and installs the catalog of available packages ❹.

The packaging system is now installed. To smooth things out, the stub pkg(8) that ships with FreeBSD tells the newly installed packaging system to install the program you really wanted. In this case, our new pkg(8) installs dmidecode for you.

You can install the packaging system on its own, without adding other packages, by running **`pkg bootstrap`**—but seriously, nobody does that at the command line. Running **`pkg bootstrap`** does nothing when the packaging system is already installed, so it's useful for setup scripts.

Common pkg Options

While each pkg subcommand has unique features, a few command options work across almost all of them.

In its default configuration, pkg prompts you for confirmation before

doing anything. Tell pkg to take action without prompting you with the `-y` flag.

On the other hand, maybe you want pkg to show you what it would do if you ran a command, but not actually do anything. Perform a dry run by using the `-n` flag. For example, a package install using `-n` would show the names of every package to be installed, including dependencies. There's no risk that the system would install the packages, however. Dry runs can help you prepare for changes during a maintenance window.

Many pkg operations produce a bunch of output. Reduce the amount of output with `-q`.

The `-a` flag usually applies a command to all installed packages.

Finally, pkg usually refuses to do pointless things or things that damage the system. The `-f` flag forces pkg to do what you said. Forcing package activities is usually, but not always, a bad idea. For example, you might need `-f` to forcibly reinstall a damaged package.

Configuring pkg(8)

The pkg(8) program is designed to be highly flexible. While each subcommand has a whole bunch of options, you can establish customized but consistent behavior for most programs with the configuration file, */usr/local/etc/pkg.conf*.

The *pkg.conf* file contains commented-out defaults for pkg(8). It's a great place to look to see how the system behaves when you haven't mucked with it at all. The configuration is written in UCL (see Chapter 2). Variables can be set to an integer; a string, such as a file path; or a Boolean value, like `YES` or `NO`. `YES`, `ON`, and `TRUE` are synonyms, as are `NO`, `OFF`, and `FALSE`. All are case-insensitive.

```
#PKG_DBDIR = "/var/db/pkg";
#PKG_CACHEDIR = "/var/cache/pkg";
#PORTSDIR = "/usr/ports";
#INDEXDIR = "";
--snip--
```

FreeBSD runs perfectly well with an empty *pkg.conf*. The default configuration contains a whole bunch of commented-out entries and quite a few aliases. You might consider these example settings as you proceed.

Most pkg operations offer a yes/no dialog, showing the default as a capital letter. Being conservative, pkg normally defaults to `NO`. Change that default to `YES` with the `DEFAULT_ALWAYS_YES` option.

You can make a pkg command assume you'll answer yes to everything by adding the `-y` flag. If you get tired of typing `-y`, make pkg assume you always answer yes by setting the `ASSUME_ALWAYS_YES` flag to `YES`.

As I'm lazy but not reckless, I prefer these *pkg.conf* settings:

```
DEFAULT_ALWAYS_YES = true;
ASSUME_ALWAYS_YES = false;
```

If installing a package runs amok, you might want debugging output. Setting `DEBUG_LEVEL` turns on debugging output. This variable accepts an integer from 0 (no debugging) to 4 (complete debugging).

Many packages include scripts as part of their installation procedure. Turn on debugging for each script by setting `DEBUG_SCRIPTS` to `YES`.

Any *pkg.conf* settings are also usable as environment variables. Environment variables override anything in the configuration file. You could install a package with debugging like this:

```
# env DEBUG_LEVEL=4 pkg upgrade
```

All of the options are documented in pkg.conf(5). Not all of the options have a commented-out entry, though. If a sample of an option doesn't exist but you want it, add it. We'll examine many of them in the following sections.

Finding Packages

Now that you have a package manager installed, you can install packages. Sysadmins familiar with a variety of Unix-like operating systems know that different operating systems assign different names to packaged versions of the same software. A package for the Apache web server on FreeBSD will have a completely different name than the packaged Apache on illumos or even different Linux distributions. Before you can install anything, you'll need to figure out what it's called.

Suppose the client wants to run WordPress on Apache. Your job isn't to question the client's choice in web servers; your job is to build and support

the web server. First, find Apache with the pkg search command. You'll need to provide a text string for pkg to perform a case-insensitive search.

```
# pkg search apache
apache-ant-1.9.7    Java- and XML-based build tool
apache-forrest-0.9  Tool for rapid development of small sites
apache-mode.el-2.0  Emacs major mode for editing Apache configuration files
--snip--
```

I deliberately picked an annoying example; FreeBSD has 50-odd packages related to the Apache web server. Fortunately, each search result lists a one-line package description. It's pretty easy to flip through the results until you find the actual web servers.

```
--snip--
apache22-2.2.31_1              Version 2.2.x of Apache web server with prefork MPM
apache22-event-mpm-2.2.31_1    Version 2.2.x of Apache web server with event MPM
apache22-itk-mpm-2.2.31_1      Version 2.2.x of Apache web server with itk MPM
apache22-peruser-mpm-2.2.31_1  Version 2.2.x of Apache web server with peruser MPM
apache22-worker-mpm-2.2.31_1   Version 2.2.x of Apache web server with worker MPM
apache24-2.4.25_1              Version 2.4.x of Apache web server
--snip--
```

Six different versions of Apache. First, look at the package names. When a piece of software comes in multiple versions, the major version number gets integrated into the package name. Apache 2.2 is a very different beast than Apache 2.4, so the packages are named *apache22* and *apache24*. The actual version number follows. Our first Apache 2.2 package is actually for Apache 2.2.31. The trailing _1 is the package version number, which means that this an updated package. The included software hasn't changed, but the package has been altered somehow. Package version numbers get bumped for two reasons. When the source port changes in a way that has a material impact on the package, the version number is increased. When an ABI change in a required shared library demands recompiling the package, that also merits a version bump.

Apache 2.2 comes in five different packages. People familiar with Apache probably remember that this version of Apache could use different Multi-Processing Modules (MPMs), but the MPM had to be selected at compile time. I have blissfully forgotten everything I ever knew about MPMs, so rather than fuss with them, I'll choose to install the Apache 2.4 package, apache24.

Package Searching Options

Some searches can generate hundreds of results. Try searching for Perl, and you'll get about 150 packages. Perl modules all begin with the string `p5-`; FreeBSD has packages for over 5,200 Perl modules! Use command line options to trim or adjust the search results. While pkg-search(8) lists many options, here are some of the most common.

- Make a search case-sensitive with `-c`.
- If you know exactly which package you want, and you only want to see whether it's available for your system, use the `-e` flag to search for an exact match. Your search term must include the package version number, though.
- If you need to highly customize your searches and your search results, investigate the `-L`, `-S`, and `-Q` flags in pkg-search(8).

Examining Found Packages

Perhaps you're not sure whether a package is what you really want. You might look up details on the package from a third-party site, like FreshPorts (*https://www.freshports.org/*), but that would require leaving your terminal, and I can't countenance that. Use the `-R` flag to examine the repository catalog's metadata for the package. This metadata is a subset of the full *package manifest* built into each package.

```
# pkg search -R apache24
name: "apache24"
origin: "www/apache24"
version: "2.4.25_1"
comment: "Version 2.4.x of Apache web server"
maintainer: "apache@FreeBSD.org"
www: "http://httpd.apache.org/"
--snip--
```

The package manifest includes fields for the package name, the port the package is built from, the software version, the package repository, dependencies, and more. It's rarely enough used and subject to change, so we won't discuss it in detail, but scrolling through this information provides more details about the software inside the package.

One important detail here is the *www* field, which gives the website the original software comes from. This is the Apache web server, not a fork or

some other project using that name.

The default format for this raw manifest is YAML, or "YAML Ain't Markup Language." It's yet another syntax for formatting configuration files, but it's fairly human-readable. Use the `--raw-format` flag to choose an alternate format. Other supported formats include json and json-compact.

```
# pkg search -R --raw-format json-compact apache24
```

If you want to automatically parse package information, this is how you grab the raw data.

Installing Software

Use pkg's `install` subcommand and the name of a package to install a package. You don't need to give the complete package name.

```
# pkg install apache24
```

The first thing that happens is that pkg checks to see whether its local copy of the package database is the same as that on the package server. You'll either get a message like "Updating FreeBSD repository catalogue" or be told that the "FreeBSD repository is up to date."

The system then checks for any packages that your chosen package requires. Read the dependency list. Is there anything here you don't want installed on this host? Does the list give you a reason *not* to install the package?

```
The following 8 package(s) will be affected (of 0 checked):
New packages to be INSTALLED:
        apache24: 2.4.25_1
        libxml2: 2.9.4
--snip--
Number of packages to be installed: 8
```

As a final warning, pkg tells you how much disk space and bandwidth the installation requires. You then get prompted to change your mind.

```
The process will require 139 MiB more space.
33 MiB to be downloaded.

Proceed with this action? [y/N]:
```

Enter **y**, and pkg fetches the package from the repository and installs it to your system.

The `install` subcommand assumes that you're either giving the complete name of a package or the name of a package without the package version number. You can request the apache24 package and pkg will figure out that the current package is apache24-2.4.25_1. You can also use the name of the port the package was built from, as in `pkg install www/apache24`.

In the last section, our package search turned up five different Apache 2.2 packages, each a slightly different variant. If you ask `pkg install` to grab the apache22 package, it installs the version named apache22 plus a package version number. If you want a variant, such as apache22-event-mpm, specify the full package name in the `install` command.

Some packages include installation messages. These messages might be helpful instructions, warnings, caveats, or anything else relevant. If the package creator felt a chunk of information sufficiently important enough to spend her precious time composing a message about it, then the least you can do is read it. You might use script(1) to record this information or run `pkg info --pkg-message` and the package name to show it again.

Fetching Packages

FreeBSD installs packages by downloading them over the internet. You might want to download packages in one location to install them elsewhere or at another time. Use the **`pkg fetch`** command to download but not install packages. Fetching packages makes the most sense when combined with `-d`, which makes `pkg fetch` grab all the dependencies as well as the named package.

```
# pkg fetch -d apache24
```

You'll see the normal repository update messages, followed by a notice of what pkg will download.

```
New packages to be FETCHED:
        apache24-2.4.25_1 (5 MiB: 14.25% of the 33 MiB to download)
        libxml2-2.9.4 (821 KiB: 2.43% of the 33 MiB to download)
--snip--
```

Verify that what pkg plans to download matches what you expect, and

then hit `y` to proceed. The packages are downloaded to the package file cache.

To install a downloaded package, run `pkg install` normally. The installation process uses the cached files rather than the downloaded ones.

Those of you who read man pages might notice the `-a` flag, which downloads the entire package repository. Don't use that. The `-a` option is intended for public repository mirrors. Average sysadmins who mirror the entire repository waste bandwidth and slow down the system for everyone. Generous people donate FreeBSD's package server bandwidth. Don't waste it. You might need a whole bunch of packages. With dependencies, you might need hundreds or even thousands of packages. You don't need tens of thousands of packages. Download only what you need.

Download Timing

Any tool that accesses the internet needs to set a maximum length of time to try to download files. You can customize pkg's download behavior with two *pkg.conf* settings.

If a download fails, pkg tries again. The `FETCH_RETRY` option controls how many times pkg retries a download. The default is three, which means that it tries to download one time and retries up to three more times.

Downloads happen fairly quickly on most modern internet connections. If your uplink isn't quite so modern, you might need to increase the amount of time pkg will spend on a single download attempt. The `FETCH_TIMEOUT` setting controls how long pkg waits for any one file to download. The default, 30, limits downloads to 30 seconds. If you're downloading LibreOffice over a 33.6 modem, you'll want to increase this setting and consider having files shipped to you on a removable drive via the Pony Express.

The Package Cache

The ability to download packages and install them later implies that pkg(8) sticks those packages somewhere on the disk for later consumption. The package cache, */var/cache/pkg*, contains the original package files downloaded from the internet. While you can administer FreeBSD hosts for years without futzing with the cache, here are a few things you should know.

Cleaning the Cache

What with upgrades, new packages, removed packages, and the gleeful randomness of system administration, the cache directory can fill up. My web server has only a few packages but somehow has accumulated 1.7GB of old package files. The `pkg clean` command removes any cached packages that have been replaced by newer versions, as well as any package files that are no longer in the repository. You'll get a list of all the files that will get removed, plus a chance to proceed or quit.

```
# pkg clean
The following package files will be deleted:
        /var/cache/pkg/php56-mbstring-5.6.26.txz
        /var/cache/pkg/mod_php56-5.6.21-c80f5ce183.txz
--snip--
```

If you've never cleaned the package cache on a long-running system, the list will be pretty long. At the prompt, hit **y** to proceed.

If you want to remove all cached packages, use the `-a` flag.

Remember that `pkg clean` removes package files that are no longer available in the package repository. If you depend on a package that's been removed from the repository, back up that file outside the cache before a thoughtless cleaning removes it forever. You could also try pkg-create(8) to rebuild a package from its installed components.

If you want to clean the package cache automatically after each package install or upgrade, set the *pkg.conf* option `AUTOCLEAN` to **`true`**. I find autocleaning too aggressive, as sometimes the new bugs in an upgraded package compel me to revert to the older version. We cover upgrading packages at the end of this chapter.

Moving the Cache

You might want the package cache located elsewhere on the filesystem. Use the *pkg.conf* option `PKG_CACHEDIR` to set a new package cache directory.

Why move the cache directory? Many server farms share a package cache across multiple machines. You can safely share a package cache between hosts running the same FreeBSD major release and hardware architecture. Verify that your NFS configuration uses locking, and set the *pkg.conf* option `NFS_WITH_PROPER_LOCKING`.

Package Information and Automatic Installs

After a while, you'll forget which packages you've installed on a system. Get the complete list of installed software with `pkg info`.

```
# pkg info
gettext-runtime-0.19.8.1_1  GNU gettext runtime libraries and programs
indexinfo-0.2.6             Utility to regenerate the GNU info page index
--snip--
```

If you want more information about an installed package, use `pkg info` and the package name. This shows the package manifest and installation details in a human-friendly report.

```
# pkg info apache24
apache24-2.4.25_1
Name           : apache24
Version        : 2.4.25_1
Installed on   : Tue Mar 14 16:56:14 2017 EDT
Origin         : www/apache24
Architecture   : freebsd:12:x86:64
--snip--
```

When was the package installed? Was the package installed on this machine built from the Ports Tree with certain options enabled? What's the license? What shared libraries does every program in the package require? Answer all these and more with `pkg info` and the package name.

The `pkg info` subcommand has many other features. We'll see some of them, such as locking status, later this chapter. The pkg-info(8) man page has the complete details.

Automatic Packages

Look back at the sample `pkg info` output. I deliberately installed a few different programs on this system, but I'm pretty sure I never knowingly installed anything about GNU info pages or gettext.

I did install those programs. I merely didn't pay much attention to what they were because I was more concerned about installing the package that required them. They're dependencies.

FreeBSD records whether you requested a package be installed or it was brought along as a dependency. Packages installed as dependencies are called *automatic* packages. Packages you requested are just packages, although

they're sometimes called *nonautomatic* packages.

You might want to know which packages you requested to be installed and which were dragged along as dependencies. That's when things get tricky.

Querying the Package Database

The pkg tools can't cover every possible contingency a sysadmin might face. The simplest way to get some information is to interrogate the installed package database. While you could use raw SQLite, that would mean you'd need to become intimate with the database's innards. Most sysadmins don't have that kind of time, especially when that database might change any time. FreeBSD insulates from that with the `pkg query` subcommand. A complete survey of package queries would fill a chapter, but here's a quick overview.

REMOTE QUERIES

Use pkg-query(8) to search the database of installed packages. If your database of packages available in the repository is up to date, though, you can search it using pkg-rquery(8). The database of available packages doesn't contain all the metadata of an installed package, however, so not all patterns are available.

Anything you might want to get out of the package database has a convenient representation in `pkg query`. The catch is, everything than anyone might possibly want to extract from the package database is in `pkg query`, as a quick perusal of pkg-query(8) shows. The query and command structure is deliberately designed for use in scripts, but we'll use it interactively now and then.

Run queries by using *patterns*. A pattern is a variable that has an assigned meaning, represented by a percent sign and a letter. For example, `%n` contains the package name, `%o` contains the port the package was built from, and `%t` contains the timestamp indicating when the package was installed.

Running `pkg query` and giving a pattern produces that value for every

installed package. As `%n` represents the package name, here's how you'd get a list of everything on the system:

```
# pkg query %n
apache24
apr
--snip--
```

We don't get the extra information `pkg info` shows—but maybe that's what you want.

You can request multiple items in a single query. The `%v` pattern represents the package version, while `%c` represents the comment. Here, I separate the package name and version with a dash but put a tab between the version and the comment. Using the shell tab character `\t` means I must quote the `pkg query` argument.

```
# pkg query "%n-%v\t%c"
apache24-2.4.25_1       Version 2.4.x of Apache web server
apr-1.5.2.1.5.4_2       Apache Portability Library
--snip--
```

You know, this looks an awful lot like the output of `pkg info`. When a pkg command queries or manipulates the package database, it uses these exact same patterns. You have the same visibility into the packaging system that the rest of the tools do.

If you want to get a pattern for a specific package, give the package name as a final argument. Here, I get the port the apache24 package came from:

```
# pkg query %o apache24
www/apache24
```

We do have a middle ground between asking all the packages and specific packages, however.

Evaluations in Queries

Here's one last nifty package querying feature. Many—not all, but many—patterns are available as variables. A command can evaluate those variables and take action based on the results. Use the `-e` command line option to evaluate a variable with using a logical operator. A complete list of logical operators appears in pkg-query(8).

Evaluation breaks down into "if this is true, do that." The test goes inside quotes. Here's an example:

```
# pkg query -e '%a = 0' %n
```

This query goes down the whole list of installed packages. The `-e` shows we're evaluating a variable for each package. The statement inside the quotes, `%a = 0`, means we're testing the value of `%a` in that package. If `%a` equals 0, the query evaluates to true and `pkg query` prints out the contents of `%n`. If `%a` equals anything except 0, the statement is false and `pkg query` proceeds to the next package without doing anything.

We already know that `%n` contains the package name. The variable `%a` contains pkg's record of whether or not the package was automatically installed. If you requested this particular package, it's set to 0. If a package was originally installed as a dependency, it's set to 1. So: if a package is not a dependency, print the name. This query prints nonautomatic packages.

```
# pkg query -e '%a = 0' %n
apache24
dmidecode
pkg
youtube_dl
```

A couple things stand out here. First, I didn't deliberately ask pkg to install pkg(8). I requested dmidecode, and pkg bootstrapped itself. The pkg suite itself is always considered a nonautomatic package, though.

Second: who installed youtube_dl on this box?

To find out which packages were installed as dependencies, evaluate whether `%a` is set to 1.

= OR ==?

You'll see the examples using double equal signs, as if `pkg query` were a programming language. My examples use a single equal sign. Surely there's some subtle difference between the two and special conditions under which you should use each?

Nope!

You can use either single or double equal signs, as your

muscle memory prefers.

Realistically, though, I'm not going to bother remembering how to run this query on all my hosts. I need a simple way to make pkg(8) remember it for me.

Pkg Command Aliases

You can define aliases for pkg subcommands in *pkg.conf*. This lets you, say, create aliases to show automatic and nonautomatic commands. I could do something similar in my shell, but it wouldn't show up as pkg(8) subcommands and I'm easily confused.

At the bottom of *pkg.conf*, you'll find a section labeled `ALIAS`.

```
ALIAS             : {
  all-depends: query %dn-%dv,
  annotations: info -A,
--snip--
  }
```

An alias is a single word for the alias name, either a colon or an equal sign, and then the pkg command to run. If you run `pkg all-depends`, pkg(8) looks in *pkg.conf* and runs `pkg query %dn-%dv`. Every alias ends in a colon to indicate that the aliases list continues on the next line.

Many of the aliases in the default configuration represent hangovers from the pkg_add aeon, created for us old timers. The existing aliases are a great place to find sample queries and searches, though. And searching through the aliases turns up this fine entry:

```
noauto = "query -e '%a == 0' '%n-%v'",
```

This alias, *noauto*, runs a `pkg query` command to evaluate `%a` and print the package's name and version number if it's 0. It prints packages that weren't automatically installed. I added a very similar alias to print automatic packages.

```
auto = "query -e '%a == 1' '%n-%v'",
```

When you find yourself repeatedly running complex commands, add

aliases.

Uninstalling Packages

We've all installed software only to rip it out in disgust. The only difference is what, exactly, disgusted us. Uninstall packages with the `pkg delete` subcommand. It's also available as `pkg remove`. That extraneous youtube_dl package? Let's remove it from the system.

```
# pkg delete youtube_dl
Checking integrity... done (0 conflicting)
```

The removal process makes sure that nothing terrible has happened to the package, that nobody else needs it, and that its removal won't do terrible things that the package system can predict.[2]

You'll then get a list of packages to be removed and how much space they'll free up. At the end is a final chance to say no.

```
Proceed with deinstalling packages? [Y/n]: y

[1/1] Deinstalling youtube_dl-2017.02.11...
[1/1] Deleting files for youtube_dl-2017.02.11: 100%
```

The package is deleted from your system.

Removing Dependencies

If you remove a package that other packages depend on, pkg removes the depending packages as well.

```
# pkg delete trousers
--snip--
Installed packages to be REMOVED:
        trousers-0.3.14_1
        gnutls-3.5.9
        emacs-nox11-25.1,3
--snip--
```

The gnutls package needs trousers, and emacs-nox11 needs gnutls. Removing trousers breaks both of them, so pkg figures you clearly don't want them on your system either.

If you really want to delete a package that other packages depend on, add

the `-f` flag.

Read the warnings from `pkg delete` *very* carefully!

Autoremoval

Leaving unnecessary software installed on a host increases the security risks and sysadmin workload. On a long-running system, you don't always know which software to remove. Removing software you chose to install is easy, but that software might have brought along dependencies that you never really paid attention to. Or maybe a new version of a package has fewer or different dependencies than the previous version.

I removed the youtube_dl package from my test system. That leaves me with other packages I deliberately installed and their dependencies. It also leaves the packages youtube_dl depended on but that nothing else needs. The `pkg autoremove` subcommand identifies packages that were installed as dependencies but are no longer required by any other package. It offers to remove these no longer needed. I strongly recommend performing a dry run before removing unneeded dependencies, simply to give your feeble human brain a chance to look at the list twice.

```
# pkg autoremove
```

Pkg runs a database query to identify unneeded dependencies and proposes them for removal.

```
Installed packages to be REMOVED:
        python27-2.7.13_1
        readline-6.3.8
        rtmpdump-2.4.20151223
        librtmp-2.4.20151223
--snip--
```

Study this list carefully. It's not uncommon for a piece of nonpackaged software to need a package that was brought in elsewhere. You probably don't need the video processing tools rtmpdump and librtmp without youtube_dl, but an awful lot of software needs a Python interpreter. Do you *really* want to blow that away?

If you really can remove all these packages, answer `y` and proceed. If one of those dependencies has become critical, though, change your database to tell it so.

Changing the Package Database

Thinking of changing the package database outside of pkg(8)? Don't. You will only cause yourself pain, and your pleas for assistance will be met either with derisive laughter or suggestions to blow away all your packages and start over.

There are a couple circumstances where pkg(8) supports altering the package database, though. That's when you can use `pkg set`. The pkg-set(8) subcommand lets you correctly adjust a few sensible values within the database without corrupting the data. The most common is when you want to make an automatic package no longer automatic.

The `-A` flag to `pkg set` lets you change a package's automatic setting. Setting this flag to 1 means that the package was installed automatically, as a dependency, while a 0 means that the package was specifically requested by the user.

In the previous section, the list of four packages to be deleted by `pkg autoremove` included Python. I want to keep Python—not just this time, but any time I perform autoremovery in the future. The simple way to do that is to change Python from automatic to nonautomatic.

```
# pkg set -A 0 python27
Mark python27-2.7.13_1 as not automatically installed? [Y/n]: y
```

Python is now a nonautomatic package. The results of pkg autoremove now look different.

```
# pkg autoremove -n
--snip--
Installed packages to be REMOVED:
        rtmpdump-2.4.20151223
        librtmp-2.4.20151223
--snip--
```

Only two packages instead of four? Apparently Python needs readline. I'm glad that pkg figured that out for me because I can't be bothered to remember it.

We'll cover `pkg set` more as needed.

Locking Packages

Some software is like a subway's electrified rail. Touching it causes suffering or death.

My favorite example is the remote file synchronization program rsync(8). Rsync has been around for decades, and its internal protocol has changed over time. Many embedded and legacy systems use rsync, but it's never been upgraded. I've spent many painful hours debugging why a current rsync can't communicate with that on a 20th-century embedded phone switch controller. It turned out that an rsync point release dropped support for the very old protocol supported by the phone switch. Upgrading the phone switch wasn't possible, so I needed the rsync package on my host to never upgrade. *Never*.

That's where locking packages comes in.

When you lock a package, pkg won't upgrade, downgrade, uninstall, or reinstall it. It applies the same rules to the package's dependencies and the programs it depends on. The host responsible for fetching the phone switch files needed to have its rsync package locked. Use `pkg lock` to lock a package.

```
# pkg lock rsync
rsync-3.1.2_6: lock this package? [Y/n]: y
Locking rsync-3.1.2_6
```

This package is now nailed in place.

To show all the locked packages on the system, use the `-l` flag. This shows only the packages you've deliberately locked, not the dependents or dependencies.

```
# pkg lock -l
Currently locked packages:
rsync-3.1.2_6
```

Use the `pkg unlock` command to remove the lock.

```
# pkg unlock rsync
rsync-3.1.2_6: unlock this package? [Y/n]: y
Unlocking rsync-3.1.2_6
```

To lock or unlock all packages on the system, use the `-a` flag. You'll get a confirmation prompt for every package, so if you really want to affect all the packages, add the `-y` flag.

```
# pkg unlock -a
```

```
apache24-2.4.25_1: already unlocked
apr-1.5.2.1.5.4_2: already unlocked
--snip--
rsync-3.1.2_6: unlock this package? [Y/n]: y
Unlocking rsync-3.1.2_6
--snip--
```

Package locking doesn't prevent someone with root access from mucking with the files contained in a package.

On a related note, Chapter 22 covers using jails to contain really old software.

Package Files

Files installed by a package are considered system files, and you shouldn't manually edit them. Before you *can* edit those files, you must know what files came with the package. Use `pkg info -l` and the package name to see the complete list. (It's also available as `pkg list`, thanks to a *pkg.conf* alias.)

```
# pkg info -l rsync
rsync-3.1.2_6:
        /usr/local/bin/rsync
        /usr/local/etc/rc.d/rsyncd
        /usr/local/etc/rsync/rsyncd.conf.sample
--snip--
```

Another possibility is that you want to know which package a file came from. Use the `pkg which` command. I normally use this when I've found a weird library and want to know where it came from.

```
# pkg which libp11-kit.so
/usr/local/lib/libp11-kit.so was installed by package p11-kit-0.23.5
```

My question is now, "What is p11-kit?" But that's progress.

Package Integrity

While you shouldn't alter package files, eventually, someone does. You can use pkg to discover those alterations and undo the damage.

The pkg-check(8) tool includes features for identifying damage to packages and package dependencies. Developers can also use pkg-check(8) to check the bundled packages built from ports and distributed to end users,

but that's a whole separate problem.

File Corruption

Verify that a package's files are unaltered with `pkg check -s` and the package name. When my locked rsync package stops synchronizing files from the finicky remote server, one thing I verify is the package integrity.

```
# pkg check -s rsync
Checking rsync:   0%
rsync-3.1.2_6: checksum mismatch for /usr/local/bin/rsync
Checking rsync: 100%
```

Either the disk is failing or someone has mucked with my rsync(1) binary. As this system uses self-healing ZFS, there's gonna be a paddling.

You could uninstall and reinstall the package, but that might trigger changes depending on which packages require the package you're updating. Also, as discussed earlier, this particular package is special. I don't want pkg to upgrade the package to the newest version. Instead, I want to force pkg to reinstall the current package from the package cache. Use the `-f` flag to `pkg install`. While it updates the repository database, it reinstalls the cached package. If the package is locked, you must unlock it first.

```
# pkg unlock -y rsync
Unlocking rsync-3.1.2_6
# pkg install -fy rsync
--snip--
[1/1] Reinstalling rsync-3.1.2_6...
[1/1] Extracting rsync-3.1.2_6: 100%
# pkg lock -y rsync
```

My precious rsync is restored.

Check the integrity of all your packages by running `pkg check -saq`. It produces no output unless something has changed, so you could schedule it via cron (see Chapter 20).

Dependency Problems

If someone really tries, they can delete packages that other packages depend on. Use the `-d` flag of `pkg check` to identify and fix missing dependencies.

```
# pkg check -d emacs-nox11
Checking emacs-nox11: 100%
```

```
emacs-nox11 has a missing dependency: gnutls
emacs-nox11 is missing a required shared library: ❶libgnutls.so.30
--snip--
>>> Try to fix the missing dependencies? [Y/n]: y
```

The first thing to note is that when `pkg check` identifies a missing dependency, it tries to correct it. Answer **y** at the prompt to reinstall the dependency.

Note that this `pkg check` run shows us a missing library, libgnutls.so.30 ❶. The dependency check doesn't actually search for all the files in all of the packages. It knows that this library is missing only because the package that includes it is gone. If you manually remove the library, the dependency check won't find it. You need to check package file integrity, as earlier.

If you want to check all package dependencies with `pkg check -d`, don't give it a package name. You could add `-a` to explicitly check all packages, but that's not necessary. If you add the `-q` flag, this command produces output only when it finds a problem. Adding `-q` also tells `pkg check` to attempt to resolve any dependency problems it finds, without user intervention.

The combination means that while I can run this check as a scheduled job, I'm less comfortable with my host reinstalling a missing dependency. Think about your system installing packages without your attention before automating dependency corrections.

The `pkg check` subcommand includes several other useful options, such as `-B` to rebuild shared library dependencies and `-r` to manually recompute the checksum of an installed package. Read pkg-check(8) for details.

Package Maintenance

The package system includes several maintenance scripts intended to be run from periodic(8). Enable these in */etc/periodic.conf*, as discussed in Chapter 20. Each adds to the daily, weekly, or security status emails.

To have the daily maintenance check package checksums and replace damaged packages, as with `pkg check -saq`, set `daily_status_security_pkg_checksum_enable` to **`YES`**.

To determine whether installed packages have security vulnerabilities published in the FreeBSD package security system, as discussed in Chapter 19, set `daily_status_security_pkgaudit_enable` to **`YES`**.

If you want FreeBSD to back up the installed packages and the package database every day, set `daily_backup_pkg_enable` to **`YES`**.

To be notified of changes in the installed packages, set `daily_status_pkg_changes_enable` to **`YES`**.

Finally, you can check for obsolete packages each week by setting `weekly_status_pkg_enable` to **`YES`**.

Package Networking and Environment

FreeBSD's package system is designed to work for a normal network attached to the internet. That's something of a cruel joke because no network is normal. You can adjust pkg's behavior to fit your network.

The most common change is the need for a proxy server. Pkg uses fetch(3) to download package files, which takes any special networking configuration through environment variables. Set environment variables in the `PKG_ENV` section of *pkg.conf*. Each variable needs the variable name, a colon, and the value. Here, I set the `HTTP_PROXY` environment variable to my network proxy:

```
pkg_env : {
  HTTP_PROXY: "http://proxy.mwl.io/"
}
```

See fetch(3) for the complete list of proxy environment settings.

Some networks have separate bandwidth for different network stacks. I've been on more than one network that has better IPv6 connectivity than IPv4, or the other way. Direct pkg to use one network protocol or the other with the `IP_VERSION` setting in *pkg.conf*. You can set this to 4, 6, or let the host autoselect with the default of 0.

Finally, the *pkg.conf* `NAMESERVER` setting lets you override the nameservers given in */etc/resolv.conf*. Put an IPv4 or IPv6 address here. You can use a hostname here, but pkg will look up that hostname using the default system nameservers.

Package Repositories

You might want to use packages other than those provided by the FreeBSD

Project. Maybe you build your own packages, as discussed in Chapter 16. Perhaps you have access to an experimental package repository. Or maybe you want to switch which set of official packages you're using.

Pkg supports package *repositories*, or *repos*, which are named collections of packages. You can add, remove, enable, and disable repositories.

Normal repositories are very simple, but in rare cases, they can get quite complicated. We won't go into the edge cases of configuring your own repositories, but the basics will take you quite far.

Repository Configuration

Configure each repository in its own file. Official FreeBSD repositories belong in */etc/pkg*. Configure repositories in UCL format (see Chapter 2). FreeBSD ships with the repo FreeBSD enabled. You'll find the configuration file in */etc/pkg/FreeBSD.conf*.

```
  FreeBSD: {
❶ url: "pkg+http://pkg.FreeBSD.org/${ABI}/quarterly",
❷ mirror_type: "srv",
❸ signature_type: "fingerprints",
❹ fingerprints: "/usr/share/keys/pkg",
❺ enabled: yes
  }
```

This repository, named *FreeBSD*, supports the FreeBSD repo. When you decide to set up your own repository, give it a meaningful name.

The `mirror_type` entry ❷ tells pkg whether this repository is hosted on a normal website. Setting `mirror_type` to `NONE` tells pkg to use fetch to get packages using the standard network methods, like HTTP, FTP, or even a file path.

Millions of machines run FreeBSD and need access to the package repository. A single web server can't keep up. By setting `mirror_type` to `srv`, you tell pkg to check DNS for an SRV record. SRV records are used to direct high-availability services, like VoIP and Active Directory.

The `url` entry ❶ shows the internet site where this repository can be found. I'm sure you've seen http URLs before, but what about this pkg+http thing? It ties the request to the SRV record used to direct pkg requests, as

set by `mirror_type`.

The package system can verify downloaded packages with public keys or cryptographic hash fingerprints ❸. You'll need to tell pkg where to find the keys or hashes, though ❹.

Finally, you must explicitly enable or disable ❺ each repository.

Repository Customization

You can add and remove repositories as needed. As */etc/pkg* is reserved for official FreeBSD repositories, you'll need another directory. The traditional location is */usr/local/etc/pkg/repos*. If you want to use a different directory, you'll need to set a location in *pkg.conf* with the `REPO_DIRS` option. You'll see commented-out examples for the defaults.

```
#REPOS_DIR [
#    "/etc/pkg/",
#    "/usr/local/etc/pkg/repos/",
#]
```

The local repository directory doesn't exist by default, so you'll need to create it.

```
# mkdir -p /usr/local/pkg/repos
```

Put your own repository configurations in that directory.

FreeBSD searches for packages in directory order, checking directories in the order given in `REPOS_DIR`. The obvious implication is that the default FreeBSD repo can't be disabled or overridden. That's not quite true, but the reason is a little tricky.

Repository Inheritance

You can split a repository's configuration between multiple files. Entries in later files overwrite the entries in earlier files.

To see how this works, consider the default repository, called *FreeBSD*. If you have all of your custom repositories configured in */usr/local/etc/pkg/repos*, pkg finds the FreeBSD repo first.

But now create a */usr/local/etc/pkg/repos/FreeBSD.conf* file. Define the

FreeBSD repo in there, but include only a single configuration statement.

```
FreeBSD: { enabled: no }
```

Pkg finds the repo named FreeBSD first in */etc/pkg/FreeBSD.conf*. This configuration defines the defaults for this repo. It finds the second configuration later. The second configuration overrides only one option, but that option turns off the repository.

While disabling the FreeBSD repository is an extreme case for folks who don't run their own repository, there's good reason to make minor changes to the repo, as we'll see next.

Package Branches

FreeBSD's packages are built from the Ports Collection (see Chapter 16). The Ports Collection attempts to bring tens of thousands of different software suites to FreeBSD. These different programs all have their own release schedules, and the Ports Collection evolves continuously in an effort to keep up with them. As you can imagine, the Ports Collection has a whole bunch of churn. Most of us who run servers want stability. When most sysadmins consider "stability," the word *churn* isn't what comes to mind.

Most of us don't need the very latest software on our servers. Most of the time, I'm fine if my database server is a minor point release or two behind; I care only that it keeps working. I'm certainly not going to upgrade my servers just because MySQL or nginx or PHP has a new software version. That way lies the madness of constant upgrades.

I do want security and stability updates, however. The database server being a little older doesn't bother me. The database server occasionally losing its brain and sending all my data to the bit bucket, or offering everything to a Detroit hacking crew, bothers me a whole bunch.

The FreeBSD package system's *quarterly branches* try to strike a middle ground between the world's constant churning software and a sysadmin's peace of mind. Every January, April, July, and October, the Ports crew forks the Ports Collection into a quarterly branch. The quarterly branch receives only security and stability updates, while the main Ports Collection charges merrily ahead.

The FreeBSD Project builds two sets of packages for each release. The *quarterly* packages are built from the quarterly Ports Collection. The *latest* packages are built from the bleeding-edge packages.

Some of you prefer the most current packages, despite the churn. That's okay. Switching is simple. You need override only one entry in the FreeBSD repository. Create a new repository file, */usr/local/etc/pkg/repos/FreeBSD.conf*, just as in the last section. Rather than disabling the default repository, though, we're going to override the package source. Change the "quarterly" and the end of the URL to "latest."

```
FreeBSD: { url: "pkg+http://pkg.FreeBSD.org/${ABI}/latest" }
```

Welcome to the churn!

It's strongly recommended to run `pkg update -f` after changing repositories in order to force pkg to download the latest repository catalogs.

Switching package collections doesn't necessarily mean you need to reinstall everything. If your old packages work, use them. If weird problems appear, though, reinstall all of your packages with a command like `pkg upgrade -fa`. Even packages that have the same version as those in the other package collection might be subtly different.

Upgrading Packages

As much as we might wish it were otherwise, you can't set up a new system and ignore it. Either stability bugs appear or some clever jerk figures out a security exploit. (Chapter 19 discusses auditing package security.) Sometimes you must upgrade your third-party software. With FreeBSD's original packaging system, pkg_add, package upgrades risked a certain degree of heartache. With pkg(8), you still risk heartache—but it's from the newer versions of the software, not the packaging system itself.

Before upgrading packages, back up your system. Then, use the `upgrade` subcommand to have pkg(8) upgrade all your packages. I recommend running a dry run first, with `-n`.

```
# pkg upgrade -n
--snip--
Checking for upgrades (2 candidates): 100%
Processing candidates (2 candidates): 100%
```

```
The following 1 package(s) will be affected (of 0 checked):

Installed packages to be UPGRADED:
        ca_root_nss: 3.29.1 -> 3.29.3

Number of packages to be upgraded: 1

335 KiB to be downloaded.
```

Carefully look at the list of packages to be upgraded. Are there any large jumps? Do you need to look at any release notes? How intrusive is this likely to be? Does the upgrade remove any packages that you want, like *X.org* or your text editor? Should you wait until Sunday at 3 AM and have your flunky do it?[3] Studying the upgrade and considering the risks of each package upgrade might not reduce the amount of work you need to do, but it will reduce the amount of downtime and the amount of time people yell at you.

Once you're comfortable with what will change, run the upgrade.

```
# pkg upgrade -y
```

You'll see very similar messages about the packages to be upgraded and then notifications of the download and install process. Finally, pkg displays the installation message for every upgraded package.

Even the world's most flexible packaging system won't always meet your needs. FreeBSD makes customizing add-on software very easy through the Ports Collection, where we'll go next.

16
CUSTOMIZING SOFTWARE WITH PORTS

Packages provide the most common configurations of the most desirable programs. If you're building a generic web server, chances are that the official FreeBSD package of nginx or lighttpd or whatever your preferred web server is will suffice. If you have a special environment or less common needs, that's where the Ports Collection comes in. The Ports Collection is a tool for easily building customized versions of many software packages. It combines dependency, licensing, maintainer, and all other software information in a standard machine- and human-readable format. Ports let you set system options like "forbid third-party GPL-licensed code" (useful for embedded vendors), "add LDAP to everything," or "disable X11."

In the long term, ports are best managed with the poudriere package-building system. Before you can use poudriere, though, you must understand how ports work. I'd encourage you to explore ports on a test system. Rather than deploying ports on your individual servers, though, use poudriere to build your own package repository. Manage your servers entirely with packages. Never use the ports tree on a production server other than your package builder.

Before we dive into ports, let's talk about building software in general.

Making Software

Traditional software building is complicated because source code must be processed very specifically to create a workable, running program—let alone one that works well! It's a completely different process than using, say, a

JavaScript compiler. While programmers could include installation instructions with each program, full of lines like `Now type ar cru .libs/lib20_zlib_plugin.a istream-zlib.o zlib-plugin.o`, this would be downright sadistic. While Unix admins might seem to approve of sadism, they categorically disapprove of cruelty directed at themselves; if something can be automated, it will be.

The main tool for building software is make(1). When run, make looks in the current directory for a file called *Makefile*, which is full of instructions much like that horrid example in the previous paragraph. It reads the instructions and carries them out, automating the installation process no matter how complicated it might be. You don't really have to know the internals of a *Makefile*, so we're not going to dissect one.

Each *Makefile* includes one or more *targets*, or sets of instructions to carry out. For example, typing `make install` tells make(1) to check the *Makefile* for a target called *install* and, if found, execute it. A target's name usually relates to the process to be carried out, so you can safely assume that `make install` installs the software. You'll find targets to install, configure, and uninstall most software. make(1) handles a huge variety of functions, some of which far outstrip the creators' original intents. But that's part of the fun of Unix!

Source Code and Software

Source code is the human-readable instructions for building the actual machine code that makes up a runnable program. You've probably been exposed to source code in some form. If you've never seen it, take a look at a few files under */usr/src* or at *https://svnweb.freebsd.org/*. Even a neophyte sysadmin needs to recognize source code two tries out of three.

Once you have source code for a program, you build (or *compile*) the program on the type of system you want to run it on. (Building software for a foreign platform via cross-compiling demands is more complicated when it's possible.) If the program was written for an operating system that's sufficiently similar to the platform you're building it on, it works. If your platform is too different from the original, it fails. Once you've successfully built the software on your system, you can copy the resulting program (or *binary*) to other identical systems, and it should run.

Some programs are sufficiently well written that they can be compiled on

many different platforms. A few programs specifically include support for widely divergent platforms; for example, the Apache web server can be compiled on both Windows and Unix-like systems. This represents heroic effort by the software authors, and even so, you must run a few scripts and configure your environment precisely by the directions before building on Windows.

Generally speaking, if you can build a program from source, it will probably run. It might not run correctly, it might not do anything you expected, but it runs. A sufficiently experienced sysadmin can use the source code and error messages to learn why a program won't build or run. In many cases, the problem is simple and can be fixed with minimal effort. This is one reason why access to source code is important.

Back when every sysadmin was a programmer, debugging software absorbed a major part of the admin's time. Every Unix-like system was slightly different, so every sysadmin had to understand his platform, the platform the software was designed for, and the differences between the two before he could hope to make a piece of code run. The duplication of effort was truly horrendous.

Over the years, programmers developed tools such as autoconf to help address these cross-platform issues. Not every program used these tools, and when they broke, the sysadmin was kicked back to square one. Sysadmins had to edit the source code and *Makefiles* just to have a chance of making the programs work. And *working* isn't nearly the same as *working well*, let alone *working correctly*.

The FreeBSD Ports Collection was designed to simplify this process for FreeBSD users.

The Ports Collection

The *Ports Collection*, also called the *ports tree* or simply *ports*, contains an automated system for compiling software on FreeBSD.

The basic idea behind the ports system is that if source code must be modified to run on FreeBSD, the modifications should be automated. If you need other software to build this program from source code or to run the software, those dependencies should be documented and tracked. If you're going to automate the changes, you might as well record what the program

includes so you can easily install and uninstall it. And since you have a software-building process that produces exactly the same result each time, and you've recorded everything that the process creates, you can copy the binaries and install them on any similar system.

In addition to the information needed to create the packages, the Ports Collection contains legal restrictions on building the software, security information, licensing details, and more.

Ports interoperate with packages. The Ports Collection is used to create packages. You can install some software from ports and some from packages as you need, freely mixing where you install software from. You'll need to use the same version of the Ports Collection used to build your packages, either a quarterly branch or the latest version. Most ports users want the latest software, so we'll focus on that.

Ports

A *port* is a set of instructions on how to apply fixes to, or *patch*, a set of source code files and then build and install those files. A port contains a complete record of everything necessary to create the finished software. This frees sysadmins from struggling to install programs and lets them struggle to configure them.

Ports Tree Installation

If you followed the installation instructions in Chapter 3, you installed the ports tree in */usr/ports*. In that directory, you should find several files and a couple dozen directories. If you don't have anything in */usr/ports*, you apparently can't follow instructions. That's okay—I can't either—but you must install the ports tree to continue.

FreeBSD supports a couple different ways to get the ports tree. You can check it out using svn(1) or download a copy off the web. The recommended method for sysadmins is to use portsnap(8) to download the latest (nonquarterly) version of the ports tree.

```
# portsnap auto
Looking up portsnap.FreeBSD.org mirrors... 6 mirrors found.
Fetching snapshot tag from your-org.portsnap.freebsd.org... done.
Fetching snapshot metadata... done.
Updating from Mon Oct 17 15:59:41 EDT 2018 to Mon Mar 20 14:13:53 EDT 2019.
```

```
Fetching 5 metadata patches... done.
Applying metadata patches... done.
Fetching 5 metadata files... done.
Fetching 10202 patches.
(700/10202) 6.86% .........
```

Here, portsnap searches for a mirror of the portsnap files, cryptographically verifies the integrity of those files on the portsnap server, downloads the files, and verifies the integrity of the download itself.

You now have all the latest versions of all FreeBSD ports. To update an existing Ports Tree to the latest version, run `portsnap auto` again.

If you wish to schedule a regular portsnap update run via cron(1), use the `portsnap cron update` command instead of `portsnap auto`. This tells portsnap to update the ports tree at some random time within 60 minutes of the command running. This helps distribute the load on the FreeBSD portsnap server. Schedule a portsnap run at some point between 5 AM and 5:59:59 AM in root's crontab with an entry like this:

```
0    5    *    *    *    /usr/sbin/portsnap cron update
```

This kicks off the actual update at a random time between 5 AM and 6 AM, which is *much* more effective than 1 out of 24 portsnap users hitting the download server simultaneously at 5 AM.

Ports Tree Contents

Most of the directories you see here are software categories. Each category contains a further layer of directories, and each of those directories is a piece of software. FreeBSD has over 28,000 ports as I write this, so using the directory tree and categorizing software properly is vital. Of the files and directories in this category that aren't software categories, the major ones are described here.

The *CHANGES* file lists changes made to the FreeBSD ports infrastructure. It's primarily of use to the FreeBSD ports developers and people interested in the internals of the Ports Collection.

The *CONTRIBUTING.md* file exists for FreeBSD source code mirrors on GitHub. All FreeBSD source code is mirrored on GitHub for people's convenience, but FreeBSD doesn't use Git internally. GitHub users traditionally check *CONTRIBUTING.md* for information on how to

contribute—which, in FreeBSD's case, is "go to the FreeBSD website." (Work on automatically feeding GitHub pull requests into the FreeBSD PR system is ongoing as I write this.)

COPYRIGHT contains the licensing information for the Ports Collection as a whole. While each individual piece of software supported by the Ports Collection has its own copyright and licensing information, the Ports Collection is licensed under the two-clause BSD license.

The *GIDs* file contains a list of all the group IDs used by software in the Ports Collection. Many pieces of software in the collection expect to run as an unprivileged user. If each port gets to create a random user, the usernames, user IDs, and group IDs will overlap. Instead, ports that need an unprivileged GID reserve one in this file. This file records GIDs assigned to Ports Collection. GIDs aren't actually assigned in */etc/passwd* until used.

Your */usr/ports* has an *INDEX* file with a suffix named after the version of FreeBSD you're running. This FreeBSD 12 system has */usr/ports/INDEX-12*. The ports system's search and description features use this index. The index is generated locally and not stored in Subversion.

The *Keywords* directory contains information for the Universal Configuration Language system, discussed in Chapter 23.

LEGAL describes the legal restrictions on any software in the Ports Collection. Some pieces of software have specific limitations on them—such as no commercial use, no redistribution, no monetary gain, and so on. Individual ports also list these restrictions; this is just a master list built from all the ports.

MOVED lists all the ports that have been renamed, moved from one category to another, or removed, along with the reason why. Automated management tools such as portmaster(8) use this list to find the new home of moved ports. Why move a port? When I started with FreeBSD, it had one category for X Windows software. The category grew ridiculously huge, so the ports team split it, and split it again, until we reached 2017's nine categories.

The *Makefile* contains high-level instructions for the whole Ports Collection. You'll only use this if you want to build every port in the entire Ports Collection. You'd be better off using poudriere as discussed in "Private Package Repositories" on page 381 than just running `make` here.

The *Mk* subdirectory contains the logic that drives make(1) in fetching

source files from the internet, patching them, building them, and installing them. Many types of programs expect to integrate together, and these files ensure that different parts of the same tool are built and installed in a compatible manner. Some features, like LDAP and Emacs, can touch many ports. This directory contains Makefiles like *bsd.ldap.mk* and *bsd.emacs.mk* for exactly these functions.

Beneath the *Mk* subdirectory, you'll find *Uses*. This directory contains broadly used *Makefiles* for other widely used functions or software suites. For example, the KDE and GNOME desktop suites include dozens or hundreds of smaller programs, and each must be built correctly to interoperate. If you look in *Uses*, you'll see the files *gnome.mk* and *kde.mk* dedicated to configuration of these programs, as well as files for GSSAPI, Lua, Varnish, and many other software families.

If you really want to learn how the Ports Collection works, read everything in */usr/ports/Mk* and */usr/ports/Mk/Uses*. It's highly educational, even though the nature of supporting all these different programs means the *Makefiles* are as tangled as a yarn basket attacked by a horde of crazed kittens.

The *README* file contains a high-level introduction to the Ports Collection.

The *Templates* directory contains skeleton files used by other portions of the Ports Collection.

The *Tools* directory contains programs, scripts, and other automation, mostly used by ports developers.

The *UIDs* file contains unprivileged user IDs used by ports in the system. Much like the *GIDs* file, this helps the ports developers avoid conflicts between unprivileged users required by ported software.

UPDATING contains notes for use when upgrading your software. Updates that require special intervention appear here in reverse date order. Before updating your software, check this file for important notes that affect you.

The *distfiles* directory contains the original source code for ported software. When a port downloads a chunk of source code, that source code is kept under */usr/ports/distfiles*.

All the other directories are categories of ports. The following shows the contents of the *ports/arabic* directory, where software specific to the Arabic language is kept. Much software elsewhere in the Ports Collection supports

Arabic, but this category is for software focused on Arabic—such as fonts, translations of certain types of documents, and so on. This category isn't useful for most people, but it has the serious advantage of being small enough to fit in this book. Some ports categories have hundreds of entries.[1]

```
Makefile        ae_fonts_ttf    kacst_fonts     libitl
Makefile.inc    arabtex         kde4-l10n       libreoffice
ae_fonts_mono   aspell          khotot
```

This *Makefile* contains instructions for all the ports in the directory. They're more specific than the global *Makefile* in */usr/ports*, but not as specific as individual port Makefiles. The file *Makefile.inc* contains meta-instructions for the ports in this directory. All the other directories are individual software packages. We'll dissect one of those directories in "Installing a Port" on page 371.

Individual ports are often called by their directory in the Ports Collection. The gnuplot graphing program might be called math/gnuplot, as its port can be found at */usr/ports/math/gnuplot*.

The Ports Index

The ports index file contains a list of all ports that build on a particular FreeBSD release. On FreeBSD 13, this is */usr/ports/INDEX-13*. The Ports Collection uses the index for several purposes, including searching the whole ports tree.

The index file describes each port on a single line, with fields separated by pipe symbols (`|`). While this is convenient for system tools, it's not particularly human-readable. Run `make print-index` in */usr/ports* to get a longer, much more intelligible index. This index is filled with entries like this:

```
Port:   p5-Archive-Extract-0.80
Path:   /usr/ports/archivers/p5-Archive-Extract
Info:   Generic archive extracting mechanism
Maint:  perl@FreeBSD.org
Index:  archivers perl5
B-deps: perl5-5.24.1
R-deps: perl5-5.24.1
E-deps:
P-deps:
F-deps:
WWW:    http://search.cpan.org/dist/Archive-Extract/
```

The index starts with the port's name and the full path to the port directory. `Info` gives a very brief description of the port. The `Maint` heading lists the port's maintainer, a person or team who has assumed responsibility for this software's integration into the Ports Collection. The `Index` space lists every category where this port might be filed. The first category listed is the directory where it appears in the Ports Collection. In this case, the port appears in the *archivers* directory.

We then have dependencies. `B-deps` lists the build dependencies—that is, other software that must be installed to build this port. `R-deps` lists runtime dependencies, software needed for this to actually run. This is a Perl module, so it needs a Perl interpreter. Some software must be extracted or decompressed by particular tools, specified in `E-deps`. The `P-deps` field lists any dependencies for patching the software—rare pieces of software must be patched with a certain tool. The `F-deps` field is similar, specifying *fetch dependencies*—that is, any special software that must be used to download the software.

Finally, the `WWW` space gives the home page of the software.

Searching the Index

The Ports Collection includes tools to search the index. If you want a particular program, you might be better off finding the ports directory with `pkg search` or even `locate -i`. Reserve searching the Ports Collection to answer questions like "What ports use SNMP?"

If you know the name of a piece of software, search *INDEX* for it with `make search`. Here, I look for ports with names that include *net-snmp*:

```
   # cd /usr/ports
   # make search name=net-snmp
❶ Port:   net-snmp-5.7.3_12
   Path:   /usr/ports/net-mgmt/net-snmp
   Info:   Extendable SNMP implementation
   Maint:  zi@FreeBSD.org
   B-deps: perl5-5.24.1
   R-deps: perl5-5.24.1
   WWW:    http://net-snmp.sourceforge.net/

   Port:   p5-Net-SNMP-6.0.1_1rt:   p5-Net-SNMP-365-3.65
   --snip--
```

As of this writing, FreeBSD has several ports with *net-snmp* in their name.

The first is the current standard net-snmp software collection ❶. Others include Perl libraries that use SNMP over the network but otherwise have nothing to do with the net-snmp suite, old versions of net-snmp that are no longer supported, and Tcl/Tk interfaces to net-snmp. The fields in the description are taken straight from the *INDEX* file.

If you don't need this much detail, try `make quicksearch` to get only the port, path, info, and (if applicable) notes on reasons why it's not there anymore.

Key Searches

You can also search using any of the fields in the port description as a key. Remove any hyphens from the key name. You want all the ports that have a runtime dependency on Perl?

```
# make quicksearch rdeps=perl5
```

You can combine multiple search terms in one query. Suppose you want all the programs with Apache in the name but with a runtime dependency on Python.

```
# make quicksearch name=apache rdeps=python
```

Exclude a word from the search results by putting an *x* in front of the key. Here, we look for everything that has a runtime dependency on Python but *doesn't* have Apache in the name:

```
# make quicksearch xname=apache rdeps=python
```

These by-field searches don't work for all software, however. For example, if you're looking for the Midnight Commander file manager, you might search for it by name.

```
# make search name=midnight
#
```

Well, that was less than helpful. Search all the fields for a match with the term `key`.

This scans more fields and returns more hits. If you're searching for a common word, however, the `key` search can provide far too much information. Trim the output with `quicksearch`.

```
# make quicksearch key=midnight
```

This returns every port with the string `midnight` in its description, name, or dependencies. We'll quickly learn that Midnight Commander can be found under */usr/ports/misc/mc*.

Other Ways to Browse the Ports Collection

If you prefer using a web browser, build an HTML index. Just go to */usr/ports* and, as root, type `make readmes` to generate a *README.html* file with the index of your ports tree and a HTML file in every port. You can click through various categories and even view detailed descriptions of every port.

If none of these options work, try the FreeBSD Ports Tree search at *http://www.freebsd.org/cgi/ports.cgi*. Also, the FreshPorts search engine at *http://www.freshports.org/* provides a separate but very nice search function.

Between the web browser and the search engine, you should be able to find a piece of software to meet your needs. Finding the port you need might well be the most difficult part of working with ports.

Legal Restrictions

While most of the software in the Ports Collection is free for any use, some of it has a more restrictive license. The */usr/ports/LEGAL* file lists legal restrictions on the contents of the Ports Collection. The most common restriction is a prohibition on redistribution; the FreeBSD Project doesn't include such software on its FTP sites or on a CD image but provides instructions on how to build it.

Legal restrictions appear in places you might not expect. You can't download a compiled, ready-to-go package for Oracle Java, and the FreeBSD Project can't redistribute the Java source code. FreeBSD can and does distribute instructions on how to build the Oracle Java source code on FreeBSD, but the user must go to the Oracle site and download the code themselves. Fortunately, OpenJDK has supplanted Oracle Java for most software, and FreeBSD has a high-quality package for it.

Similarly, some pieces of software prohibit commercial use or embedding in commercial products. A few cannot be exported from the United States, thanks to Department of Commerce rules restricting the export of

cryptography.[2] If you're building FreeBSD systems for redistribution, export, or commercial use, you need to check this file.

Fortunately, most of the software in the Ports Collection is free for either commercial or noncommercial use. These restricted packages are the exception, not the norm.

What's In a Port?

Installing software from ports takes longer than using packages, and the Ports Collection requires a live internet connection. In exchange, the Ports Collection can produce more optimal results than packages. Let's take a look at a port. Here's the innards of dns/bind911, version 9.11 of the ISC BIND nameserver:

```
Makefile        files           pkg-help
distinfo        pkg-descr       pkg-plist
```

The *Makefile* contains the basic instructions for building the port. If you read this file, you'll quickly find that it's only a few hundred lines long. That's not a huge amount of instructions for such a complicated piece of software, and most *Makefiles* are much shorter. Most of that file is dedicated to customizations that are only rarely used. There's almost no information about BIND itself in here and not much about how to build software on FreeBSD. Most of the FreeBSD ports system's *Makefiles* are in */usr/ports/Mk*.

The *distinfo* file contains checksums for the various files the port downloads so that your system can be sure that the file transferred without error and that nobody tampered with the file before you got it.

The *files* directory contains all the add-on files and patches required to build this port on FreeBSD. BIND 9.11 takes a dozen patches. Most of these patches aren't required for building, as the ISC supports their DNS servers on FreeBSD. They provide integration only into the FreeBSD package system.

The file *pkg-descr* contains a lengthy description of the software.

A few ports include a *pkg-help* file that offers additional details on how to use the port.

Some ports (not this one) have a *pkg-message* file that contains a template

used to create the package's installation message.

Finally, the *pkg-plist* file is a list of all the files installed (the "packing list"). The port installs only the files listed in the packing list. Some ports (such as Python-related ones) use an automatically generated packing list, so don't be surprised if the packing list is missing.

Combined, these files comprise the tools and instructions needed to build the software.

Installing a Port

If you're familiar with source code, you've probably already noticed that a port contains very little actual source code. Sure, there are patches to apply to the source code and scripts to run on the source code, but no source code for the software! You might rightly ask just how building software from source is supposed to work without source code?

PORTS AND PRODUCTION

I would strongly encourage you to build your own package repository with poudriere and manage your servers' ports from that repository. Upgrading ports directly installed on a host is annoying and difficult. Tools like portmaster and portupgrade are obsolete at this moment, and while they might get updated or rewritten, poudriere is the eternal method. You have been warned. Explore ports on a disposable test system.

When you activate a port, FreeBSD automatically downloads the appropriate source code from an included list of sites. The port then checks the downloaded code for integrity errors, extracts the code to a temporary working directory, patches it, builds it, installs everything, and records the installation in the package database. If the port has dependencies, and those dependencies are not installed, it interrupts the build of the current port to build the dependencies from source. To trigger all this, you just go into the

port directory and type:

```
# make install
```

You'll see lots of text scroll down your terminal as the port carries out its work, and you'll get your command prompt back when it finishes.

As you grow more experienced in building from source, however, you'll find that this all-in-one approach isn't appropriate for every occasion. Not to worry; the Ports Collection provides the ability to take the port-building process exactly as far as you like because `make install` actually runs a whole series of subcommands. If you specify one of these subcommands, make(1) runs all previous commands as well as the one you specify. For example, `make extract` runs `make config`, `make fetch`, `make checksum`, `make depends`, and `make extract`. These subcommands are, in order:

make config

Many ports have optional components. Running `make config` lets you select which of those options you wish to support in this port. The options you select are saved in */var/db/ports* for future builds of the port. These options affect how the port is built—for example, if you choose to build a program with net-snmp support, you're adding a dependency on net-snmp. We discuss `make config` in more detail in "Port Customization Options" on page 373 later in this chapter.

make fetch

Once you've configured the port, the system searches a preconfigured list of internet sites for the program source code. The port's *Makefile* might list the authoritative download site for the file, or it might use one of several authoritative lists provided by the Ports Collection. When the port finds the source code, it downloads it. The original, downloaded source code is called a *distfile* and is stored in */usr/ports/distfiles*.

If the port requires a particular program to fetch a distfile, the port installs that program as part of `make fetch`.

make checksum

The `make checksum` step computes the distfile's cryptographic hash and

compares it to that recorded in the port's *distinfo* file. Files can be corrupted in any number of ways: during download, by malicious intruders on a download site, or sheer random what-the-heck. Checksum verification detects file damage from any cause and stops the build if the files are corrupt.

This step makes no effort to determine why or how the file was corrupted. For the port's purposes, it doesn't matter whether the source code was corrupted during download or some malicious intruder put his backdoor code into the distfile before you downloaded it. Either way, don't waste time building it, and certainly don't install it!

FOOT-SHOOTING METHOD #839: IGNORING THE CHECKSUM

Software authors, especially free software authors, sometimes make minor changes to their code but don't change the software version or the filename of the distfile. The FreeBSD port rightfully notices this problem and doesn't work after such a change. If you're absolutely certain that the distfile hasn't been compromised or corrupted, you can override this check with `make NO_CHECKSUM=yes install`.

I highly recommend consulting the software's original author—not the port maintainer—before doing so. Checking with the author ensures that you're not installing compromised software and also helps educate the software author about the importance of version numbers and release engineering.

make depends

A lot of software is built on top of other software. While FreeBSD includes make(1) and a compiler, some software can be compiled only with a particular compiler or demands a certain version of make. Maybe the distfile is distributed compressed with a rarely used algorithm. Perhaps it needs a third-party library that doesn't come with FreeBSD. At the `make depends` stage, the port checks for missing dependencies and attempts to resolve them by

building the ports.

Dependencies can have their own dependencies. The `make depends` recursively processes dependencies until the port has everything it needs to build, install, and run.

make extract

Once FreeBSD has the port distfiles, it must uncompress and extract them. Most source code is compressed with something like gzip(1), bzip(1), or xz(1), and collated with tar(1). This command creates a *work* subdirectory in the port and extracts the tarball there. If the port requires a particular program to extract the distfile, it will install it now.

make patch

This command applies any patches in the port to the extracted source code in the *work* subdirectory. If the port requires a special patch program instead of the base system's patch(1), the port installs it now.

make configure

Next, FreeBSD checks to see whether the software has a configure script. This isn't the same as the `make config` step performed by the port. If the software came with its own configure script, the port runs it. Some ports interrupt the build at this stage to prompt you for information, but most run silently.

make build

This step compiles the checked, extracted, patched, and configured software. Ports that don't compile anything might have an empty step here. Some ports exist only to conveniently package a bunch of other ports.

make install

Finally, `make install` installs the software and tells the package system to record its presence.

Port Customization Options

Many software packages have extensive custom-build features. While enabling these features isn't hard for any individual piece of software, there's no universal method for defining them. With one piece of software, you might have to edit the original software's *Makefile*; with another, you may have to offer flags to the configure script. Learning how to make these changes takes time and can be an annoyance. The FreeBSD Ports Collection offers two ways to consistently configure these options on your system.

The newer, prettier method is supported by `make config`. This brings up a dialog box much like those you saw when you first installed FreeBSD. For example, the popular access control system sudo (*http://www.sudo.ws/*) includes support for LDAP, auditing, and, most vitally, insulting the user when they enter their password incorrectly. If you go to */usr/ports/security/sudo* and type `make config`, you'll see a menu much like the one shown in Figure 16-1.

Use the spacebar to select options you like and the arrows and TAB key to move around. Hit ENTER over either OK or Cancel to finish. The port records your desired options in */var/db/ports/<category>_<portname>/options*. When you have to rebuild or upgrade the port, the port reuses those same options unless you run `make config` to change them or `make rmconfig` to blow them away.

Figure 16-1: Port configuration

Customizing at the Command Line

Sometimes you don't want a pretty arrow-select menu but a proper sysadmin interface of words on a command line. The Ports Collection lets you skip the menu and give all the configuration options in the make(1) command.

Before you can do that, you'll want to turn off the pretty menu. Set the environment variable `BATCH=1` on the command line to turn off the menu. Here, we build the port with the default configuration, exactly as the FreeBSD package cluster does:

```
# make BATCH=1 install
```

Now that you've ditched the annoying menu, see what configure options the port supports. The `make pretty-print-config` command displays the current settings in an easily readable format. Let's check out *security/sudo*.

```
# make pretty-print-config
+AUDIT -DISABLE_AUTH -DISABLE_ROOT_SUDO +DOCS -INSULTS -LDAP +NLS -NOARGS_
SHELL -OPIE -SSSD
```

Each of these represents a configuration option. Options marked with a plus are turned on, while those flagged with a minus are turned off. What do these options mean? Running `make showconfig` displays all the port's options and what they do.

```
# make showconfig
===> The following configuration options are available for sudo-1.8.19p2:
     AUDIT=on: Enable BSM audit support
     DISABLE_AUTH=off: Do not require authentication by default
     DISABLE_ROOT_SUDO=off: Do not allow root to run sudo
     DOCS=on: Build and/or install documentation
     INSULTS=off: Enable insults on failures
--snip--
```

While sudo supports LDAP and SSD and all sorts of complicated information sources, what I truly need is for sudo to insult the user any time he enters an incorrect password. I want the `INSULTS` option. Use the `WITH` environment variable on the command line to set the option. Option names are case-sensitive. Here, I set the option and check the configuration again:

```
# make WITH=INSULTS pretty-print-config
+AUDIT -DISABLE_AUTH -DISABLE_ROOT_SUDO +DOCS +INSULTS -LDAP +NLS -NOARGS_
SHELL -OPIE -SSSD
```

The `INSULTS` option is now set.

Use quotation marks to enable multiple options.

```
# make WITH="INSULTS LDAP" pretty-print-config
```

Similarly, use WITHOUT to turn off an option.

```
# make WITH=INSULTS WITHOUT="AUDIT NLS" pretty-print-config
```

If you leave the menu enabled when building the port, the make config graphical menu appears, but with your selected options set. Remember, turn the menu off with the BATCH variable.

Using Customizations Globally

You build ports to get specific features in your software. Often, you want that feature in all the ports that support it. Consider LDAP for a moment. If your enterprise uses LDAP, you probably want all of your software to use it. You'll want LDAP to be the default.

FreeBSD stores settings used for every run of make in */etc/make.conf*. Here's where you'd enable LDAP or LibreSSL or other customizations that should appear across the system. Put any options you want applied globally in *make.conf*. Unlike the command line, *make.conf* uses the variables OPTIONS_SET and OPTIONS_UNSET.

Here, I want the options LDAP and INSULTS enabled on every port:

```
OPTIONS_SET=INSULTS LDAP
```

A *make.conf* setting has no effect on a port that doesn't support the option. Many ports don't know anything about LDAP. I don't know whether any ports other than sudo include an optional feature to insult my users, but if the feature's available, I *need* it.

Why use separate options in *make.conf* as opposed to the command line? Precedence. Options applied using WITH override options set using OPTIONS_SET. In this example, I've enabled insults globally. If for some unfathomable reason I needed a particular port not to insult the user,[3] I could use WITHOUT=INSULTS on the command line when building the port to override the global default.

/etc/make.conf and Single Ports

Perhaps you want to build a particular port with a specific option, but you don't want to specify it on the command line. Use the port category, an underscore, the port name, another underscore, and the SET variable in

/etc/make.conf.

```
security_sudo_SET=INSULTS
```

While the port should cache the configuration, this would provide additional protection against fat-finger mistakes.

Setting Default Versions

FreeBSD supports dozens of port customization choices. Not all of them are sensible as port options, though. Some options must be used across the entire ports collection to be effective. The most common example is the SSL library. You can build all of your ports with the base system SSL library, and things will work fine. You can build all the ports with an external SSL library and, again, the software works. Building some ports with the base system SSL and some with a third-party SSL leads to catastrophe. The same applies to, say, different versions of the PostgreSQL database server and the Python interpreter. Different SSL libraries combined with different database server versions creates the sort of debacle I really enjoy handing off to a junior sysadmin who desperately needs an unforgettable lesson in how shared libraries work.

The Ports Collection uses the `DEFAULT_VERSIONS` variable to list critical software that should be used as the default. This replaces older variables like `DEFAULT_MYSQL_VER` and `WITH_BDB_VER`. The only way to get the complete list of variables is to trawl through */usr/ports/Mk/*. The file *bsd.default-versions.mk*, *bsd.apache.mk*, and the files under *Mk/Uses* are notably useful.

Here, I'm telling the Ports Collection always to build ports with LibreSSL instead of the base system's OpenSSL library and to use Python 3.7.

```
DEFAULT_VERSIONS += ssl=libressl
DEFAULT_VERSIONS += python=python3.7
```

I list each default version on a separate line and use the `+=` syntax to tell the ports system to add this to the list.

I recommend setting default versions before building your first port. Otherwise, you'll wind up rebuilding ports so that they link against your preferred libraries.

Don't mix prebuilt packages with ports built using an alternate

`DEFAULT_VERSIONS`. Programs built from packages will use the default libraries, while your ports will use your preferred libraries. If your system works afterword, it will be only by sheer accident.

Front-Loading Recursion

Sometimes the interactivity in building a port isn't the problem. Recursion is the problem.

Suppose you're building a big port, such as LibreOffice or GNOME. These ports had dozens or even hundreds of dependencies. Many of these ports require interactive configuration. Perhaps you decide to launch a KDE build before going to bed, thinking that you'll wake up with the latest window manager or at least an amusing error message. Instead, you'll rise to discover a dependency's `make config` menu that's been patiently awaiting your attention since 30 seconds after you walked away.

The point of building software from ports is that you can customize it. For these big builds, though, you really want to do all the customizations up front. That's where make `config-recursive` comes in.

The make `config-recursive` walks through the tree of required ports and runs `make config` on each and every one of them. You'll spend a few minutes selecting options in each port or just hitting OK on the ones you don't care about. Once you finish the recursive config, though, you can safely run `make install` on the port you actually want and go off to do other things. You'll return to an installed port or a build failure.

Changing a port's build options can add or remove dependencies. If you decide to enable, say, SNMP support in LibreOffice,[4] the port will need the proper SNMP library. The port for that library will need configuring. Re-run `make config-recursive` until none of your decisions change.

The ports system caches all your configuration choices. To remove that cache for a port and all its dependencies, run `make rmconfig-recursive`.

If bandwidth timing is the problem, you can download all the distfiles required for all the dependencies with `make fetch-recursive`. This is useful if you're in a place like Antarctica, where build time and server cooling is unlimited but you have internet only a few hours a day.

Packaged Dependencies

Some software has hundreds of dependencies, and you probably don't want to build all of them. While I might want a custom Emacs build, I probably don't want to build gmake and the latest GNU C compiler from source. The `make missing` command displays missing dependencies. You can use that command to pick and choose what you want to build.

If you don't want to build any dependencies from source but install them all from packages instead, you can feed `make missing` into a `pkg` command.

```
# pkg install -Ay $(make -DBATCH missing)
```

If a package is available, it'll be installed. The only things you'll need to install from ports are those available only from ports.

Port Flavors

Some ports have complicated dependencies. While you can build Ansible with Python 2 or Python 3, an Ansible package that works with Python 2 is very different than one for Python 3. *Flavors* is a mechanism for expressing these possibilities within a single port, and was very recently introduced into the Ports Collection. Flavors are not yet pervasive throughout the ports system, but at the time I write this, they've been implemented for Python, Perl, Qt, and Emacs. You can expect to see them more and more frequently.

To see if a port supports any flavors, go to the port directory and run `make -V FLAVORS`. Here, I see what flavors of the popular Python packaging toolkit Setuptools are available.

```
# cd /usr/ports/devel/py-setuptools
# make -V FLAVORS
py27 py36 py35 py34
```

My current ports tree supports Python 2.7, 3.6, 3.5, and 3.4.

To build Setuptools for a specific Python version, give the flavor on the command line.

```
# make FLAVOR=py34 install clean
```

If you don't specify a flavor, the port gets built with the current default Python. To set the default Python for your system, set `DEFAULT_VERSIONS` in *make.conf*.

Building Packages

You can create a package from an installed port. You can then copy your customized port to other machines and install it.

Before creating the package, create the directory */usr/ports/packages*. The ports system puts built packages in that directory. Without a *packages* directory, the package winds up in the port directory and you wind up with package files scattered all over your filesystem.

Use `make package` to create a package. If you want to package not only the current port but all its dependencies, run `make package-recursive`.

People who need a whole bunch of customized ports should consider setting up their own repositories with poudriere (discussed later this chapter), but one-off package builds are okay if you have special circumstances or you like saving trouble for later.

Uninstalling and Reinstalling Ports

While you can use `pkg remove` to uninstall a port, you can also uninstall a port from the port directory. Running `make deinstall` in the port directory removes the program from the system but leaves the port compiled and ready to reinstall.

After uninstalling a port, the compiled program and source files still live under the *work* subdirectory in the port. Running `make reinstall` reinstalls the compiled program. You can uninstall and reinstall as many times as you like.

Tracking Port Build Status

How does the Ports Collection keep track of what's already been done? If you can run `make extract` and then `make install`, how does FreeBSD know what it has already finished? The Ports Collection uses hidden files (files with a name beginning with a dot), or *cookies*, to track completed steps. See those files by listing all the files in the port's *work* directory:

```
# cd /usr/ports/security/sudo/work
# ls -a
ls -a
--snip--
.PLIST.flattened
.PLIST.mktmp
```

```
.PLIST.objdump
.PLIST.setuid
.PLIST.writable
.build_done.sudo._usr_local
.configure_done.sudo._usr_local
.extract_done.sudo._usr_local
.install_done.sudo._usr_local
.license-catalog.mk
--snip--
```

The file *.configure_done.sudo._usr_local* indicates that the `make configure` step is complete.

On more than one occasion, after multiple `make install/deinstall` cycles, I've had a port refuse to reinstall itself. That's generally caused by the hidden file indicating that the install has finished. Remove that file, and the reinstall can proceed.

Cleaning Up Ports

Ports can take up a lot of disk space. Programs with many dependencies, like GNOME, KDE, and LibreOffice, can take dozens of gigabytes! Much of this resides in the port's *work* directory, where the port puts the source code files and all the intermediate parts of the completed binaries. Once the port is installed, though, you no longer need those files.

Remove the port's working files with `make clean`. This erases the *work* directory of the current port and all dependencies, so be sure you're happy with your new program before doing this. You can also clean a port immediately upon install by running `make install clean`.

You might also want to remove the original distfiles, stored in */usr/ports/distfiles*. The `make distclean` command removes the distfiles for the current port and all dependencies.

To clean the entire ports tree, run `make clean -DNOCLEANDEPENDS` directly under */usr/ports*. The `-DNOCLEANDEPENDS` is optional, but it prevents the default recursive cleaning. Without it, you'll clean some popular ports dozens or hundreds of times. While there are faster ways to remove every *work* directory in the ports tree, this one is directly supported by the FreeBSD Project.

Read-Only Ports Tree

Many people dislike having temporary files and even packages in */usr/ports*.

You can move the various working directories to other parts of the filesystem to keep your */usr/ports* read-only except for updates.

Use the WRKDIRPREFIX option in *make.conf* to build ports in a separate directory. Many people will set this to a location like */usr/obj*.

The PACKAGES option sets a new package directory other than */usr/ports/packages*.

Finally, DISTDIR sets a location to store distfiles other than */usr/ports/distfiles*.

On a related note, it's possible to build ports and packages without being root, provided the permissions on these directories are set so that the builder can write to these directories. Only root can install software, however.

Changing the Install Path

Many environments have standards for how add-on software gets installed. I've been in organizations where */usr/local* is reserved for files specific to that machine and software installs in that directory are forbidden. Instead, software installs must go in */opt* or some other mandated location.

Set an alternate installation location with the LOCALBASE and PREFIX variables. You could do this on the command line, but if you're complying with an organization standard, use *make.conf* instead. Whichever you use, start by building pkg(8) itself.

```
# cd /usr/ports/ports-mgmt/pkg
# make LOCALBASE=/opt PREFIX=/opt install
```

The port installs all of its files under this directory. For example, programs that normally go into */usr/local/bin* end up in */opt/bin*.

Not every port can handle changing LOCALBASE and PREFIX from */usr/local*. Some software has hardcoded dependencies on */usr/local*, while others have undiscovered bugs. If a port chokes on changing the install path, file a PR (see Chapter 24). Consider taking a look at the port to figure out why it choked. Submitting fixes like this is one of the easiest ways to get involved with FreeBSD.

Private Package Repositories

Packages are great, until you need customized versions; then you need ports. Similarly, ports are great until you have dozens of machines that all need customized ports. What's easy to build on one host is difficult to maintain on several and impossible across a large server farm. When you outgrow ports, you need packages. Customized packages, that is.

The FreeBSD project uses poudriere (pronounced *poo-DRE-er*) for building packages. Why poudriere? It's French for *powderkeg*. The successor to the "tinderbox" tool,[5] poudriere is a collection of shell scripts that leverage existing FreeBSD infrastructure, such as jails and tmpfs and the Ports Collection.

Building packages that work across multiple systems is different than building software that works on the local host. Anything managed by human beings accumulates cruft. Once my desktop is more than a few months old, I'm pretty confident that some minor change I've made will make it subtly different than any newly installed system. Maybe I saved a shared library after an upgrade. Perhaps I installed something by hand and forgot about it. Gremlins could have tampered with the linker, I don't know. The important thing is, my host isn't pristinely identical to every other host running what's supposed to be the same operating system. A port built and packaged on this host might include dependencies, libraries, or who knows what that will keep it from working on other hosts.

Poudriere evades this problem by building everything in jails it manages itself. A poudriere can build packages for any supported FreeBSD release older than the host it runs on. You can't, say, build packages for 13.0-RELEASE on a 12.4-RELEASE host, because the kernel lacks the necessary interfaces.

With poudriere, you can build packages on one host and distribute them among all of your servers. While poudriere includes many advanced features, getting a basic repository running isn't hard at all.

Poudriere Resources

Package building takes system resources. You can restrict how many processors poudriere uses during builds, which helps reduce its memory use. While poudriere itself is only a few megabytes, however, the jails and build environments can take up a whole lot of disk space. The official poudriere

docs recommend allocating at least 4GB of disk for each jail and 3GB of disk space for the ports tree. I normally use about 1GB for each using ZFS, but I encourage you to err on the side of following the recommendations.

Poudriere leverages ZFS clones and snapshots to build jails, vastly reducing the needed disk space and, increasingly, performance. You can run poudriere on UFS, but it will use more space and run more slowly.

Of greater concern is the space needed to build the ports. My web servers run only a few dozen pieces of software, and many of these are tiny. Poudriere needs only a few gigabytes of disk to build them. If you're building hundreds or thousands of packages, you need a whole bunch of disk. How much? Well, are you building GnuPG or are you building LibreOffice? To get an estimate, build but don't clean all of your packages using ports, and then see how big */usr/ports* gets.

Each host should use only one package repository. Yes, it's technically possible to build your local packages and install them alongside packages from the official FreeBSD repository. The problem is that packages are interdependent. You could have your host check your repository first and then fall back to the official repository. The official repository updates every few days, however. The time between updates varies with the hardware available in the build cluster, but a few days is a good guess. Are the updates to your poudriere perfectly synchronized with the official repository's slightly irregular updates? Is your ports tree exactly identical to the one used on the ports cluster? Packages are meant to work as an integrated collection, not a pile of stuff from two different collections. Ask any Linux administrator for their horror stories about packages installed from multiple repositories and then commit to building all your own packages. Plan your disk usage accordingly.

Finally, start by building your packages on a host of the same architecture that you intend to install them on. If you're building packages for arm64 systems, use an arm64 host for poudriere. You can build i386 packages on amd64, but amd64 hardware is literally designed to run i386 code. Once you're comfortable with poudriere, you can use the qemu-user-static package to cross-build packages for slow platforms.

Can you add poudriere to an existing production host? Maybe. A few poudriere runs on a test system will provide insight into the resources your environment needs.

Installing and Configuring Poudriere

Poudriere has no build options, so install it from packages.

```
# pkg install poudriere
```

Configure poudriere in */usr/local/etc*. You'll find a directory for configuring specific package builds, *poudriere.d*, but we'll start with the generic configuration file, *poudriere.conf*. Here's where you'll tell poudriere how to behave. While you can customize directories and paths, we'll stick with the defaults.

You must tell poudriere where to download FreeBSD install files from by setting the `FREEBSD_HOST` variable. If you don't have a local install mirror, use the default of `download.freebsd.org`.

```
FREEBSD_HOST=https://download.FreeBSD.org
```

Poudriere includes ZFS-aware features. ZFS isn't necessary for poudriere, of course, but if it's run on ZFS, it will create, clone, and destroy datasets as needed. Running on UFS won't hinder poudriere, but copying files is slower than cloning. If you're using UFS, uncomment the `NO_ZFS=yes` configuration option. That's it.

ZFS users need to specify the ZFS pool poudriere will use. My main operating install might be on the pool *zroot*, but that pool's on a pair of flash SATADOMs that I don't want to abuse too badly. I have a *scratch* pool specifically for churning data. Set `ZPOOL` in *poudriere.conf*.

```
ZPOOL=scratch
```

Before your first poudriere run, create a */usr/local/poudriere* dataset. You'll be happier.

All of poudriere's work files get put under */usr/local/poudriere*. If you're using a separate ZFS pool, the mount points for the datasets on that pool get set to various locations under */usr/local/poudriere*. On UFS, it's a directory like any other.

My examples run on ZFS because I can. Poudriere's output might look slightly different on UFS systems, but the commands you run are identical no matter the underlying filesystem.

We'll look at a few poudriere customizations later, but this will get you started. Now create jails for your packages.

Poudriere Jail Creation

Poudriere can create jails from a whole bunch of different sources. You can download from a few different sources, build from a source tree, and more. Read poudriere(8) for a full list. Here, I'll install three different jails from my three favorite methods: from the internet, from an install image, and from my custom-built */usr/src* and */usr/obj*. All the installation commands use the same general syntax. Some installation methods will add a new option, but everything starts with these.

```
# poudriere jail -c -j jailname -v version
```

The `jail` subcommand tells poudriere to work on a jail. The `-c` flag means create, and `-j` lets you assign a name to the jail. A jail can have any name that doesn't include a period. I name my poudriere jails after the architecture and release, substituting a dash for any dots. This gives me jails like *amd64-12-0*, *amd64-11-4*, and so on. The `-v` flag takes one argument, the FreeBSD version from `uname -r` but without any patch level information. If your hosts are currently running 12.3-RELEASE-p20, just use 12.3-RELEASE. The patch level will change in subsequent poudriere runs—yes, poudriere applies security patches to jails.

Install Jail from Network

The default jail install grabs the FreeBSD software from the download site specified in *poudriere.conf*. FreeBSD's main download site is geographically load balanced, so there's no need to use any other site unless you have your own mirror. Here, I create a jail called *amd64-11-1* for building 11.1 packages:

```
# poudriere jail -c -j amd64-11-1 -v 11.1-RELEASE
[00:00:00] ====>> Creating amd64-11-1 fs... done
[00:00:01] ====>> Using pre-distributed MANIFEST for FreeBSD 11.1-RELEASE amd64
[00:00:01] ====>> Fetching base.txz for FreeBSD 11.1-RELEASE amd64
--snip--
```

Poudriere goes to the website and starts downloading the distribution

files. Once it has all the files locally, it copies */etc/resolv.conf* into the jail and runs `freebsd-update` to get all the latest security patches. The poudriere run ends with:

```
[00:04:21] ====>> Recording filesystem state for clean... done
[00:04:21] ====>> Jail amd64-11-1 11.1-RELEASE-p1 amd64 is ready to be used
```

You can now configure this jail.

Install Jail from Media

Downloading from the internet is fine, but what if you have the install media locally? Why redownload what you already have sitting on an ISO or a memory stick image? Extract those distribution files onto your local hard drive and you can use them for as many jails as you need. For an ISO, use tar(1).

```
# tar -xf ../FreeBSD-11.0-RELEASE-amd64-disc1.iso usr/freebsd-dist
```

A memory stick image is slightly more complicated; sadly, libarchive can't open disk images yet. You must attach the image to a memory device and mount it.

```
# mdconfig -at vnode -f FreeBSD-11.0-RELEASE-amd64-memstick.img
md0
```

If you try to mount */dev/md0*, you'll get an error. It's not a filesystem; it a partitioned disk image. Identify the partitions on the disk.

```
# gpart show md0
=>      3  1433741  md0  GPT  (700M)
        3     1600    1  efi  (800K)
     1603      125    2  freebsd-boot  (63K)
     1728  1429968    3  freebsd-ufs  (698M)
  1431696     2048    4  freebsd-swap  (1.0M)
```

Partition 3 is a UFS filesystem. That looks promising.[6] Mount it.

```
# mount /dev/md0p3 /mnt
```

The distribution files are now available in */mnt/usr/freebsd-dist*. I could copy them out or just install from their current location.

Here, I create a jail for building FreeBSD 11.0 packages. It'll be called

amd64-11-0 and use the files from the mounted memory stick. Use the `-m` flag to specify where poudriere should grab the files from.

```
# poudriere jail -c -j amd64-11-0 -v 11.0-RELEASE -m url=file:///mnt/usr/freebsd-dist/
```

Note that the argument to `-m` is a URL. I could specify a website here, but `file://` is a perfectly valid type of URL. On a Unix host, a `file://` URL has a third slash to indicate the filesystem root.

Install Jail from a Local Build

I run -current and regularly build from source. I want to build packages for my custom build, so the jail needs a version of FreeBSD that matches my host. The easy way to get that is to install from the same */usr/src* you built the host from. (You could also use Subversion to download a fresh copy of the source code you used to build this system, but that requires understanding Subversion.) Use `-m` to give the location to a source directory.

```
# poudriere jail -c -j amd64-current -v 12.0-CURRENT -m src=/usr/src

[00:00:00] ====>> Copying /usr/src to /usr/local/poudriere/jails/amd64-
current/usr/src...
--snip--
```

Poudriere runs `make installworld` on the prebuilt world in */usr/obj* to create your jail. It won't run `freebsd-update` because -current doesn't support it.

We'll use the *amd64-current* jail in all future examples.

Viewing Jails

To see all the jails poudriere has set up, run `poudriere jail -l`. The output is very wide, so I can't reproduce it in this book, but you'll see the jail's name, the installed version of FreeBSD, the hardware architecture, the method used to install, the timestamp of the installation, and the path to the jail.

Install a Poudriere Ports Tree

Poudriere can use different ports trees for different builds. You might use a quarterly ports branch for one host, the current ports tree for another, and last year's ports tree for a third. (You need to use Subversion to extract particular ports trees from the FreeBSD mirrors, so we won't cover them.)

The possibility of supporting multiple ports trees means you must assign a name to each ports tree you do install. Multiple jails can share a ports tree. The default is the current ports tree.

Use the `poudriere ports` subcommand for all ports-related actions. The `-c` flag tells poudriere to create a ports tree, and the `-p` flag lets you assign the name.

```
# poudriere ports -cp head
[00:00:00] ====>> Creating head fs... done
[00:00:00] ====>> Extracting portstree "head"...
Looking up portsnap.FreeBSD.org mirrors... 6 mirrors found.
--snip--
```

Poudriere leverages portsnap(8), which we discussed earlier this chapter.

If you install multiple ports trees, view them with `poudriere ports -l`.

Configuring Poudriere Ports

The whole point of building a port is to customize it. You don't need to build the whole ports tree as packages, though—not unless you're running the FreeBSD package building cluster or something analogous! You have to tell poudriere which ports to build. Once you have that list, you might need specific options for certain ports, but you might also need global options. You'd normally use */etc/make.conf* to set those options, but you don't want poudriere to use the system's settings. Poudriere needs an isolated *make.conf*. Similarly, you might use `make config` to set up a port, but how can you do that in poudriere?

The Package List

Start by defining the list of packages you want poudriere to build. This list usually goes in the file */usr/local/etc/poudriere.d/pkglist*, although you can put it anywhere you want. Specify each port by its category and directory. To build poudriere itself, use an entry like this:

```
ports-mgmt/poudriere
```

The difficult part here is establishing a base package set. You have to build all the packages the host needs. A host might need dozens or hundreds of packages. Do you really need all of those packages? How did all of those

packages get on this system anyway?

Remember, you probably didn't choose to install all of those packages. You installed an application like Emacs or Apache or LibreOffice, and that application dragged in all those dependencies. You care only about those dependencies as they affect the software you want. If LibreOffice loses a dependency, you don't want poudriere to build that dependency anymore. Poudriere automatically builds and packages dependencies. All you need to specify is the application you want, and let poudriere do the rest.

Use pkg-query(8) to get a list of all the nonautomatically installed software on one of your production systems.

```
# pkg query -e '%a=0' %o
www/apache24
shells/bash
sysutils/beadm
--snip--
```

Use this as a base for your package list. Review it for unneeded stuff. Get a similar list from your other production hosts. Use them to assemble your repository's package list.

Poudriere make.conf

Poudriere assembles a unique *make.conf* for each jail from files in */usr/local/etc/poudriere.d/*. The file */usr/local/etc/poudriere.d/make.conf* contains the global *make.conf* options that you want set for all of your jails. Other *make.conf* files can override those settings, as discussed in poudriere(8), but we'll focus on per-jail *make.conf* files.

Suppose I want LDAP everywhere across my enterprise. Poudriere's */usr/local/etc/poudriere.d/make.conf* would contain:

```
OPTIONS_SET=LDAP
```

Hosts running my custom FreeBSD build all use LibreSSL, though. I would create a separate *make.conf* just for that jail, named *amd64-current-make.conf*. It would contain the LibreSSL configuration.

```
DEFAULT_VERSIONS += ssl=libressl
```

More specific files override general files. Settings in the per-jail files

override poudriere's global *make.conf*. I could turn off LDAP on this one jail even as I enable LibreSSL.

Running make config

Use `poudriere options` to run `make config` for your jail. Each combination of jail and ports tree can have its own unique port options, so you need to specify them on the command line. You must specify the jail with `-j`, the name of the ports tree with `-p`, and the package file with `-f`.

```
# poudriere options -j amd64-current -p head -f pkglist
```

Poudriere figures out which ports actually get built and all their dependencies. It runs you through `make config` for every one of them.

Take note of the options you select; should some of those go into the global or per-jail *make.conf*? Setting them as defaults can save you trouble in future poudriere runs.

You can now build your package repository.

Running Poudriere

The `poudriere bulk` subcommand builds packages in bulk. Use **`-j`** to specify the jail, **`-p`** to give the ports tree name, and **`-f`** to specify the package list file. (Yes, those are the same flags as configuring a port; it's like the poudriere designers wanted to be consistent or something.)

```
# poudriere bulk -j amd64-current -p head -f pkglist
```

Poudriere fires up the jail, mounts all the ports, copies the various configuration files into the jail, decides what order to build stuff in, and starts building. You'll see the name of each port as it starts building.

Some of those port builds might run quite a while. Hit CTRL-T to get the current status, or check the logs to see the current status.

At the end of the build, you'll see the list of any ports that get built and a list of ports that failed to build. Here are the results from an itty-bitty pkglist:

```
[00:04:56] ====>> Built ports: ports-mgmt/pkg devel/pkgconf security/libressl
[00:04:56] ====>> Failed ports: www/obhttpd:build
```

The ports `pkg`, `pkgconf`, and `libressl` built fine. They might not run, but the ports collection could build and package them. The `obhttpd` package did not build, however. If this package is critical, I'll want to fix this problem before letting my clients use this repository.

Let's look at the problems first and then examine the repository.

Problem Ports

After the list of ports that gets built, you'll see a message pointing out where to find the logs.

```
[00:04:56] ====>> Logs: /usr/local/poudriere/data/logs/bulk/amd64-current-head/2018-
10-10_15h05m43s
```

The logs go in a directory named after the jail and the ports tree, with a subdirectory by date. If you don't want to type out the date, there's a convenient *latest* that takes you straight to the most recent log directory.

```
# cd /usr/local/poudriere/data/logs/bulk/amd64-current-head/latest
```

You won't find only logs here; you'll find a website. If you configure your web server to serve up */usr/local/poudriere/data*, you can use a web browser to check poudriere builds (as well as to serve repositories to clients). The *logs* subdirectory here contains poudriere's build logs for every port. If you don't want to sort through those, the *logs/errors* subdirectory contains only the logs for the failed builds.

Now you need to do something terribly radical: read the error log. Perhaps poudriere couldn't fetch the distfile. Maybe the host ran out of disk space. Perhaps something truly weird happened. Or, maybe, the port is actually broken with the build options you chose. Not all ports are built with all options all the time; it's very easy for a port maintainer to miss that a rarely used function is busted. Remember, though, that poudriere is FreeBSD's official port-building mechanism. If a port fails to build under poudriere, it's busted and you should consider filing a bug (see Chapter 24).

Package Repository

Find your completed packages under */usr/local/poudriere/data/packages*. Each combination of jail and ports tree gets its own subdirectory. I build this set of packages on the jail amd64-current using the ports tree head, so my new

repository is in */usr/local/poudriere/data/packages/amd64-current-head*. You'll find the catalogs as the various *.txz* files and the *Latest* subdirectory for the most recent packages.

Congratulations. You have a private package repository. Now to get your clients to use it.

Using the Private Repository

The easiest way to use a private repository is on the poudriere host itself. Local repository configurations for pkg(8) go in */usr/local/etc/pkg/repos*. That directory doesn't exist by default, so create it.

```
# mkdir -p /usr/local/etc/pkg/repos
```

Create a *FreeBSD.conf* file therein. Local repository configurations augment or override the system defaults—that's built into UCL. We need to add one setting to the default repository configuration in */etc/pkg/FreeBSD.conf*.

```
FreeBSD: {
        enabled: no
}
```

This leaves the file */etc/pkg/FreeBSD.conf* untouched but sets `enabled` to `no` for the repository named *FreeBSD*. The default repository is no more.

Now create a separate configuration file for our custom repository. I'm naming this repository *amd64-current*, after the jail.

```
amd64-current: {
        url: "file:///usr/local/poudriere/data/packages/amd64-current-head",
        enabled: yes,
}
```

Your host is now ready to use those packages. You'll want to forcibly reinstall all the current packages to stop using the FreeBSD repository's versions and use your local versions.

```
# pkg install -fy
```

The pkg(8) program will download the repository catalog, but the download will look a little different than usual.

```
--snip--
Updating amd64-current repository catalogue...
Fetching meta.txz: 100%    260 B   0.3kB/s    00:01
Fetching packagesite.txz: 100%   17 KiB  17.4kB/s    00:01
Processing entries: 100%
amd64-current repository update completed. 62 packages processed.
--snip--
```

Compared to the official repository catalog, this catalog is pretty tiny. It extracts the catalog and metadata in one second. The last line shows that this repository has only 62 packages. You're using the new repository. Install your custom packages!

Remote Custom Repositories

The whole point of a package repository is that you build packages once and deploy them everywhere. You could use a read-only NFS export to provide packages to your local machines, but the internet loves to abuse publicly accessible NFS servers. The *pkg.conf* file defines the repository location with a URL. While I used a file for the URL, there's no reason this repository can't use a website instead. Install a web server on your package builder, and have it offer the contents of */usr/local/poudriere/data/packages* to your other servers. Then give the other hosts that should use that repo their own repository configuration.

```
amd64-current: {
        url: "https://pkg.mwl.io/amd64-current-head",
        enabled: yes,
}
```

All our machines now get an identical set of customized ports. This change gets my flunky Bert out of building ports on a dozen machines and into polishing my car.

All Poudrieres, Large and Small

Poudriere performs pretty well by default but has a couple options that can help on small and large systems.

Small Systems

If you have a resource-constrained host, you don't want to let poudriere run amok. Here's a couple *poudriere.conf* options to restrain it.

Generally speaking, if you can build a port on a host, poudriere can build that port. What you don't want is for multiple simultaneous poudriere runs to overwhelm the host. Poudriere normally runs the same number of simultaneous processes as the number of processors in the host. Use the `PARALLEL_JOBS` option to limit the number of parallel builds.

```
PARALLEL_JOBS=1
```

Other restrictions, like reducing the amount of memory a poudriere build can use, are less useful than you might think. A piece of software takes as much memory to build as it requires. Building LibreOffice with only 1GB of RAM will not end well.

Remember that you can also globally deprioritize poudriere runs with nice(1), as discussed in Chapter 21.

Large Systems

Poudriere can take advantage of beefy systems to accelerate builds. You can't speed up the disk, but you can take advantage of memory to use tmpfs(5) for critical parts of the build. Set the `USE_TMPFS` option to use memory for the working directory.

```
USE_TMPFS=yes
```

You can use tmpfs(5) for parts of the build beyond the working directory, but few of us have *that* much memory. Read the *poudriere.conf.sample* for details.

If you build many package repositories, investigate poudriere's cache (*https://ccache.samba.org/*) support. You'll use about 5GB of disk space per jail but save a whole bunch of time rebuilding packages.

Updating Poudriere

New ports get added all the time, with new options. Other software projects continually release new versions, and the FreeBSD port is correspondingly

updated. You'll want those new versions on your servers. If you build your ports with poudriere, updating is pretty simple. You'll need to update your jail and your ports tree. Before updating either, though, make sure *poudriere.conf* is set up to handle updates.

/USR/PORTS/UPDATING

Before updating your ports, check */usr/ports/UPDATING* for any special notes that might affect your environment. An unexpected change in the default version of Python or Perl can ruin your whole day.

Poudriere has two options for handling dependency changes. You'll want to enable both. `CHECK_CHANGED_DEPS` tells poudriere not to trust earlier dependency calculations and perform those checks again. This catches changes in underlying Perl, Python, and so on. Similarly, `CHECK_CHANGED_OPTIONS` tells poudriere to verify each port's options. Setting this to `verbose` tells poudriere to show you any changes.

```
CHECK_CHANGED_OPTIONS=verbose
CHECK_CHANGED_DEPS=yes
```

Now you can update your jails and the ports tree. Use the `-u` flag to update the jail. Give the jail name with `-j`. Here, I update poudriere's amd-11-1 jail.

```
# poudriere jail -j amd64-11-1 -u
```

For jails installed from official media, poudriere runs freebsd-update(8) and applies any missing security patches. If you installed from source, poudriere repeats the install process.

Similarly, update the ports tree with `-u`. Specify the name of the ports tree with `-p`.

```
# poudriere ports -p head -u
```

You'll see poudriere use portsnap(8) to grab the latest updates. Now you

can build the new version of the package repository, exactly as you did the first time.

```
# poudriere bulk -j amd64-current -p head -f pkglist
```

Poudriere determines what needs updating and what must be rebuilt and proceeds accordingly. Once the build is complete, your clients can upgrade their packages from the repository.

More Poudriere

Poudriere has many more features than what I cover here. You can cryptographically sign your packages with the `PKG_REPO_SIGNING_KEY` variable. Package sets let you define different build options for different repositories. You want to build an experimental package run with the latest Python? Look at package sets. You can blacklist ports so that they're never built, even if called as a dependency. See poudriere(8) for all kinds of nifty stuff.

Between ports and poudriere, you can now customize software any way you need. If you really want to get into the nitty-gritty of the Ports Collection, check out the FreeBSD Porter's Handbook on *https://www.freebsd.org/*. The rest of us will move on to some of FreeBSD's advanced software features.

17
ADVANCED SOFTWARE MANAGEMENT

FreeBSD offers unique features that help system administrators better meet users' needs. Knowing how the system really works helps you make better decisions. For example, while multiple processors, multicore processors, and hardware threads can all increase system performance, they don't always help as much as you might think. Knowing how different types of multiprocessing affect different types of workloads tells you where you can improve performance and where you can't.

For your programs to start at boot and stop cleanly at shutdown, you must be able to create and edit proper startup and shutdown scripts. While some programs stop nicely when you just kill the operating system under them, others (for example, databases) demand a gentler shutdown. Starting and stopping system services cleanly is an excellent habit to develop, so we'll learn more about the FreeBSD startup and shutdown scripts.

Under normal circumstances, you'll never need to know how FreeBSD's linking and shared library support works, but we'll discuss them anyway. Why? Because normal circumstances are, oddly, quite rare in the computer business.

Finally, FreeBSD can run Linux software with the Linux compatibility layer, as well as software written for other hardware architectures.

Using Multiple Processors: SMP

If you have a desktop or server built in the last 10 years, it almost certainly

has multiple processors. Some of those processors are for dedicated purposes, such as the graphics processor in your video card. Modern operating systems use *symmetric multiprocessing (SMP)*, or multiple identical general-purpose processors. Modern hardware includes many different dedicated-purpose processors, such as the graphics card and server remote management and so on, but the hardware presented to the operating system has identical processors.

SMP systems have many advantages over single processors, and it's not just the obvious "more power!" If you think about it at the microscopic level, in very small timeframes, a CPU can do only one thing at a time. Every process on the computer competes for processor time. If the CPU is performing a database query, it isn't accepting the packet that the Ethernet card is trying to deliver. Every fraction of a second kernel directs the CPU to perform a *context switch* and work on another request. This happens so often and so quickly that the computer appears to be doing many things at once—much as a television picture appears to be moving when it's really just showing individual pictures one after the other very quickly.

My desktop has cwm providing window management, Firefox with eighty bajillion tabs, and LibreOffice accepting my typing. There's a whole bunch of terminal windows attached to SSH sessions. Network interrupts are arriving; the screen is displaying text; the MP3 player is streaming Nurse With Wound to Stereohenge. The computer's "seamless multitasking" only appears seamless to my feeble brain. In reality, the computer merely switches from one task to another very quickly. One millisecond, it's sending another sliver of sound to my headphones, and the next, it's updating text on the screen.

With multiple processors, your computer really *can* perform multiple operations simultaneously. This is very useful—but system complexity skyrockets.

Kernel Assumptions

To understand SMP and the problems associated with it, we must delve into the kernel. All operating systems face the same challenges when supporting SMP, and the theory here is applicable across a wide variety of platforms. What follows is a gross simplification. Kernel design is a tricky subject, and it's almost impossible for any description to do it justice. Nevertheless,

here's a rough stab at it.

FreeBSD divides CPU utilization into time slices. A *time slice* is the length of time one CPU spends doing one task. One process can use the CPU either for a full-time slice or until there's no more work for it to do, at which point the next task may run. The kernel uses a priority-based system to allocate time slices and to determine which programs may run in which time slices. If a process is running, but a higher-priority process presents itself, the kernel allows the first process to be interrupted, or *preempted*. This is commonly referred to as *preemptive multitasking*.

Although the kernel is running, it isn't a process. Any process has certain data structures set up by the kernel, and the kernel manipulates those data structures as it sees fit. You can consider the kernel a special sort of process, one that behaves very differently from all other processes. It can't be interrupted by other programs—you can't type `pkill kernel` and reboot the system. Way back in the day, the kernel might have been called the *control process* or *monitor*.

The kernel has special problems, not faced by other parts of the system. Imagine that you have a program sending data over the network. The kernel accepts data from the program and places it in a chunk of memory to be handed to the network card. If the computer can do only one thing at a time, nothing happens to that piece of memory or that network card until the kernel gets back to that task. If you have multiple processors, however, the computer can perform multiple tasks simultaneously. What if two different CPUs, both working on kernel tasks, direct your network card to perform different actions at the same time? The network card behaves much as you do when you have your boss screaming in one ear and your spouse in the other; nothing you do can satisfy either of them. What if one CPU allocates memory for a network task, while the other CPU allocates that same memory for a filesystem task? The kernel becomes confused, and the results will not please you.

Unix-like kernels designed for a single processor declare that the kernel is nonpreemptive and can't be interrupted. This simplifies kernel management because everything becomes completely deterministic: when a part of the kernel allocates memory, it can count on that memory being unchanged when it executes the next instruction. No other part of the kernel will alter that chunk of memory. When the computer could do only one thing at a

time, this was a safe assumption. Start doing many things at once, however, and this assumption blows apart.

SMP: The First Try

The first implementation of SMP support in FreeBSD was very simple minded. Processes were scattered between the CPUs, achieving a rough balance, and there was a lock on the kernel. Before a CPU would try to run the kernel, it would check to see whether the lock was available. If the lock was free, the CPU held the lock and ran the kernel. If the lock wasn't free, the CPU knew that the kernel was being run elsewhere and went on to handle something else. This lock was called the *Big Giant Lock (BGL)*, or later just *Giant*. Under this system, the kernel could know that data wouldn't change from under it. Essentially, Giant guaranteed that the kernel would run on only one CPU, just as it always had.

This strategy worked kind of adequately for two CPUs. You could run a medium-level database and a web server on a twin-CPU machine and feel confident that the CPU wouldn't be your bottleneck. If one CPU was busy serving web pages, the other would be free to answer database queries. But if you had an eight-CPU machine, you were in trouble; the system would spend a lot of time just waiting for Giant to become available!

This simplistic SMP technique is neither efficient nor scalable. The standard textbooks on SMP rarely mention this method because it's so clunky. Some other SMP-handling methods are worse, however. For example, several early versions of Microsoft's server OS dedicated one processor to the user interface and the other to everything else. This technique also rarely appears in the textbooks, although it does help your mouse appear more responsive.

Today's SMP

Once you have a lock on the kernel, though, you can divvy up that lock. FreeBSD has fragmented Giant into many smaller locks, and now every part of the kernel uses the smallest possible lock to perform its tasks. Initially, the locks were implemented on core kernel infrastructure, such as the scheduler (the part of the kernel that says which tasks may have which time slices), the network stack, the disk I/O stack, and so on. This immediately improved

performance because while one CPU was scheduling tasks, the other could be processing network traffic. Then, locks were pushed lower into the various kernel components. Each part of the network stack developed its own lock, then each part of the I/O subsystem, and so on—allowing the kernel to use multiple processors to do multiple things simultaneously. These separate kernel subprocesses are called *threads*.

Each type of locking has its own requirements. You'll see references to many different locks such as mutexes, sx locks, rw locks, spin mutexes, semaphores, read-mostly locks, and more. Each has its own benefits and drawbacks, and each must be carefully applied within the kernel.

Fine-grained locking is *a lot* harder than it sounds. Lock too finely, and the kernel spends more time processing locks than pushing data. Lock too coarsely, and the system wastes time waiting for locks to become available. Locking sufficient for a 2-processor system stalls and chokes a 32-processor system, and what works for the 32-core host is totally inadequate for the new 192-core systems. Lock adjustment and tuning has taken years, is still ongoing, and will continue forever.

While every part of the kernel uses the smallest lock currently possible, sometimes that lock is the Giant lock. Unplugging a USB device means grabbing Giant for a fraction of a second as the kernel says, "Hold everything! I'm reconfiguring the hardware!" A few device drivers still use Giant, as do certain tricky parts of the virtual memory stack, the sysctl handlers, and so on.

SMP Problems: Deadlocks and Lock Order Reversals

All of these kernel locks have complicated rules for their use, and they interact with each other in myriad ways. The rules protect against unexpected lock interactions. Suppose that kernel thread A needs resources Y and Z, kernel thread B also needs Y and Z, but B needs Z before it needs Y. If A locks Y while B locks Z, then A winds up waiting for Z while B waits for Y. Neither thread can proceed until the missing resource is freed. This *deadlock* (also called a *deadly embrace*) will destabilize the system, probably bringing it down. Proper locking avoids this problem, among others.

You might see a console message warning of a *lock order reversal*, meaning that locks have been applied out of order. While this kernel notice isn't always an omen of impending doom, it's important to pay attention.

The `WITNESS` kernel option specifically watches for locking order and lock ordering violations. This option is enabled by default on FreeBSD-current (see Chapter 18), and if you report a problem with your system, the development team might ask you to enable it. The `WITNESS` option makes the kernel inspect every action it takes for locking order violations, which reduces system performance. You can enable and disable `WITNESS` with the `debug.witness.watch` sysctl. Running `WITNESS`, reading the messages, and acting on them is an excellent way to help improve FreeBSD, however.

Handling Lock Order Reversals

When you get one of these lock order reversal (LOR) messages, copy the LOR message in its entirety. In addition to appearing on the console, such messages are logged to */var/log/messages* for your convenience. Once you have the lock order message, search the FreeBSD-current mailing list for the first few lines of your LOR message to see whether someone has already filed it. If you find your LOR on the mailing lists, read the message and take the recommended action. There's no need to post a "me too" message on the mailing list unless a developer recently and specifically requested notification of further LORs of that type.

If you have a new LOR, congratulations! Discovering a new LOR isn't as satisfying as discovering a new insect species—you don't get to name your LOR, for one thing—but it does help the FreeBSD Project. Email your report to the FreeBSD-current mailing list. Provide full details on your system, especially the work being performed at the time the LOR appeared. You might be asked to file a bug report, as discussed in Chapter 24.

Processors and SMP

You'll see three different types of multiprocessor systems: multiple cores, multiple packages, and hardware threads. You need to understand the differences among them, as the different processor types have a direct impact on system and application behavior.

The basic unit in processors is the *CPU core*. Each CPU core consists of a set of resources like execution units, registers, cache, and so on. Once upon a time, a core was the same thing as a processor.

A CPU *package* is the chip socketed or soldered to your mainboard. It's

what many people think of as "a CPU" or "a processor." That expensive part you can accidentally crush underfoot? That's a package. Each package contains one or more cores. Prior to SMP, one package had only one core in it. These days, most packages have at least two cores, and the upper number keeps increasing. CPU cores within the same package can communicate with each other relatively quickly.

Some hosts have more than one package. Multiple packages give you multiple groups of multiple cores, giving you the chance for even more parallelism. Communication between packages is slower than communication between cores on the same package. Also, each package usually has its own memory controller. CPU cores in one package will take longer to retrieve data in memory attached to a different package. Yes, this means a 16-core package will perform better than two 8-core packages. In reality, though, very little software is so heavily threaded that it can take advantage of the difference.

Lastly some CPU cores can try to make more efficient use of their execution resources by being able to run more than one thread at a time. This is referred to as *hardware threading*, *Simultaneous Multi-Threading (SMT)* or (if you're Intel) *HyperThreading*. The additional threads are sometimes called virtual processors or virtual cores. The virtual processor isn't a full-fledged CPU, however; for example, it's available only when the first CPU is waiting for something. FreeBSD's default scheduler, sched_ule(4), is aware of which cores are real and which are virtual, and schedules work appropriately.

Hardware threading presents a variety of potential security problems. A task running on one virtual processor can capture data such as cryptographic keys from a task running on another virtual processor using a variety of subtle timing attacks. It's not a script-kiddie-friendly attack, but if you don't trust your users, you can disable hardware threads by setting the boot-time tunable `machdep.hyperthreading_allowed` to `0`.

Using SMP

Remember that multiple processors don't necessarily make the system faster. One processor can handle a certain number of operations per second. A second processor just means that the computer can handle twice as many operations per second, but those operations aren't necessarily any faster.

Think of the number of CPUs as the lanes on a road. If you have one lane, you can move one car at a time past any one spot. If you have four lanes, you can move four cars past that spot. Although the four-lane road won't allow those cars to reach their destination more quickly, there'll be a lot more of them arriving at any one time. If you think this doesn't make a difference, contemplate what would happen if someone replaced your local freeway with a one-lane road. CPU bandwidth is important.

While one CPU can do only one thing at a time, one process can run on only one CPU at a time. Many programs can't perform work on multiple processors simultaneously. Threaded programs are an exception, as we'll see later in this chapter. Some programs work around this limitation by simultaneously running multiple processes and letting the operating system scatter them between processors as needed. The popular Apache web server has done this for many years. Threaded programs are specifically designed to work with multiple processors without spawning multiple processes. Many threaded programs simply create a whole bunch of threads to process data and scatter those threads across CPUs, which is a simple, if not always effective, way to handle parallelism. Other programs don't handle multiple CPUs at all.

If you find that one of your CPUs is 100 percent busy while the others are mostly idle, you're running a program that doesn't handle multiple CPUs in any way. Chapter 21 dives into performance issues, but not much can be done to help such a program.

SMP and make(1)

The make(1) program, which is used to build software, can start multiple processes. If your program is cleanly written, you can use multiple processes to build it. This doesn't help for small programs, but when you're building a large program, such as FreeBSD itself (see Chapter 18) or LibreOffice, using multiple processors can really accelerate the work. Use make(1)'s `-j` flag to tell the system how many processes to start simultaneously. A good choice is the number of processors or cores in the system plus one. For example, on a dual-processor system with two cores on each processor, I would run five processes to build a program.

```
# make -j5 all install clean
```

Some programmers don't design their *Makefiles* correctly, so their programs can't handle being built with the `-j` flag. If a build gives you trouble, stop using `-j` and try again—or, better still, figure out the problem and file a bug report with the author.

Threads, Threads, and More Threads

One word you'll hear in various contexts is *thread*. Some CPUs support HyperThreading. Some processes have threads. Some parts of the kernel run as threads. My pants have many many threads (although some have fewer than my wife thinks necessary for those pants to be worn in public). What are all these threads, and what do they mean?

In most contexts, a thread is a lightweight process. Remember, a *process* is a task on the system, a running program. Processes have their own process ID in the system and can be started, stopped, and generally managed by the user. Threads are pieces of a process, but they're managed by the process and can't be directly addressed by the user. A process can do only one thing at a time, but individual threads can act independently. If you have a multiprocessor system, one process can have threads running on multiple processors simultaneously.

Any threaded program needs to use a *threading library* that tells the application how to use threads on that operating system by interacting with the kernel. Threading libraries implement threading in different ways, so using particular libraries can impact application performance.

Similarly, a kernel thread is a subprocess within the kernel. FreeBSD has kernel threads that handle I/O, network, and so on. Each thread has its own functions, tasks, and locking. The threading within the kernel doesn't use any userland libraries.

Hardware threads are virtual CPU cores, as discussed in “"Using Multiple Processors: SMP" on page 396. While you need to understand what hardware is and how it impacts your system, hardware threads aren't really part of threading.

Startup and Shutdown Scripts

The service(8) command is a frontend to the system startup and shutdown scripts. These scripts are known as *rc scripts* after */etc/rc*, the script that manages the multiuser boot and shutdown process. While the main rc scripts are in */etc/rc.d*, scripts in other locations manage add-on software. Ports and packages install startup scripts, but if you install your own software, you'll need to create your own rc script. If you've never used shell scripts before, read carefully. Shell scripting isn't hard, and the best way to learn is by reading examples and making your own variations on those examples. Additionally, changing an existing package's startup or shutdown process requires understanding how the startup scripts function.

During boot and shutdown, FreeBSD checks */usr/local/etc/rc.d* for additional shell scripts to be integrated into the startup/shutdown process. (You can define additional directories with the `local_startup` *rc.conf* variable, but for now we'll assume that you have only the default directory.) The startup process specifically looks for executable shell scripts and assumes that any script it finds is a startup script. It executes that script with an argument of `start`. During shutdown, FreeBSD runs those same commands with an argument of `stop`. The scripts are expected to read those arguments and take appropriate actions.

rc Script Ordering

For decades, Unix-like operating system encoded service startup order in the startup scripts. That got really annoying, *really* quickly. Many, but not all, Unixes have moved on from this. Similarly, FreeBSD's rc scripts arrange themselves in order. Each rc script identifies what resources it needs before it can start. The rc system uses that information to sort the scripts into order. This is performed by rcorder(8) at boot and at shutdown, but you can do this by hand at any time to see how it works. Just give rcorder(8) the paths to your startup scripts as arguments.

```
# rcorder /etc/rc.d/* /usr/local/etc/rc.d/*
/etc/rc.d/growfs
/etc/rc.d/sysctl
/etc/rc.d/hostid
/etc/rc.d/zvol
/etc/rc.d/dumpon
/etc/rc.d/ddb
/etc/rc.d/geli
/etc/rc.d/gbde
```

```
--snip--
```

The rcorder(8) program sorts all the scripts in */etc/rc.d* and */usr/local/etc/rc.d* into the order used at system boot, using markers within the scripts themselves. If your rc scripts have any ordering errors, such as deadlocked scripts, those errors appear at the beginning of your rcorder(8) output.

A Typical rc Script

The rc script system is pretty simple—while scripts can get complicated, the complexity comes from the program the script runs, not the rc system. The script that starts the NFS server has a whole bunch of dependencies and requirements. The script for a simpler daemon, like timed(8), illuminates the rc system.

```
   #!/bin/sh

❶ # PROVIDE: timed
❷ # REQUIRE: DAEMON
❸ # BEFORE:  LOGIN
❹ # KEYWORD: nojail shutdown

❺ . /etc/rc.subr

❻ name="timed"
❼ desc="Time server daemon"
❽ rcvar="timed_enable"
❾ command="/usr/sbin/${name}"

❿ load_rc_config $name
   run_rc_command "$1"
```

The PROVIDE label ❶ tells rcorder(8) the official name of this script. This script is called *timed*, after timed(8).

The REQUIRE label ❷ lists other scripts that must run before this script runs. Scripts that need timed to run before they can start list timed in REQUIRE. This script can run any time after the DAEMON script has been run.

The BEFORE label ❸ lets you specify scripts that should run after this one. This script should run before the LOGIN script. Both */etc/rc.d/LOGIN* and

/etc/rc.d/timed specify that they have to run after `DAEMON`, but the `BEFORE` label lets you set additional ordering requirements.

The `KEYWORD` command ❹ lets the startup system select only certain startup scripts. The timed(8) script includes `nojail` and `shutdown`. Jails don't run this script, even if enabled. This script gets run at system shutdown.

The */etc/rc.subr* file ❺ contains the rc script infrastructure. Every rc script must include it.

While the script has a name, the program run by the script might have a separate name ❻. Most often, though, an rc script officially called *timed* will run the program *timed*.

The `description` field ❼ provides a brief description of the service the script provides, exactly as you'd expect.

The `rcvar` statement ❽ lists the *rc.conf* variable that toggles this script.

The `command` ❾ identifies exactly which command this script should run—after all, you might have multiple commands of the same name on your system, just in different directories.

The last two actions the script takes are to load ❿ the configuration for this service from */etc/rc.conf* and then actually run the command.

While this might look intimidating, it's not really that hard in practice. Start your customized rc script by copying an existing one. Set the command name to that of your command and change the path appropriately. Decide what the script must have run before it: Do you need the network to be running? Do you need particular daemons to be started already, or do you need to run your program before certain daemons? If you really don't know, have your script run at the very end by using a `REQUIRE` statement with the name of the last script run on your system. By looking through other rc scripts that provide similar functions, you'll learn how to do almost anything in a startup script.

With this simple script, you can enable, disable, and configure your program by adding information to */etc/rc.conf*. For example, if your custom daemon is named `tracker`, the startup script will look for variables `tracker_enable` and `tracker_flags` in */etc/rc.conf* and use them each and every time you run the startup script.

Special rc Script Providers

You might have noticed the services named *DAEMON* in our example and thought, "That's odd. I don't know of any system processes called *DAEMON*." That's because it's not a process. The rc system has a few special providers that define major points in the boot process. Use these to make writing rc scripts easier.

The FILESYSTEMS provider guarantees you that all local filesystems are mounted as defined in */etc/fstab*.

The NETWORKING provider appears after all network functions are configured. This includes setting IP addresses on network interfaces, PF configuration, and so on.

The SERVERS provider means that the system has enough basic functionality to support basic servers, such as named(8) and the NFS support programs. Remote filesystems aren't mounted yet.

The DAEMON provider ensures all local and remote filesystems are mounted, including NFS and CIFS, and that more advanced network functions, such as DNS, are operational.

At LOGIN, all network system services are running and FreeBSD is beginning to start up services to support logins via the console, FTP daemons, SSH, and so forth.

By using one of these providers in a `REQUIRE` statement in your custom rc script, you can specify roughly when you want your custom program to run without going too far into nitty-gritty details.

Vendor Startup/Shutdown Scripts

Perhaps you're installing a complicated piece of software, and the vendor doesn't support FreeBSD's rc system. This isn't a problem. Most vendor-supplied scripts expect to get a single argument, such as `start` or `stop`. Remember that at boot time, FreeBSD runs each rc script with an argument of `start`, and at system shutdown, it runs the scripts with an argument of `stop`. By adding `PROVIDE` and `REQUIRE` statements as comments to this vendor script and confirming that it accepts those arguments, you can make the script run at the proper time in the startup and shutdown process.

Use of the rc system features in management scripts isn't mandatory. At

the tail end of the boot process, FreeBSD runs */etc/rc.local*. Add your local commands there. You can't use service(8) to manage anything in *rc.local*, however.

Debugging Custom rc Scripts

Local scripts, such as those installed by the Ports Collection, are run by */etc/rc.d/localpkg*. If your custom script is causing problems, you might try running the `localpkg` script with debugging to see how your script is interacting with the rc system. The best way to do this is to use debugging.

```
# /bin/sh -x /etc/rc.d/localpkg start
```

This attempts to start every local daemon on your server again, which might not be desirable on a production system. Try it on a test system first. Also, remember that the `-x` debugging flag isn't passed on to the child scripts; you're debugging the system startup script */etc/rc.d/localpkg* itself, not the local scripts. Run your script with the `-x` flag to debug it.

Managing Shared Libraries

A shared library is a chunk of compiled code that provides common functions to other compiled code. Shared libraries are designed to be reused by as many different programs as possible. For example, many programs must generate *hashes*, or cryptographic checksums, on pieces of data. If every program had to include its own hashing code, programs would be harder to write and more unpleasant to maintain. What's more, programs would have interoperability problems if they implemented hashes slightly differently, and program authors would need to learn an awful lot about hashes to use them. By using a shared library (in this example, `libcrypt`), the program can access hash generation functions without any compatibility and maintenance problems. This reduces the average program size, both on disk and in memory, at a cost in complexity.

Shared Library Versions and Files

Shared libraries have a human-friendly name, a version number, and an

associated file. The human-friendly name is usually (but not always) similar to the associated file. For example, version 1 of the shared library called *libjail* is in the file */lib/libjail.so.1*. On the other hand, version 11 of the main Kerberos library is in the file */usr/lib/libkrb5.so.11*. Version numbering starts at 0.

Historically, when changes to the library made it incompatible with earlier versions of the library, the version number was incremented. For example, libjail.so.0 became libjail.so.1. The FreeBSD team doesn't bump these versions except at the beginning of a release cycle (see Chapter 18). Each library also has a symlink for the library name without a version, pointing to the latest version of the library. For example, you'll find that */usr/lib/libwres.so* is actually a symlink pointing to */usr/lib/libwres.so.10*. This makes compiling software much easier, as the software has to look only for the general library file rather than a specific version of that library.

FreeBSD's main libraries support *symbol versioning*, which lets shared libraries support multiple programming interfaces. With symbol versioning, a shared library provides every program with the version of the library the program requires. If you have a program that requires version 2 of a library, version 3 will support the functions just as well.

Just because FreeBSD supports symbol versioning doesn't mean that all the software in the Ports Collection supports it. You must be alert for library version problems.

Attaching Shared Libraries to Programs

So, how does a program get the shared libraries it needs? FreeBSD uses ldconfig(8) and rtld(1) to provide shared libraries as needed but also offers a few human-friendly tools for you to adjust and manage shared library handling.

The rtld(1) is perhaps the simplest program to understand, at least from a sysadmin's perspective. Whenever a program starts, rtld(8) checks to see what shared libraries the program needs. The rtld(8) program searches the library directories to see whether those libraries are available and then links the libraries with the program so everything works. You can't do very much at all with rtld(1) directly, but it provides the vital glue that holds shared libraries together.

The Library Directory List: ldconfig(8)

Instead of searching the entire hard drive for anything that looks like a shared library every time any dynamically linked program is run, the system maintains a list of shared library directories with ldconfig(8). (Older versions of FreeBSD built a cache of actual libraries on a system, but modern versions just keep a list of directories to check for shared libraries.) If a program can't find shared libraries that you know are on your system, this means ldconfig(8) doesn't know about the directory where those shared libraries live.[1] To see the libraries currently found by ldconfig(8), run `ldconfig -r`.

```
# ldconfig -r
/var/run/ld-elf.so.hints:
        search directories: /lib:/usr/lib:/usr/lib/compat:/usr/local/lib:/usr/
local/lib/perl5/5.24/mach/CORE
        0:-lcxxrt.1 => /lib/libcxxrt.so.1
        1:-lalias.7 => /lib/libalias.so.7
        2:-lrss.1 => /lib/librss.so.1
        3:-lkiconv.4 => /lib/libkiconv.so.4
        4:-lpjdlog.0 => /lib/libpjdlog.so.0
--snip--
```

With the `-r` flag, ldconfig(8) lists every shared library in the shared library directories. We first see the list of directories searched and then the individual libraries in those directories. My main mail server has 170 shared libraries; my main web server, 244; my desktop, 531.

If a program dies at startup with a complaint that it can't find a shared library, that library won't be on this list. Your problem then amounts to installing the desired library into a shared library directory or adding the library directory to the list of directories searched. You could just copy every shared library you need to */usr/lib*, but this makes system management very difficult—much like with a filing cabinet where everything is filed under *P* for *paper*. Adding directories to the shared library list is a better idea in the medium to long term.

Adding Library Directories to the Search List

If you've added a new directory of shared libraries, you must add it to the list ldconfig(8) searches. Check these ldconfig(8) entries in */etc/defaults/rc.conf*:

```
ldconfig_paths="/usr/lib/compat /usr/local/lib /usr/local/lib/compat/pkg"
ldconfig_local_dirs="/usr/local/libdata/ldconfig"
```

The `ldconfig_paths` variable lists common locations for libraries. While out-of-the-box FreeBSD doesn't have the directory */usr/local/lib*, most systems grow one shortly after install. Similarly, libraries for compatibility with older versions of FreeBSD go in */usr/lib/compat*. The location for storing old versions of libraries installed by packages is */usr/local/lib/compat/pkg*. The */lib* and */usr/lib* directories get searched by default, but the paths in this variable are common locations for shared libraries.

Ports and packages use the `ldconfig_local_dirs` variable to get their shared libraries into the search list without just dumping everything into */usr/local/lib*. Packages can install a file in this directory. The file is named after the package and contains a list of directories with the libraries installed by the package. The ldconfig program checks these directories for files, reads the paths in the files, and treats those as additional library paths. For example, the Perl 5 package installs shared libraries in */usr/local/lib/perl5/5.24/mach/CORE*. The port also installs a file called */usr/local/libdata/ldconfig/perl5*, containing only a single line with this path in it. The ldconfig startup script adds the directories in these files to its list of places to check for shared libraries.

/USR/LOCAL/LIB VS. PER-PORT LIBRARY DIRECTORIES

Isn't */usr/local/lib* specifically for libraries installed by ports and packages? Why not just put all your shared libraries into that directory? Most ports do exactly that, but sometimes having a separate directory makes maintenance simpler. For example, I have Python 2.7 installed on my laptop, and */usr/local/lib/python27* includes 647 files! Dumping all those into */usr/local/lib* would overwhelm my non-Python libraries and make it harder for me to find the files installed by ports with only one or two shared libraries.

To get your directory of shared libraries into the search list, either add it to the ldconfig_paths in */etc/rc.conf* or create a file listing your directory in */usr/local/libdata/ldconfig*. Either works. Once you add the directory, the libraries in that directory are immediately available.

ldconfig(8) and Weird Libraries

Shared libraries have a couple of edge cases that you should understand and many more that you really don't have to worry about. These include libraries for different binary types and libraries for other architectures.

FreeBSD supports two different formats of binaries, a.out and ELF. System administrators don't need to know the details of these binary types, but you should know that ELF binaries are the modern standard and became FreeBSD's standard in version 3.0, back in 1998. Older versions of FreeBSD used a.out. Programs compiled as one type can't use shared libraries of the other type. While a.out binaries have largely vanished, the cost of supporting them is so low that this support has never been removed. ldconfig(8) maintains separate directory lists for a.out and ELF binaries, as you can see from the output of */etc/rc.d/ldconfig*. You'll find separate configuration options for ldconfig(8) with a.out libraries in *rc.conf*. It's barely conceivable that you'll need an a.out program.

Another odd case is when you're running 32-bit binaries on a 64-bit FreeBSD install. This is most common when you're running the amd64 install and want to use a program from an older version of FreeBSD. 64-bit binaries cannot use 32-bit libraries, so ldconfig(8) keeps a separate directory list for them. You'll find options to configure those directories in *rc.conf* as well. Don't mix your 32-bit and 64-bit libraries!

A few hardware platforms, such as ARM, have special versions of libraries for soft floating-point operations. You'll find *rc.conf* options for those as well, pointing to a third set of directories.

In short, don't mix unusual libraries with the standard libraries. The results will confuse FreeBSD, which will in turn upset you.

LD_LIBRARY_PATH and LD_PRELOAD

While FreeBSD's built-in shared library configuration system works well if you're the sysadmin, it won't work if you're just a lowly user without root access.[2] Also, if you have your own personal shared libraries, you probably don't want them to be globally available. Sysadmins certainly won't want to take the risk of production programs linking against random user-owned

libraries! Here's where LD_LIBRARY_PATH comes in.

Every time rtld(1) runs, it checks the environment variable LD_LIBRARY_PATH. If this variable has directories in it, it checks these directories for shared libraries. Any libraries in these directories are included as options for the program. You can specify any number of directories in LD_LIBRARY_PATH. For example, if I want to do some testing and use libraries in */home/mwlucas/lib* and */tmp/testlibs* for my next run of a program, I'd just set the variable like this:

```
# setenv LD_LIBRARY_PATH /home/mwlucas/lib:/tmp/testlibs
```

You can set this automatically at login by entering the proper command in *.cshrc* or *.login*.

Similarly, the LD_PRELOAD environment variable lets you load a particular library first. You have test your custom libc by giving the full path to it in LD_PRELOAD. When rtld(1) runs, it takes the library from LD_PRELOAD and ignores later libraries that offer the same symbols.

LD_ ENVIRONMENT VARIABLES AND SECURITY

Using LD_LIBRARY_PATH or LD_PRELOAD is not secure. If you point this variable to an overly accessible directory, your program might link against whatever anyone put in there. The LD_LIBRARY_PATH variable overrides the shared library directory list, so if someone can put arbitrary files in your library directory, they can take over your program. For this reason, setuid and setgid programs ignore these variables.

What a Program Wants

Lastly, there's the question of what libraries a program requires to run correctly. Get this information with ldd(1). For example, to discover what libraries Emacs needs, enter this command:

```
# ldd /usr/local/bin/emacs
/usr/local/bin/emacs:
```

```
        libtiff.so.5 => /usr/local/lib/libtiff.so.5 (0x800a78000)
        libjpeg.so.8 => /usr/local/lib/libjpeg.so.8 (0x800cf1000)
        libpng16.so.16 => /usr/local/lib/libpng16.so.16 (0x800f63000)
        libgif.so.7 => /usr/local/lib/libgif.so.7 (0x80119d000)
        libXpm.so.4 => /usr/local/lib/libXpm.so.4 (0x8013a6000)
        libgtk-3.so.0 => /usr/local/lib/libgtk-3.so.0 (0x801600000)
--snip--
```

This output tells us the names of the shared libraries Emacs requires and the locations of the files that contain those libraries. If your program can't find a necessary library, ldd(1) tells you so. The program itself announces the name of the first missing shared library when you try to run it, but ldd(1) gives you the complete list so that you can use a search engine to find all missing libraries.

Between ldconfig(8) and ldd(1), you should be fully prepared to manage shared libraries on your FreeBSD system.

Remapping Shared Libraries

Occasionally, you'll find a piece of software that you want to run with particular shared libraries not used by the rest of the system. For example, FreeBSD's standard C library is libc. You could have a second copy of libc with special functions provided just for a particular program, and you could make only that program use the special libc while using the standard libc for everything else. FreeBSD allows you to change any shared library any application gets. This sounds weird, but it's terribly useful in all sorts of edge cases. Developers use this feature to test code on a small scale before pushing it out to their whole system. Use */etc/libmap.conf* and files in */usr/local/etc/libmap.d/* to tell rtld(1) to lie to client programs.

While *libmap.conf* entries are useful for developing software, you can also use them to globally replace libraries. Some video card drivers installed via package require you use their driver rather than certain system libraries. A few Nvidia drivers want to provide libGL graphics functions. Don't overwrite the libGL package that everything depends on: instead, remap that library. You can configure library substitution for the whole system, for individual program names, or for the program at a specific full path.

A libmap file (either *libmap.conf* or a file in */usr/local/etc/libmap.d/*) has two columns. The first column is the name of the shared library a program

requests; the second is the shared library to provide instead. All changes take place the next time the program is executed; no reboot or daemon restart is required. For example, here we tell the system, whenever any program requests libGL, to offer it the Nvidia version of the library instead. These global overrides must appear first in *libmap.conf* :

```
libGL.so         libGL-NVIDIA.so
libGL.so.1       libGL-NVIDIA.so.1
```

"May I have libGL.so.1?"

"Certainly, here's libGL-NVIDIA.so.1."

Globally remapping libraries is a rather bold step that might get you talked about by other sysadmins, but remapping libraries on a program-by-program basis is much less ambitious and more likely to solve more problems than it creates. Simply specify the desired program in square brackets before the remapping statements. If you specify the program by its full path, the remap will work only if you call the program by its full path. If you give only the name, the remap will work whenever you run any program of that name. For example, here we remap emacs(1) so that it uses Nvidia's library instead of the system library when called by its full path:

```
[/usr/local/bin/emacs]
libGL.so         libGL-NVIDIA.so
libGL.so.1       libGL-NVIDIA.so.1
```

How can you prove this worked? Well, check ldd(1):

```
# ldd /usr/local/bin/emacs | grep libGL
        libGL.so.1 => /usr/local/lib/libGL-NVIDIA.so.1 (0x80ad60000)
```

You can see that when */usr/local/bin/emacs* requests libGL.so.1, rtld(1) attaches it to libGL-NVIDIA.so.1 instead. We specified the full path to the Emacs binary, however, so we need to call the program by its full path. Try to use ldd(1) on Emacs without calling it by its full path:

```
# cd /usr/local/bin
# ldd emacs | grep libGL
        libGL.so.1 => /usr/local/lib/libGL.so.1 (0x0x8056fa000)
```

By going to */usr/local/bin* and running ldd(1) directly on Emacs without having to specify the full path, rtld doesn't see the full path to the emacs(1)

binary. */etc/libmap.conf* says to use Nvidia's library only for the full path of */usr/local/bin/emacs*. When plain naked `emacs` requests libGL.so.1, it gets what it asked for.

If you want to have a program use the alternate library no matter whether it's called by full path or base name, just give the program name in square brackets rather than the full name:

```
[emacs]
libGL.so        libGL-NVIDIA.so
libGL.so.1      libGL-NVIDIA.so.1
```

Similarly, you can choose an alternate library for all of the programs in a directory by listing the directory name followed by a trailing slash. In this */usr/local/etc/libmap.d/oracle* file, we force all programs in a directory to use an alternate library:

```
[/opt/oracle/bin/]
libc.so.7      libc-special.so.2
```

Using *libmap.conf* lets you arbitrarily remap shared libraries. Developers use this feature to test code. Ports use this to override libraries for certain programs. You'll find a use for it too.

Running Software from the Wrong OS

Traditional software is written for a particular OS and runs only on that OS. Many people built healthy businesses changing software so that it would run on another system, a process called *porting*. As an administrator, you have a few different ways to use software written for a platform other than FreeBSD. The most effective is to recompile the source code to run natively on FreeBSD. If this isn't possible, you can run nonnative software under an emulator, such as Wine, or by reimplementing the application binary interface (ABI) of the software's native platform.

Recompilation

Many FreeBSD packages are actually ports of software originally designed for other platforms. (That's why it's called the *Ports* Collection.) Software written for Linux, Solaris, or other Unix-like operating systems can

frequently be recompiled from source code with little or no modification and run flawlessly on FreeBSD. By simply taking the source code and building it on a FreeBSD machine, you can run foreign software natively on FreeBSD.

Recompiling works best when the platforms are similar. Unix-like platforms should be fairly similar, no? FreeBSD and Linux, for example, provide many similar system functions; both are built on the standard C functions, both use similar tools, both use the GCC compiler, and so on. Over the years, though, the various Unix-like operating systems have diverged. Each version of Unix has implemented new features, new libraries, and new functions, and if a piece of software requires those functions, it won't build on other platforms. The POSIX standard was introduced, in part, to alleviate this problem. POSIX defines the minimal acceptable Unix and Unix-like operating systems. Software written using only POSIX-compliant system calls and libraries should be immediately portable to any other POSIX-compliant operating system, and most Unix vendors comply with POSIX. The problem is ensuring that developers comply with POSIX. Many open source developers care only about having their software run on their preferred platform. Much Linux-specific software is not only not POSIX-compliant but also contains a bunch of unique functions commonly called *Linuxisms*. And POSIX-only code doesn't take advantage of any special features offered by the operating system.

In all fairness, FreeBSD also has FreeBSDisms, such as the hyperefficient data-reading system call kqueue(2). Other Unix-like operating systems use select(2) and poll(2) instead or implement their own system calls. Application developers ask themselves whether they should use kqueue(2), which would make their software blindingly fast on FreeBSD but useless everywhere else, or they should use select(2) and poll(2) to allow their software to work everywhere, albeit more slowly. The developer can invest more time and support kqueue(2), select(2), poll(2), and any other OS-specific variant equally, but while this pleases users, it rather sucks from the developer's perspective.

FreeBSD takes a middle road. If a piece of software can be recompiled to run properly on FreeBSD, the ports team generally makes it happen. If the software needs minor patches, the ports team includes the patches with the port and sends them to the software developer as well. Most software developers gladly accept patches that would allow them to support another operating system. Even though they might not have that OS available to test,

or they might not be familiar with the OS, a decent-looking patch from a reputable source is usually accepted.

Emulation

If software would require extensive redesign to work on FreeBSD, or if the source code is simply unavailable, we can try emulation. An *emulator* translates system and library calls for one operating system into the equivalent calls provided by the local operating system, so programs running under the emulator think they're running on their native system. Translating all these calls creates additional system overhead, however, which impacts the program's speed and performance.

FreeBSD supports a wide variety of emulators, most of which are in the Ports Collection under */usr/ports/emulators*. In most cases, emulators are useful for education or entertainment. If you have an old Commodore 64 game that you've had an itch to play again, install */usr/ports/emulators/frodo*. (Be warned: Mounting that C64 floppy on a modern FreeBSD system will teach you more about disks than humanity was meant to know.) There's a Nintendo GameCube emulator in */usr/ports/emulators/dolphin-emu*, a PDP-11 emulator in */usr/ports/emulators/simh*, and so on.

Emulators, though way cool, aren't really useful for servers, so we won't cover them in any depth.

ABI Reimplementation

In addition to recompiling and emulating, the final option for running foreign programs is the one FreeBSD is best known for: *application binary interface (ABI) reimplementation*. The ABI is the part of the kernel that provides services to programs, including everything from managing sound cards to reading files to printing on the screen to starting other programs. As far as programs are concerned, the ABI is the operating system. By completely implementing the ABI of a different operating system on your native operating system and providing the userland libraries used by that operating system, you can run nonnative programs as if they were on the native platform.

While ABI reimplementation is frequently referred to as emulation, it isn't. When implementing ABIs, FreeBSD isn't emulating the system calls

but rather providing native implementations for the application. No program runs to translate the system calls to their FreeBSD equivalents, and there's no effort to translate userland libraries into FreeBSD ones. By the same token, it would be incorrect to say, "FreeBSD implements Linux." When this technique was created, there was no one word to describe it, and even today there isn't really a good description. You can say that FreeBSD implements the Linux system call interface and includes support for directing a binary to the appropriate system call interface, but that's quite a mouthful. You'll most often hear it referred to as a *mode*, as in "Linux mode."

The problem with ABI reimplementation is overlap. Many operating systems include system calls with generic names, such as read, write, and so on. FreeBSD's read(2) system call behaves very differently from Microsoft's read() system call. When a program uses the read() call, how can FreeBSD know which version it wants? You can give your system calls different names, but then you're violating POSIX and confusing the program. FreeBSD works around this by providing multiple ABIs and controlling which ABI a program uses through *branding*.

Binary Branding

Operating systems generally have a system function that executes programs. When the kernel sends a program to this execution engine, it runs the program.

Decades ago, the BSD (Unix at the time) program execution system call was changed to include a special check for programs that began with `#!/bin/sh` and to run them with the system shell instead of the execution engine. BSD took this idea to its logical extreme: its execution engine includes a list of different binary types. Each program's binary type directs it to the correct ABI. Thus, a FreeBSD system can implement multiple ABIs, keep them separate, and support programs from a variety of different operating systems.

The nifty thing about this system is that there's minuscule overhead. As FreeBSD must decide how to run the program anyway, why not have it decide what ABI to use? After all, binaries for different operating systems all have slightly different characteristics, which FreeBSD can use to identify them. FreeBSD just makes this process transparent to the end user. A binary's identification is called its *branding*. FreeBSD binaries are branded

FreeBSD, while binaries from other operating systems are branded appropriately.

Supported ABIs

As a result of this ABI redirection, FreeBSD can run Linux binaries as if they were compiled natively. Older versions of FreeBSD could also run OSF/1, SCO, and SVR4 binaries, but the demand for these platforms has dramatically decreased.[3] If you need one of these, you might try running an older version of FreeBSD on a virtual machine.

Linux mode, also known as the *Linuxulator*, is quite thorough because Linux's source code is available and its ABI is well documented. In fact, Linux mode works so well that many programs in the Ports Collection rely on it.

Installing and Configuring the Linuxulator

While ABI reimplementation solves one major issue, programs require more than just the ABI. Without shared libraries, supporting programs, and the rest of the userland, most programs won't run properly. No matter which ABI you use, you must have access to the userland for that platform.

If you want to use a piece of Linux software available in the Ports Collection, install the port. That automatically installs any userland dependencies.

If you're looking to run an arbitrary piece of Linux software, you must install a Linux userland first. FreeBSD usually has a couple different Linux userlands available as packages. To see what's available, search the package database for linux_base.

```
# pkg search linux_base
linux_base-c6-6.9_2            Base set of packages needed in Linux mode (Linux CentOS
6.9)
linux_base-c7-7.3.1611_6       Base set of packages needed in Linux mode (Linux CentOS
7.3.1611)
```

This version of FreeBSD has two Linux userlands: one based on CentOS 6.9 and one based on CentOS 7.3. The Linux distribution might change in the future, depending on Linux's direction.

Check to see what versions of Linux your software runs on. Install the most appropriate userland for your application. FreeBSD installs Linux userlands under */usr/compat/linux*.

The port also loads the Linux mode kernel module. To load that module automatically at boot, use this *rc.conf* entry:

```
linux_enable="YES"
```

That's it! Linux mode isn't a proper service, as you can't restart it or get the status, so you can't configure it with service(8). Run `/etc/rc.d/abi start` to activate Linux mode without rebooting.

Before we dive into running a Linux program, let's explore the userland a bit.

The Linuxulator Userland

Just as the *linux.ko* kernel module provides the Linux ABI, the Linuxulator requires a very minimal Linux userland. Take a look under */usr/compat/linux* and you'll see something much like the following:

```
# ls
bin     etc     lib64   proc    selinux sys     var
dev     lib     opt     sbin    srv     usr
```

Looks a lot like the contents of FreeBSD's / directory, doesn't it? If you poke around a bit, you'll find that, generally speaking, the contents of */usr/compat/linux* are comparable to your core FreeBSD installation. You'll find many of the same programs in both.

One thing Linux devotees immediately notice about any linux_base port is that its contents are minimal compared to a typical Linux install. That's because each Linux-based package installs only what it requires to run. FreeBSD's Linux packages impose the minimalist BSD philosophy on Linux software.

Whenever possible, programs in Linux mode try to stay under */usr/compat/linux*, which is somewhat like a weak jail (see Chapter 22). When you execute a Linux binary that calls other programs, the Linux ABI first checks for the program under */usr/compat/linux*. If the program doesn't exist there, Linux mode looks in the main system. For example, suppose you have a Linux binary that calls ping(8). The ABI first searches under

/usr/compat/linux/ for a ping program; as of this writing, it'll find none. The ABI then checks the main FreeBSD system, finds */sbin/ping*, and uses it. The Linuxulator makes heavy use of this fallback behavior to reduce the size of the Linux mode's userland.

Alternatively, suppose a Linux binary wants to call sh(1). The Linux ABI checks under */usr/compat/linux*, finds */usr/compat/linux/bin/sh*, and executes that program instead of the FreeBSD native */bin/sh*.

linprocfs and tmpfs

Linux uses a process filesystem, or *procfs*. FreeBSD eliminated procfs as a default decades ago as a security risk, but some Linux programs will require it. Using Linux software that requires procfs means accepting the inherent risks. FreeBSD makes a Linux procfs available as linprocfs(5).

To enable linprocfs(5), add the following to */etc/fstab* after installing the Linuxulator:

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

FreeBSD loads filesystem kernel modules on demand, so enter `mount /compat/linux/proc` to activate linprocfs(5) without rebooting.

Many Linux programs also expect */dev/shm* for shared memory. FreeBSD can emulate this with tmpfs(5).

```
tmpfs    /compat/linux/dev/shm  tmpfs   rw,mode=1777    0       0
```

Enter `mount /compat/linux/dev/shm`, and the shared memory device is ready.

Testing Linux Mode

Now that you have some idea what's installed in Linux mode, testing Linux functionality is easy. Run the Linux shell and ask it what operating system it's running on:

```
# /usr/compat/linux/bin/sh
sh-4.1# uname -a
Linux storm 2.6.32 FreeBSD 12.0-CURRENT #0 r322672: Fri Aug 17 16:31:34 EDT
2018 x86_64 x86_64 x86_64 GNU/Linux
sh-4.1#
```

When we ask what type of system this command prompt is running on,

this shell responds that it's a Linux system running on top of a Linux 2.6.32 kernel called *FreeBSD*. Pretty cool, eh?

Remember, however, that Linux mode isn't a complete Linux userland. You can't cross-compile software in the default Linuxulator install. You can perform only very basic tasks.

Identifying and Setting Brands

Branding software binaries is easier than branding cattle, but not nearly as adventurous. Most modern Unix-like binaries are in ELF format, which includes space for a comment. That's where the brand lives. FreeBSD assigns each program an ABI by the brand on that binary. If a binary has no brand, it's assumed to be a FreeBSD binary.

View and change brands with brandelf(1):

```
# brandelf /bin/sh
File '/bin/sh' is of brand 'FreeBSD' (9).
```

No surprise there. This is a FreeBSD binary, so it'll be executed under the FreeBSD ABI. Let's try a Linux binary:

```
# brandelf /usr/compat/linux/bin/sh
File '/usr/compat/linux/bin/sh' is of brand 'Linux' (3).
```

See the brands FreeBSD supports with the `-l` flag.

```
# brandelf -l
known ELF types are: FreeBSD(9) Linux(3) Solaris(6) SVR4(0)
```

If you have a foreign program that won't run, check its branding. If it isn't branded or is branded incorrectly, you've probably discovered your problem: FreeBSD is trying to run the program under the native FreeBSD ABI. Change this by setting the brand manually with `brandelf -t`. For example, to brand a program Linux, do this:

```
# brandelf -t Linux /usr/local/bin/program
```

The next time you try to run the program, FreeBSD will run it under the Linux ABI and the Linux userland, and the program should work as expected.

You can also use sysctls to set a *fallback brand*. All FreeBSD binaries get

branded properly, but random programs you copy to your host might not be. Unbranded binaries get treated with the chosen fallback brand. The sysctl `kern.elf32.fallback_brand` gives a fallback brand for 32-bit hosts, while `kern.elf64.fallback_brand` sets the fallback brand for 64-bit hosts. This sysctl takes the brand's numerical identifier, which for Linux is 3.

```
# sysctl kern.elf64.fallback_brand=3
```

You should now be able to run Linux programs without any further configuration. All that's left are the minor annoyances and peccadilloes of Linux mode. Sadly, there's a few of those, as we'll illustrate next.

Using Linux Mode

Many Linux programs are available only as ports. The Ports Collection is smart enough to realize that a piece of software needs Linux mode and chooses the appropriate pieces of Linux to install. One popular choice is Skype. Installing this port triggers installation of the proper Linux userland.

The downside of having a minimal Linux userland is that any port will have a whole bunch of dependencies. Some of those will be FreeBSD binaries, others Linux. I recommend using the port's `make missing` command to display missing dependencies, or even to auto-install dependencies from packages as discussed in Chapter 16. Once all the required packages are installed, once you have linprocfs and the Linux shared memory device installed, and once all the kernel modules are loaded, installing Skype is as easy as `make install clean`.

Debugging Linux Mode

Linux mode isn't Linux, and nowhere is this clearer than when a program breaks. Many programs have cryptic error messages, and Linux mode can obscure them further. You need tools that can dig past the error messages and see what's really going wrong.

Linux Mode and truss(1)

The best tool I've ever found for debugging Linux mode is truss(1), the FreeBSD system call tracer. Some people have told me that using truss(1) for

this is like putting the 12-cylinder engine from a Mack truck into a Volkswagen Beetle, but after much thought and careful consideration, I've decided that I don't care. It works. Once you learn about truss(1), you'll wonder how you ever lived without it.[4]

The truss(1) program identifies exactly which system calls a program makes and the results of each call. Remember, system calls are a program's interface to the kernel. When a program tries to talk to the network, open a file, or even allocate memory, it makes a system call. This makes truss(1) an excellent way to see why a program is failing. Programs make a lot of system calls, which means that truss(1) generates a huge amount of data, making debugging with truss(1) a good candidate for script(1).

So let's run Skype.

```
$ skype
Segmentation fault (core dumped)
```

Here's the good news: The program runs! The bad news is, it chokes on something. The most common errors I find are missing libraries, files, and directories, but which is it? The output of truss(1) can tell me. Start a script(1) session, run the program under truss(1), and end the script.

Your script file will be hundreds or thousands of lines long; how can you possibly find the problem? Search for a relevant part of your error message or for the string ERR. In this case, I searched for the string `directory` and found this near the end of the output:

```
$ truss skype
--snip--
linux_open("/usr/local/Trolltech/Qt-4.4.3-
static/lib/tls/i686/sse2/libasound.so.2",0x0,00)
ERR#-2 'No such file or directory'
linux_stat64("/usr/local/Trolltech/Qt-4.4.3-static/lib/tls/i686/sse2",0xffffb248)
ERR#-2 'No
such file or directory'
linux_open("/usr/local/Trolltech/Qt-4.4.3-static/lib/tls/i686/libasound.so.2",0x0,00)
ERR#-2 'No such file or directory'
--snip--
```

Aha! Skype can't find needed libraries. The package maintainer might have missed these, or perhaps I've screwed up somehow. Check to see whether the libraries exist on your host. If not, you'll need to install them. Perhaps a port exists. Or I might need to install the Linux package.

Installing Linux Packages

If a port doesn't exist for the Linux libraries or software you need, you have a couple choices. One is to create a port for that software. Ports are a great way to be involved with the FreeBSD community. If your goal is to get the software up and running so you can get on with your day, however, you'll need to install the appropriate Linux software from the source RPM. Be warned, though: once you install something outside of the Ports Collection, you'll need to maintain it by hand.

Find the RPM for the software you want. Be sure that the package version matches that installed in linux_base. It's no good to find a CentOS 8 package for your missing libraries if your FreeBSD host uses CentOS 7.3.1611. Download the RPM.

COMMERCIAL LINUX SOFTWARE AND LINUX MODE

Remember, commercial software vendors don't support their Linux software in FreeBSD's Linux mode. If you're in an industrial environment with service-level agreements and run the risk of paying penalties, think very carefully before using Linux mode. The main benefit of commercial software is having someone to blame when it breaks, but FreeBSD's Linux mode eliminates that benefit.

Suppose my life has taken a horrible turn[5] and I need to run Supermin in Linux mode. I find and download the package file, and then I install it with tar(1). FreeBSD's libarchive-based tar can crack open RPM files as well as it does everything else.

```
# cd /compat/linux
# tar -xf /home/mwl/supermin-5.1.16-4.el7.x86_64.rpm
```

Now I get to find the next missing dependencies. Once I have the whole list of dependencies, I'll write a port to save others this tedium.

Running Software from the Wrong Architecture or Release

When you run FreeBSD's amd64 platform, you'll eventually find some piece of software that's available only for i386 platforms. If your kernel has the `COMPAT_FREEBSD32` option (already in GENERIC), FreeBSD/amd64 can run all FreeBSD/i386 software. What you can't do is use FreeBSD/amd64 shared libraries for FreeBSD/i386 software. If you want to run a complicated 32-bit program on a 64-bit computer, you must provide 32-bit versions of the necessary libraries. This is well supported; if you check *rc.conf*, you'll find the ldconfig(8)'s options `ldconfig32_paths` and `ldconfig_local32_dirs`. These options are specifically for telling your amd64 system where to find 32-bit libraries. FreeBSD includes 32-bit libraries on the installation media.

Additionally, FreeBSD can run software from older versions of FreeBSD. The GENERIC kernel includes all the system calls, but you'll still need the base system libraries. These libraries are available as packages, one for each major FreeBSD release. Each package is named *compat*, followed by a version number, and ending in *x*. If you must run a FreeBSD 8 binary, install the compat8x package. The compat packages include 64-bit and 32-bit libraries.

If you need to run binaries that aren't i386 or amd64, you can even use binmiscctl(8) to automatically fire up the proper emulator whenever you run a non-x86 binary.

While there's always more to learn about software management, you now know enough to scrape by. Let's go on and learn about upgrading FreeBSD.

18
UPGRADING FREEBSD

Upgrading servers is perhaps the most annoying task in the system administrator's routine. I can manage unexplained behavior on my desktop after an upgrade, but when my whole organization or hundreds of customers depend on one server, even thinking of touching that system makes my bowels churn. Any operating system upgrade can expand your burgeoning gray hair collection. Even very experienced sysadmins, faced with a choice between upgrading a critical system in-place and jabbing red-hot needles into their own eyes, frequently have to sit down and consider their choices. Virtualized and orchestrated cloud systems might seem less troublesome, but even with these, preparing for an upgrade can cause sleepless nights. Remember, despite all its benefits, automation is a wonderful way to go wrong at scale.

One of FreeBSD's greatest strengths is its upgrade procedure. FreeBSD is designed as a monolithic operating system, not a collection of packages. (Even if FreeBSD migrates to providing the base system as packages, it will remain designed and built as a monolithic entity.) I've had hosts running through five different major releases of FreeBSD and innumerable patch levels in between without reinstalling the system. I decommission FreeBSD systems only when they are so old that the risk of hardware failure keeps me awake at night.[1] While I might worry about applications running on top of the operating system, even upgrading across major FreeBSD releases doesn't worry me anymore.

FreeBSD Versions

Why is upgrading FreeBSD a relatively simple matter? The key is FreeBSD's development method. FreeBSD is a continually evolving operating system. If you download the current version of FreeBSD in the afternoon, it'll be slightly different from the morning version. Developers from around the world continually add changes and improvements, which makes the traditional strict release numbering used by less open software impractical. At any given moment, you can get several different versions of FreeBSD: releases, -stable, -current, and snapshots.

Releases

FreeBSD issues major and minor releases. A *major release* has a version number like 11.0, 12.0, 13.0, or so on. Each major release includes large features not found in earlier major releases. Sweeping changes appear only in major releases.

A *minor release* is an updated version of a major release. You'll see minor releases like 12.1-RELEASE, 12.2-RELEASE, 12.3-RELEASE, and so on. (Most people drop the word *release* from these names.) These minor releases add small features and bug fixes to the major release. You might get new functions or programs, but only if they don't interfere with the existing functions. Unexpected, disruptive changes are avoided.

You'll also see *patch levels*. Thanks to freebsd-update(8), patching a release is quick and easy. Patch numbers are given as numbers after the release. This means you'll see FreeBSD versions like 12.1-RELEASE-p20, 11.4-RELEASE-p9, 13.0-RELEASE-p31, and so on.

Users are expected to closely track their major release by upgrading through successive minor releases, much like other operating systems.

FreeBSD-current

FreeBSD-current, also called *-current* or *HEAD*, is the bleeding-edge, latest version of FreeBSD, which contains code that's making its first public appearance. While the developers have test servers and post patches for review before applying, that's still much less exposure than the wide userbase of FreeBSD-current. FreeBSD-current is where much initial peer review

takes place; at times, current undergoes radical changes that give experienced sysadmins migraines.

FreeBSD-current is made available for developers, testers, and interested parties, but it's not intended for general use. Support for user questions about -current is very slim because the developers simply don't have time to help a user fix his web browser while thousands more critical problems demand attention. Users are expected to help fix these problems or to patiently endure them until someone else fixes them.

To make matters worse, -current's default settings include assorted debugging code, special warnings, and related developer features. These make -current run slower than any other version of FreeBSD. You can disable all this debugging, but if you do so, you won't be able to file a proper trouble report when you have a problem. This means that you're even more out on your own. Check out the file */usr/src/UPDATING* on a -current system for debugging details.

If you can't read C and shell code, don't feel like debugging your OS, don't like computer functions failing arbitrarily, or just don't like being left hanging until your problem annoys someone who can fix it, -current isn't for you. The brave are certainly welcome to try -current, as is anyone willing to devote a large amount of time to learning and debugging FreeBSD or anyone who needs a lesson in humility. You're not forbidden to use -current; you're just on your own. FreeBSD-current isn't always the bleeding edge, but sometimes it might be the why-are-my-fingers-suddenly-little-wiggling-stumps? edge. You've been warned.

To run -current, you really *must* read the *FreeBSD-current@FreeBSD.org* and *svn-src-head@FreeBSD.org* mailing lists. These are high-traffic lists with hundreds of warnings, alerts, and comments a day. If you're reading this book, you probably shouldn't post on these lists; just read and learn. If someone discovers that the newest filesystem patches transform hard drives into zombie minions of Cthulhu, this is where the information will be made available.

-current Code Freezes

Every 12 to 18 months, FreeBSD-current goes through a month of *code freeze*, during which no noncritical changes are permitted and all known critical problems are being fixed. The goal is to stabilize FreeBSD's latest

and greatest and to polish off the rough corners. At the end of the code freeze (or shortly after), -current becomes the *.0* version of a new FreeBSD major release. For example, FreeBSD 12.0 was -current at one point, as was FreeBSD 13.0. When a new major release happens, the -current version number gets incremented. The release of FreeBSD 17.0 means that -current will be called *FreeBSD 18*.

Once the .0 major release escapes into the wild, development work branches into two lines: FreeBSD-current and FreeBSD-stable.

FreeBSD-stable

FreeBSD-stable (or just *-stable*) is the "bleeding edge for the average user," containing some of the most recent peer-reviewed code. Sysadmins familiar with Linux know -stable as a "rolling release." You'll find a version of FreeBSD-stable for each major release.

Once a piece of code is thoroughly tested in -current, it might be merged back into -stable. The -stable version is the one that's mostly safe to upgrade to at almost any time; you might think of it as FreeBSD-beta.

Three or four times a year, the Release Engineering team asks the developers to focus on resolving outstanding problems with -stable rather than making major changes. The Release Engineering team cuts several release candidates from this code and offers each for public testing. When the FreeBSD team is satisfied with the results of its own and the community's testing, the result is given a release number. The development team then returns their attention to their regular projects.[2]

How does this work in practice? Consider FreeBSD 13. Once 13.0 comes out, developers will start merging bug fixes and additions to the 13-stable version. Users who want to help test the new FreeBSD release can run 13-stable. After a few months of merging features and some testing, 13-stable will become 13.1. After 13.1 comes out, that development track reverts to 13-stable. FreeBSD 13.1, 13.2, and 13.3 are just points on the continuum of FreeBSD 13-stable.

FreeBSD-stable is expected to be calm and reliable, requiring little user attention.

As -stable ages, the differences between -stable and -current become greater and greater, to the point where it becomes necessary to branch a new

-stable off of -current. The older -stable is actively maintained for several months while the new -stable is beaten into shape. Some users upgrade to the new version of -stable immediately, others are more cautious. After a release or two of the new -stable, the older -stable is obsoleted and the developers encourage users to migrate to the new version. After some time, the older stable will receive only critical bug fixes, and finally, it'll be abandoned entirely. You can see how this works in Figure 18-1.

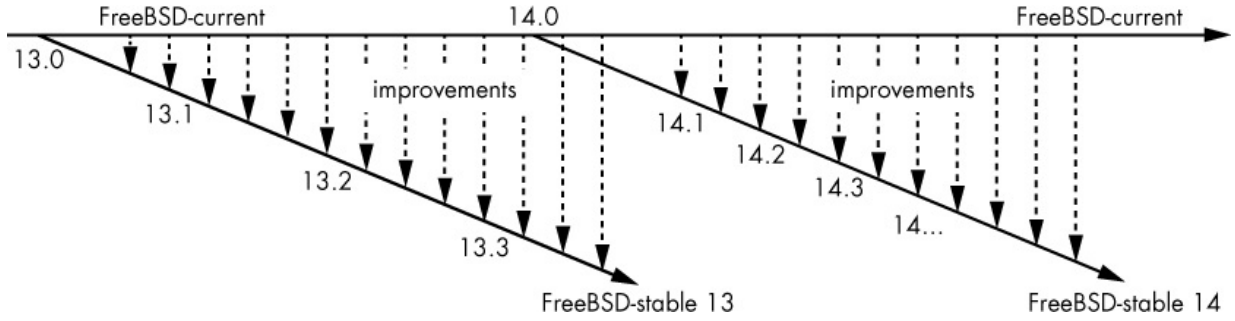


Figure 18-1: FreeBSD development branches

Each release really should have a little dangling tail off to the side for patch levels, but that makes the diagram really confusing.

Users of FreeBSD-stable must read the *FreeBSD-stable@FreeBSD.org* mailing list. While this mailing list has a moderate level of traffic and a fair amount of question-and-answer exchanges that really should be on -*questions*@, important messages from developers generally have a subject beginning with HEADS UP. Look for those messages; they generally mean that a change in the system can ruin your day if you don't know about it.

THE STABILITY OF -STABLE

The word *stable* describes the code base, not FreeBSD itself. Running code from a random point along a stable branch doesn't guarantee that your system will be stable, only that the underlying code won't change radically. The API and ABI are expected to remain unchanged. While the developers take pains to ensure that -stable remains, well, stable, mistakes can and do happen. If this risk worries you, stick with a patched release.

Merging from -current

The phrase *merged from -current (MFC)* means that a function or subsystem has been backported from FreeBSD-current into FreeBSD-stable (or, rarely, into an errata branch). Not all features are MFC'd, however. The point of FreeBSD-current is that it's where major changes take place, and many of those changes require months of testing and debugging. Those large changes can't be backported, as they'd badly impact the -stable users who expect a stable codebase.

Sometimes, features that seem "obvious MFC candidates" can't be merged. Sometimes the kernel infrastructure changes to support new drivers and features, and that infrastructure can't be safely merged. New drivers that require such infrastructure can't be MFC'd. This happens most often with video and wireless network drivers.

Select new drivers, bug fixes, and minor enhancements can be MFC'd—but that's about it. The FreeBSD Project makes it a point not to MFC large changes that could break user applications.

Snapshots

Every month or so, the FreeBSD Release Engineering team releases snapshots of -current and -stable and makes them available on an FTP site. Snapshots are just points along the development branch; they undergo no special packaging or testing. Snapshots don't receive the same attention to quality that releases do, but they're intended as a good starting point for people interested in running -current or -stable. There's only modest quality control, and many developers have no idea that a snapshot has come out until it appears on the FTP servers. You'll find bugs. You'll find errors. You'll experience issues that will turn your mother's hair white, assuming you haven't done that to the poor woman already.

FreeBSD Support Model

With FreeBSD 11.0, the Project's support model changed to more closely resemble that used by other commercial and noncommercial operating systems.

Each major release is supported with security and stability patches for five years after the first release. If FreeBSD 13 is released on January 1, 2021, support will end on January 1, 2026.

Each minor release is supported for three months after the release of the next minor release. Support for FreeBSD 12.3 will end three months after the release of FreeBSD 12.4. This gives you three months to schedule an upgrade.

A loss of official support doesn't mean that you can't continue to run unsupported versions. However, you'll need to examine each security announcement, determine whether it affects your environment, and create your own patches. You're better off running the upgrade.

The whole point of FreeBSD minor releases is that they're minimally intrusive. Upgrading from FreeBSD 12.3 to 12.4 should have a similar impact to applying Windows updates or going from Centos 8.1 to 8.2. Applications should continue to run just fine.

The last FreeBSD minor release of a version gets supported and patched out to the five-year mark. If FreeBSD 12.5 is the last release of FreeBSD 12, it'll get security patches out until five years after the release of FreeBSD 12.0.

Testing FreeBSD

Each version and release of FreeBSD is tested in a variety of ways. Individual developers check their work on their own hardware and ask each other to double-check their work. If the work is sufficiently complicated, they might use the official FreeBSD Phabricator system (*https://reviews.FreeBSD.org/*) or even a private source code repository to offer their work to a broader community before committing it to -current. Analysis companies have donated analysis software to the FreeBSD team so that the source code can be automatically audited, tested, and debugged on an ongoing basis, catching many errors before they have a chance to affect real-world users. Corporations such as Sentex, EMC, Netflix, and iX Systems provide testing for the Project. Several highly regarded FreeBSD developers have made testing a major issue within the Project. They now have an automated Kyua testing framework.

Ultimately, however, a volunteer project with a few hundred developers

can't purchase all computer hardware ever made, nor can they run that hardware under all possible loads. The FreeBSD Project as a whole relies on companies and people that use FreeBSD to test the software.

The most useful testing comes from users who have real-world equipment and real-world testbeds with real-world workloads. Sadly, most of these users perform testing when they put a release CD into the computer, run an install, and fire up the system. At that point, it's too late to benefit the release. Any bugs you find might help the next release, but in the meantime, a patch might fix your problem.

The solution here is obvious—test FreeBSD on your real-world workloads before the release is cut. Requests for testing of new -stable releases appear on *FreeBSD-stable@FreeBSD.org*. By testing a -stable or -current, you'll get even better value from FreeBSD.

Which Version Should You Use?

-current, -stable, releases, snapshots—the head spins. Yes, this seems complicated, but it ensures specific quality levels. Users can rest assured that a release is as reliable as possible and has survived extensive testing and peer review. The same users know that the nifty new features in -stable and -current are available—if they're willing to assume the risk inherent in each version. So, which version should you use? Here are my suggestions:

Production

If you're using FreeBSD in a production setting, install the most recent minor release. When the next minor release comes out, upgrade to it.

Staging

If you need to know what's coming in the next FreeBSD minor release and how it'll affect your production environment, track -stable in your staging environment.

Test

The question here is what you want to test. The Project appreciates testing on both -current and -stable. If you're in doubt, start by running -stable.

Development

Operating system developers, people with too much spare time and too little excitement, and utter fools should run -current. When -current destroys your MP3 collection, debug the problem and submit a patch to fix it.

Hobby

If you're a hobbyist, run any version! Just keep in mind the limitations of the branch you choose. If you're just learning Unix, I'd recommend -release. Once you have your feet under you, upgrade to -stable. While -current is far more steady than it was 20 years ago, if you think a chance of an adrenaline-boosting system failure makes your day more exciting, that's where to go. The high-stakes gamblers running -current welcome like-minded company.

Upgrade Methods

FreeBSD provides two ways to upgrade: binary updates and building from source.

FreeBSD supports *binary updates* through freebsd-update(8). This is very similar to the binary update services offered for Windows, Firefox, and other commercial software. (The software author states that freebsd-update(8) was named after Windows Update.) You can use FreeBSD Update to upgrade across major releases, minor releases, and patch levels.

Upgrading from source code means downloading the FreeBSD source code, building the programs that make up FreeBSD, and installing them to your hard drive. For example, if you have the source code for FreeBSD 13.1, you can upgrade to that version. This requires more effort to set up and use, but it gives you much more flexibility. Upgrade from source when tracking -stable or -current.

PROTECT YOUR DATA!

Chapter 5 is called "Read This Before You Break Something Else!" for good reason. Upgrades can destroy your data. Back

up your system before attempting any sort of upgrade! I upgrade my desktop every week or so, just for fun (see my earlier comment about adrenaline junkies running -current). But before I upgrade, I make sure that all my important data is safely cached on another machine. Copy your data to tape, file, or whatever, but don't run an upgrade without a fresh backup. You've been warned.

Binary Updates

Many operating systems offer binary updates, where users can download new binaries for their operating system. FreeBSD provides a similar program through freebsd-update(8), allowing you to easily upgrade your system.[3] You can't track -stable or -current with freebsd-update(8), only releases. For example, if you installed FreeBSD 12.0, freebsd-update(8) can upgrade you to 12.0-p9, 12.1, or 13.0, but not 12-stable or 14-current.

If you have a custom kernel, you must build updates to your kernel from source instead of relying upon the update service. Similarly, if you've upgraded a host from source (discussed later this chapter), running freebsd-update(8) overwrites your custom binaries with default ones.

Configure updates in */etc/freebsd-update.conf*.

/etc/freebsd-update.conf

Updating with freebsd-update(8) is designed to be seamless for the average user, and changing its configuration is rarely advisable. You might have unusual circumstances, however, so here are the most useful options you'll find in this file:

`KeyPrint 800...`

`KeyPrint` lists a cryptographic signature for the update service. If the FreeBSD Update service suffered a security breach, the FreeBSD Project would need to repair the breach and issue new cryptographic keys. In this case, the breach would be announced on the security announcements

mailing list (and would also be big news in the IT world). In other words, there's no reason to change this in normal use. (Building your own customized FreeBSD and distributing it via freebsd-update(8), while both possible and practical, is considered abnormal use.)

`ServerName update.freebsd.org`

The `ServerName` tells freebsd-update(8) where to fetch its updates from. While the FreeBSD Project does provide the tools to build your own updates, there's really no need to do so. If you have enough servers that you'd consider building your own update server, you probably also have a proxy server that can cache the official updates.

`Components src world kernel`

By default, FreeBSD Update provides the latest patches for the source code in */usr/src*, the userland (world), and the GENERIC kernel. You might not need all of these components, however. While the userland is mandatory, you might have a custom kernel. Remove the kernel statement to make freebsd-update(8) ignore the kernel. Custom kernel users could also copy the GENERIC kernel to */boot/GENERIC*. The update will update the GENERIC kernel but leave your custom kernel alone. Or, you can remove the `kernel` entry and save yourself the warning. If you don't have the source code installed on your machine, freebsd-update realizes that and doesn't try to patch it, but you could eliminate the `src` entry and save the software the trouble. You could also choose to receive only portions of the source code update, as described in freebsd-update.conf(5).

`UpdateIfUnmodified /etc/ /var/ /root/ /.cshrc /.profile`

The updates include changes to configuration files in */etc*. If you have modified these files, however, you probably don't want freebsd-update(8) to overwrite them. Similarly, */var* is very fluid, designed for customization by the sysadmin; you don't want FreeBSD Update to muck with your settings. FreeBSD Update applies patches to files in the directories listed in `UpdateIfUnmodified` only if they're unchanged from the default.

`MergeChanges /etc/ /boot/device.hints`

Updating to a new release can change configuration files. The update

process will give you a chance to merge changes into files that appear in the `MergeChanges` locations.

`MailTo root`

If you schedule a run of FreeBSD Update (as described later in this chapter), freebsd-update(8) sends an email of the results to the account listed in `MailTo`.

`KeepModifiedMetadata yes`

Perhaps you've modified the permissions or owner of a system file or command. You probably don't want freebsd-update(8) to change those permissions back. With `KeepModifiedMetadata` set to `yes`, freebsd-update(8) leaves your custom permissions and ownership unchanged.

See freebsd-update.conf(5) for more possibilities.

Running freebsd-update(8)

Updating your system with binary updates has two stages: downloading the updates and applying them. The process looks slightly different if you're applying patches versus if you're crossing major releases.

If you're using ZFS, always create a new boot environment before upgrading or patching!

Updating to the Latest Patch Level

To download the latest patches to your chosen release, run `freebsd-update fetch`. Here, I'm updating a FreeBSD 11.0 host to the latest patchlevel.

```
# freebsd-update fetch
```

You'll see the program finding the download sources for the patches, comparing cryptographic keys for those download sources, and eventually downloading patches into */var/db/freebsd-update*. Inspecting the system might take a couple minutes, depending on the speed of your storage.

Occasionally, you'll see a message similar to this:

```
  The following files will be removed as part of updating to 11.0-RELEASE-p12:
❶ /boot/kernel/hv_ata_pci_disengage.ko
❷ /usr/share/zoneinfo/America/Santa_Isabel
```

```
❸ /usr/share/zoneinfo/Asia/Rangoon
```

We're updating a .0 release, the first version of a major FreeBSD release, straight to 11.0-RELEASE-p12, so there's a few accumulated patches. Why would such a patchset start by removing files?

The time zone files are pretty straightforward. Between the release of FreeBSD 11.0 and the present time, Santa Isabel ❷ and Rangoon ❸ changed their time zones. Sadly, nations don't coordinate their time zones with FreeBSD releases. Removing those time zones from the system simplifies life for sysadmins in those countries and doesn't hurt the rest of us.

But they're also removing a kernel module ❶. Why would that happen? A little research on the FreeBSD mailing lists shows that this module should never have been shipped with 11.0, and you *really* shouldn't be using it. This sort of change is rare but can happen right after a major release.

You'll then see files added as part of this patchset, if any.

```
The following files will be added as part of updating to 11.0-RELEASE-p12:
/usr/share/zoneinfo/Asia/Barnaul
/usr/share/zoneinfo/Asia/Famagusta
/usr/share/zoneinfo/Asia/Tomsk
--snip--
```

It seems sysadmins in Rangoon are quite busy this summer.

Almost all patches alter existing files on the system. You'll see those next.

```
The following files will be updated as part of updating to 11.0-RELEASE-p12:
/bin/freebsd-version
/boot/gptboot
/boot/gptzfsboot
/boot/kernel/cam.ko
/boot/kernel/hv_storvsc.ko
--snip--
```

If your release is nearing its End of Life, you'll get a notice like this:

```
WARNING: FreeBSD 11.0-RELEASE-p1 is approaching its End-of-Life date.
It is strongly recommended that you upgrade to a newer
release within the next 1 month.
```

If the release has gone past End of Life, the notice gets more . . . emphatic.

To install the downloaded files, run `freebsd-update install`:

```
# freebsd-update install
Installing updates... done.
```

If the update requires any more steps, you'll see them here. Reboot your system, and you'll see that you're running the newest patchlevel.

Updating Releases

This FreeBSD 11.0-p12 machine is within a month of End of Life? Let's update it with `freebsd-update upgrade`. Specify the target release with the `-r` flag.

```
# freebsd-update -r 11.1-RELEASE upgrade
```

The hardest part of this is to remember that -RELEASE is part of the version name.

The upgrade will inspect your system and present its conclusions.

```
The following components of FreeBSD seem to be installed:
kernel/generic world/base world/lib32

The following components of FreeBSD do not seem to be installed:
kernel/generic-dbg world/base-dbg world/doc world/lib32-dbg

Does this look reasonable (y/n)? y
```

Remember the install process, when you selected FreeBSD components to set up on your new host? That's what freebsd-update is checking for. You could have added or removed components, though, so take a look at the list. If it looks right, hit **y** to continue.

The update will more carefully scrutinize the system, comparing all existing files to the new release, and then start downloading.

```
Fetching 10697 patches.....10....20....30....40....50....60....70....80....90
```

Go make a cup of tea. Depending on your host's bandwidth, you should return to see:

```
Applying patches...
```

You tea's probably too hot. Let it cool a bit.

```
Fetching 236 files...
```

More downloading? Fine, enjoy your tea and watch the program work.

```
Attempting to automatically merge changes in files... done.

The following files will be removed as part of updating to 11.1-RELEASE-p1:
/usr/include/c++/v1/__undef___deallocate
/usr/include/c++/v1/tr1/__undef___deallocate
/usr/include/netinet/ip_ipsec.h
--snip--
```

You can search the mailing list archives and the FreeBSD source code tree to learn why each of these files was removed.

The minor release will include new features backported from -current. Those probably involve adding files to the system.

```
The following files will be added as part of updating to 11.1-RELEASE-p1:
/boot/kernel/amd_ecc_inject.ko
/boot/kernel/bytgpio.ko
/boot/kernel/cfiscsi.ko
/boot/kernel/cfumass.ko
--snip--
```

None of these new features should interfere with existing functions, but perusing the list might give you some interesting reading.

An upgrade should change just about every file on the system, as we'll see next.

```
The following files will be updated as part of updating to 11.1-RELEASE-p1:
/.cshrc
/.profile
/COPYRIGHT
/bin/[
/bin/cat
--snip--
```

Eventually you'll get to:

```
To install the downloaded upgrades, run "/usr/sbin/freebsd-update install".
```

Who are you to ignore instructions?

Up until this point, the update process has only downloaded files and done comparisons in temporary staging areas. The operating system hasn't been touched. If you feel comfortable proceeding, run the installation.

```
# freebsd-update install
```

```
src component not installed, skipped
Installing updates...
Kernel updates have been installed.  Please reboot and run
"/usr/sbin/freebsd-update install" again to finish installing updates.
```

Why reboot between parts of the update? New userland programs might require new kernel features. Installing a nonfunctional version of the `reboot` command results in needing to power cycle the host, which would earn you an embarrassing punch on your geek card.

```
# reboot
```

Once the machine comes back up, complete the userland upgrade.

```
# freebsd-update install
src component not installed, skipped
Installing updates...
Completing this upgrade requires removing old shared object files.
Please rebuild all installed 3rd party software (e.g., programs
installed from the ports tree) and then run "/usr/sbin/freebsd-update
install" again to finish installing updates.
```

What madness is this?

The update process works hard not to leave you with a damaged system or dysfunctional software. If freebsd-update removes older versions of shared libraries required by your add-on software, it won't run. The update pauses so you have a chance to upgrade your software. We discuss upgrading packages and ports later this chapter. Upgrades along a -stable branch don't normally need to remove old cruft.

This last run of freebsd-update removes old shared libraries and such.

```
# freebsd-update install
```

Your upgrade is now complete. As with any time you perform wide-ranging system maintenance, reboot one last time to verify everything starts cleanly.

Reverting Updates

You thought the upgrade would go easily. They always have before. But this time, you were wrong. Some subtle interaction between the new patches and your software has done brung you low. If you're using boot environments,

this is the time to revert to your previous install. If you're not, remove the most recently installed upgrade with freebsd-update's `rollback` command.

```
# freebsd-update rollback
Uninstalling updates... done.
```

A rollback is much faster than installing patches. There's no need to inspect the system; freebsd-update just reads the log of its previous actions and undoes everything.

Scheduling Binary Updates

Best practice would say to download and apply updates at a consistent time on a regular schedule, such as on your monthly maintenance day. The freebsd-update program includes specific support for this to avoid flooding the download servers with requests every hour, on the hour. The `freebsd-update cron` command tells the system to download the updates at a random point in the next hour. Put this command in */etc/crontab* to download updates during that one-hour window. This helps reduce the load on the download servers. You'll get an email when the system has updates, so you can schedule a reboot at your convenience.

Optimizing and Customizing FreeBSD Update

Two common questions about FreeBSD Update concern the custom builds of FreeBSD and distributing updates locally.

Many people build their own versions of FreeBSD for internal use. Frequently, this is just a version of FreeBSD with various sections cut out, but some companies use extensive modifications. If you have deleted files from your FreeBSD install, freebsd-update(8) won't attempt to patch them.

Similarly, many companies like to have internal update servers for patch management. The FreeBSD Update system is specifically designed to work with caching web proxies. While all the files are cryptographically signed and verified, they're transmitted over vanilla HTTP so that your proxy can cache them.

Upgrading via Source

Another way to update your system is to build it from source code. FreeBSD is *self-hosting*, meaning that it includes all the tools needed to build FreeBSD. You don't need to install any compilers or development toolkits. The only thing you need to build a new FreeBSD from source code is the newer source code.

When a developer releases improvements to FreeBSD, the changes are made available worldwide within minutes. The FreeBSD master source code server tracks the source code, all changes made to that code, and the author of those changes. Developers can check in new code, and users can check out the latest versions through *Subversion (SVN)*. FreeBSD has a simple SVN client, svnlite(1), that suffices for all source code operations. It's a standard Subversion client built without any of the complicated options svn(1) supports. Many people find svnlite(1) perfectly adequate for all their non-FreeBSD Subversion needs.

CSUP, CVSUP, CVS, SUP, AND CTM?

Documentation on the internet unfortunately survives well past reason and rises to sow confusion at the worst possible time. Undead FreeBSD documentation and third-party tutorials might mention using a tool called *CVS* or *CVSup* for source code updates. These tools were replaced in 2013. Any mention of these programs indicates you're reading old docs. If you see a mention of CTM, you're reading docs that predate CVS.[4]

Upgrading from source requires console access. You won't always need it, but recovering from a bad build might require intervention at the keyboard. Test your serial console, Java app, or IPMI console before installing your home-built operating system version. In my experience, the only upgrades that require console access are those where I don't have console access.

Which Source Code?

Every FreeBSD release ships with the source code used to build the system you're installing. If you didn't choose to install the source when installing the system, you'll find it on the install media in */usr/freebsd-dist/src.txz*. If you did install the source code, you'll find it in */usr/src*.

Unfortunately, this version of the source code lacks the version control tags needed to build FreeBSD. It's for reference only. You'll need to use svnlite(1) to download a version of the code with those tags intact.

Is your copy of source code in */usr/src* suitable for building FreeBSD? Ask svnlite(1).

```
# svnlite info /usr/src
svn: E155007: '/usr/src' is not a working copy
```

The "not a working copy" error means that any source code here can't be used with Subversion.

The svnlite(1) command to grab source code looks like so.

```
# svnlite checkout ❶svn.freebsd.org/❷repository/❸branch ❹localdir
```

The mirror ❶ is a FreeBSD Subversion mirror. The mirror *svn.FreeBSD.org* is a geo-routed alias for the closest subversion mirror.

The repository ❷ is the group of code you're working with. You can get a complete list of current repositories at *https://svnweb.FreeBSD.org/*. The main Project repositories include *base*, for the operating system; *doc*, for documentation; and *ports*, for the Ports Collection.

The branch ❸ is the version of FreeBSD you want. For the very latest stumpy-fingered -current, use *head*. To get a stable version, use the branch *stable/* followed by the major release. FreeBSD 12-stable would be *stable/12*. To get a release plus all current patches, use *releng/* and the release number. The fully patched FreeBSD 12.2 would be at *releng/12.2*.

If you have trouble figuring out which branch you need, wander through *https://svnweb.freebsd.org/*. The branch literally tells svnlite(1) which subdirectory to grab from this site.

Finally, the localdir ❹ is the local directory where I want to put the source code.

This host is running FreeBSD 11.1. I want to be adventuresome and

move up to FreeBSD 11-stable. Here's how I'd do that:

```
# svnlite checkout https://svn0.us-east.FreeBSD.org/base/stable/11 /usr/src/
Error validating server certificate for 'https://svn0.us-east.freebsd.org:443':
 - The certificate is not issued by a trusted authority. Use the
   fingerprint to validate the certificate manually!
Certificate information:
 - Hostname: svnmir.ysv.FreeBSD.org
 - Valid: from Jul 29 22:01:21 2013 GMT until Dec 13 22:01:21 2040 GMT
 - Issuer: svnmir.ysv.FreeBSD.org, clusteradm, FreeBSD.org, CA,
US(clusteradm@FreeBSD.org)
 - Fingerprint: 1C:BD:85:95:11:9F:EB:75:A5:4B:C8:A3:FE:08:E4:02:73:06:1E:61
(R)eject, accept (t)emporarily or accept (p)ermanently? p
```

What fresh madness is this? We try to get the FreeBSD source code and instead get a certificate error?

Compare the certificate fingerprint shown to the server's fingerprint on the FreeBSD.org website. If it matches, permanently accept the certificate by entering **p**. Filenames of source code files will stream past your terminal.

Once svnlite(1) finishes, take a look in */usr/src*.

```
# ls /usr/src/
COPYRIGHT              cddl                   sbin
LOCKS                  contrib                secure
MAINTAINERS            crypto                 share
Makefile               etc                    sys
Makefile.inc1          gnu                    targets
Makefile.libcompat     include                tests
ObsoleteFiles.inc      kerberos5              tools
README                 lib                    usr.bin
README.md              libexec                usr.sbin
UPDATING               release
bin                    rescue
```

This is the top directory of the FreeBSD source tree, which contains all the code needed to build your chosen FreeBSD version.

Updating Source Code

So you built FreeBSD yesterday? Cool. But now you want to build today's version.

The good news is, Subversion needs only to update the code you've downloaded, not download the whole source code tree. The better news is, the source code records where you got it from and what branch it's supposed to be from in the *.svn* directory. This makes updating much simpler than the

initial download.

FreeBSD has integrated the Subversion commands into the source code. All you'll need to do is tell the system that it may call svnlite(1) for updates by setting SVN_UPDATE in */etc/src.conf*.

```
# echo 'SVN_UPDATE=yes' >> /etc/src.conf
```

Now run `make update` to get the latest source code.

```
# cd /usr/src
# make update
```

You'll see the same sorts of updates flow past. These updates will be much quicker than the original download, though.

Building FreeBSD from Source

Once you have the latest source code, look at */usr/src/UPDATING*. The beginning of this file lists, in reverse chronological order, any warnings and special notices about changes to FreeBSD that are of special interest to people who build from source. These notes tell you whether you must take any particular actions before rebuilding your system or whether any major system functionality has changed. If you want your system to work after the upgrade, follow these instructions exactly.

The end of the *UPDATING* file gives the official instructions for building FreeBSD from source. The procedure described in this book has been used since FreeBSD 6-current, which changed only slightly from 5-current, but I still recommend double-checking the instructions herein against those in *UPDATING*.

If you use a custom kernel, also examine the new GENERIC or NOTES kernel configuration files for any new options or interesting kernel changes.

CUSTOMIZING YOUR FREEBSD BUILD

Remember back in Chapter 16 when we discussed */etc/make.conf* ? FreeBSD uses a separate file to handle customizations for building FreeBSD itself. While settings in

/etc/make.conf affect all software built on the system, anything in */etc/src.conf* affects only building FreeBSD from source.

If you hang around the FreeBSD community for a while, you'll hear all sorts of stories about special methods people use for building FreeBSD. You'll hear anecdotal evidence that one method is faster, more efficient, or somehow mystically "better" than the standard. While you are certainly free to use any build method that strikes your fancy, the only method supported by the FreeBSD Project is that documented at the end of */usr/src/UPDATING*. If you follow some other procedure and have trouble, you'll be referred to the documented procedure.

Build the World

First, build the new userland:

```
# cd /usr/src
# make buildworld
```

The `make buildworld` command builds from source the basic tools needed to build the system compiler and then builds the compiler and associated libraries. Finally, it uses the new tools, compiler, and libraries to build all the software included in a core FreeBSD install. (This is much like building a car starting with the instruction, "Dig iron ore out of the ground.") The `buildworld` places its output under */usr/obj*. It can take anywhere from one to several hours, depending on your hardware. You can continue working normally as the `buildworld` runs, if your hardware is robust enough; while the build consumes system resources, it won't take any of your attention.

When the `buildworld` finishes, confirm that it completed without errors. If the build ends with a bunch of messages like those you see during a failed kernel compile, do not proceed with the upgrade. If you can't figure out why the build failed, go to Chapter 1 to see how you can get help. Never attempt to install a damaged or incomplete upgrade.

PARALLEL MAKE WORLD

Experienced sysadmins have probably used the `-j` flag of make(1) to increase build speed. This starts multiple build processes and allows the system to take advantage of multiple CPUs. If you have a multi-CPU system or if your CPU has multiple cores, `-j` can work when building FreeBSD. A reasonable number of builds to start is one more than the number of CPUs you have. For example, if you have an eight-core processor, you can reasonably use nine build processes by running `make -j9 buildworld && make -j9 kernel`.

The FreeBSD Project doesn't officially support `-j` for upgrades, even though many developers use it. If your build fails when using `-j`, try without `-j` before complaining.

Build, Install, and Test a Kernel

The best way to test your upgrade is to build a new GENERIC kernel. This separates problems in your custom kernel from general FreeBSD issues. The impetuous are certainly welcome to upgrade straight to their custom kernel configuration, but if your kernel fails, you'll need to try a GENERIC kernel. Be sure to compare your custom kernel to the new GENERIC configuration, however, to catch any alterations your custom setup needs. You can use the Subversion history at *https://svnweb.FreeBSD.org/* to compare the kernel configurations of different releases.

By default, the kernel upgrade process builds a GENERIC kernel. If you want to upgrade straight to a custom kernel, use the variable `KERNCONF` to tell make(1) the kernel name. You can set `KERNCONF` on the command line, in */etc/make.conf*, or in */etc/src.conf*.

You can build a new kernel in one of two ways. The `make buildkernel` command builds a new kernel but doesn't install it. Follow a `make buildkernel` with a `make installkernel` to install the kernel. The `make kernel` command runs these two commands right after each other. Use the one that best matches your schedule. For example, if I'm doing a system upgrade at work during my Sunday maintenance window, I might run `make buildworld` and `make buildkernel` during the preceding week to save a few hours of my precious

weekend. I don't want to install that kernel before the maintenance day, however—if the machine has a problem on Friday and needs a reboot, I want to boot the old production kernel and not the new, upgraded kernel. On Sunday morning, when I'm ready to actually upgrade, I run `make installkernel`. On the other hand, using `make kernel` makes sense when upgrading my desktop. So, to upgrade with my custom kernel, I'd run:

```
# make KERNCONF=THUD kernel
```

Again, do not attempt to install a kernel that didn't successfully compile. If your `make buildkernel` errors out and dies, fix that problem before proceeding.

Once you have a new kernel installed, reboot your computer into single-user mode. Why single-user mode? The userland might expect different kernel interfaces than the new kernel provides. While */usr/src/UPDATING* should list all of these, no document can cover all possible changes and their impact on third-party software. Such changes happen rarely on -stable but unpredictably on -current. If you update your host every week, your userland might have a problem on the new kernel. If you haven't updated the host for a year, you get a year's worth of changes dumped on you simultaneously. While many people get away with installing the upgrades in full multiuser mode, single-user mode is safest.

If your system runs correctly in single-user mode with the new kernel, proceed. Otherwise, fully document the issue and boot the old kernel to restore service while you solve the problem.

Prepare to Install the New World

Beware, grasshopper! This is the point of no return. You can easily back out a bad kernel—just boot the older, known good one. Once you install a freshly built world, you can't revert it out without recovering from backup or using a ZFS boot environment. Confirm that you have a good backup before proceeding, or at least recognize that the first irrevocable step is happening right now.

If your new kernel works, proceed to installing your freshly built userland. First, confirm that your system can install the new binaries. Each new version of FreeBSD expects that the old system supports all the

necessary users, groups, and privileges that the new version requires. If a program must be owned by a particular user and that user doesn't exist on the system, the upgrade will fail. That's where mergemaster(8) comes in.

mergemaster(8)

The mergemaster program compares the existing configuration files under */etc* to the new files in */usr/src/etc*, highlights the differences between them, and either installs them for you, sets them aside for evaluation, or even lets you merge two different configuration files into one. This is extremely useful during upgrades. You run mergemaster once before installing the new world to ensure that your system can install the new binaries, and you run it once after installing the new world to synchronize the rest of */etc* with your new world.

Start with mergemaster(8)'s prebuildworld mode, using the `-Fp` flags. The `-F` flag automatically installs any files that differ only by version control information. The `-p` flag compares */etc/master.passwd* and */etc/group* and highlights any accounts or groups that must exist for an `installworld` to succeed.

```
  # mergemaster -Fp

❶  *** Creating the temporary root environment in /var/tmp/temproot
   *** /var/tmp/temproot ready for use
❷  *** Creating and populating directory structure in /var/tmp/temproot

   *** Beginning comparison
```

These initial messages, all preceded by three asterisks, are mergemaster explaining what it's doing. We start with a temporary root directory ❶ in */var/tmp/temproot* so mergemaster can install a pristine set of configuration files ❷ to compare with the installed files. After that, mergemaster shows its first comparison.

```
❶   *** Displaying differences between ./etc/group and installed version:

  --- /etc/group  2017-09-01 11:12:49.693484000 -0400
  +++ ./etc/group 2017-09-01 13:22:15.849816000 -0400
  @@ -1,6 +1,6 @@
❷  -# $FreeBSD: releng/11.1/etc/group 294896 2016-01-27 06:28:56Z araujo $
❸  +# $FreeBSD: stable/11/etc/group 294896 2016-01-27 06:28:56Z araujo $
```

```
    #
❹  -wheel:*:0:root,mwlucas
❺  +wheel:*:0:root
    daemon:*:1:
    kmem:*:2:
    sys:*:3:
  @@ -33,4 +33,3 @@
    hast:*:845:
    nogroup:*:65533:
    nobody:*:65534:
❻  -mwlucas:*:1001:
```

One vital piece of information is the file being compared, and mergemaster displays the filename ❶ up front. We're examining */etc/group* on the installed system and comparing it to a new */etc/group*.

We then see the two different versions of the file being compared, the installed file first ❷ and the upgraded version of the file second ❸. Notice the minus and plus signs at the beginning of these lines. A minus sign indicates that a line is from the currently installed file, while a plus sign shows that a line is from the version in */usr/src*.

This is nicely illustrated by the next two lines mergemaster shows. The first group listed, marked by a minus sign, is for the current wheel group ❹. The second line is the password entry ❺ for the out-of-the-box upgrade. This host's wheel group has a user that's not in the default install. I want to keep my account there.

At the end of the listing, there's a group with a minus sign in front of it ❻. This group exists on the local system, but not in the source code. That's perfectly expected.

None of the changes here are relevant, this time.

Once mergemaster displays all the changes in this file, it displays my options.

```
  Use 'd' to delete the temporary ./etc/group
  Use 'i' to install the temporary ./etc/group
  Use 'm' to merge the temporary and installed versions
  Use 'v' to view the diff results again

  Default is to leave the temporary file to deal with by hand

How should I deal with this? [Leave it for later] d
```

I have four choices. I can delete the temporary */etc/group* with `d`. If I want to throw away my current configuration and install one straight from the source code, I can install it with `i`. If I need some of both the old and new versions, I can use `m`. And if I wasn't paying attention, I can see the comparison again with `v`.

The new */etc/group* has no changes I need. I delete it, letting mergemaster go to the next file, */etc/passwd*.

The mergemaster display of the password file starts off much like the groups file. Yes, the root password has changed—I'd hope so! There's an extra entry for my account. But in the middle of the display, there's an entry like this:

```
   _pflogd:*:64:64::0:0:pflogd privsep user:/var/empty:/usr/sbin/nologin
❶ +_dhcp:*:65:65::0:0:dhcp programs:/var/empty:/usr/sbin/nologin
   uucp:*:66:66::0:0:UUCP pseudo-user
```

The line for the user _dhcp ❶ is preceded by a plus sign, and there's no corresponding _dhcp entry with a minus sign. The user _dhcp exists in the downloaded source code, but not on the currently running system. If a new user appears in the default FreeBSD configuration, it's because a program or files in the new system expect to be owned by that user.

Installing the new world will fail if this user isn't present.

I can't replace my current */etc/passwd*, as it contains active user accounts. I can't throw away the new */etc/passwd* because it has users I need in it. I guess I have to merge the two files together.

```
How should I deal with this? [Leave it for later] m
```

When merging files, mergemaster splits your command window in half with sdiff(1). The left side displays the beginning of the currently installed file, while the right side shows the new version. Only the sections that differ are shown. Pick the side you want in your new *master.passwd* file.

```
# $FreeBSD: releng/11.1/etc/master.passwd 299 | # $FreeBSD:
stable/11/etc/master.passwd 29936
```

This line displays the version control information from both copies of */etc/passwd*. On the left, we have the version of this file from the releng/11.1 branch, or 11.1. On the right, we have the version from stable/11, or 11-

stable. Future mergemaster runs will use the version information (among other tools) to determine whether a file needs updating, so our merged file needs the correct version number. Choose between the left (`l`) and right (`r`) column. I want the entry on the right, so I enter `r`.

Mergemaster displays the next difference.

```
root:$6$fD7a5caQtkZbG93E$wGfw5G2zNORLq8qxlT8z | root::0:0::0:0:Charlie
&:/root:/bin/csh
```

I've changed my root password, and I want to keep it. Enter **`l`** to keep the left-hand version.

```
                                                > _dhcp:*:65:65::0:0:dhcp
programs:/var/empty:/
```

In this next example, there's no entry in the current file and the new user _dhcp is in the new file. We need the user _dhcp to complete the `installworld`, so enter **`r`** to choose the right-hand entry and get the next difference.

```
mwlucas:$1$zxU7ddkN$9GUEEVJH0r.owyAwU0NFX1:10 <
```

And here's my account. If I want to log on as myself after the upgrade, I better enter **`l`**.

Once we walk through every difference in the file, mergemaster presents our next choices:

```
  Use 'i' to install merged file
  Use 'r' to re-do the merge
  Use 'v' to view the merged file
  Default is to leave the temporary file to deal with by hand

    *** How should I deal with the merged file? [Leave it for later]
```

Viewing the merged file is always a good idea, unless you already know you screwed up and want to do it over. Review your merged file with **`v`**, and if it looks correct to you, install it with **`i`**.

```
*** You installed a new master.passwd file, so make sure that you run
    '/usr/sbin/pwd_mkdb -p /etc/master.passwd'
     to rebuild your password files

    Would you like to run it now? y or n [n]y
```

You need to rebuild the password database only if you want your new user account to work. Enter `y`.

You can now install your new userland.

Installing the World

Still in single-user mode, you can install your brand new FreeBSD with `make installworld`. You'll see numerous messages scroll down the screen, mostly including the word *install*.

```
# cd /usr/src
# make installworld
```

You now have a new userland to go with your shiny new kernel. Congratulations!

Obsolete Files

Installing all the new programs isn't quite enough, though. An update can remove programs and files from the base system. To see what's obsoleted, run `make check-old`.

```
# make check-old
>>> Checking for old files
/usr/include/sys/ksyms.h
/usr/lib/clang/4.0.0/include/sanitizer/allocator_interface.h
/usr/lib/clang/4.0.0/include/sanitizer/asan_interface.h
/usr/lib/clang/4.0.0/include/sanitizer/common_interface_defs.h
--snip--
```

This lists every part of the system that was once installed on your system but is no longer required. Confirm that you're no longer using these programs; if you are, either preserve the existing unsupported program or find an alternative.

A little later in the output, you'll see the shared libraries that are now obsolete:

```
>>> Checking for old libraries
/lib/libzfs.so.2
/usr/lib/debug/lib/libzfs.so.2.debug
/usr/lib/libarchive.so.6
/usr/lib/debug/usr/lib/libarchive.so.6.debug
/usr/lib/libmilter.so.5
```

```
--snip--
```

Finally, you might see a list of directories that are no longer required. Removing a directory is fairly rare, compared to removing a file.

If you're not specifically using any of the old programs or directories, delete them with `make delete-old`. make(1) prompts you with the name of each file and asks you to confirm that you want to delete the file.

```
# make delete-old
>>> Removing old files (only deletes safe to delete libs)
remove /usr/include/sys/ksyms.h? y
remove /usr/lib/clang/4.0.0/include/sanitizer/allocator_interface.h? y
remove /usr/lib/clang/4.0.0/include/sanitizer/asan_interface.h?
```

This is stupid. There's dozens of these files. And I'm going to delete every single one. Fortunately, every real Unix includes tools to automate stupidity.

```
# yes | make delete-old
```

All of these files go away. Or, if you want to use FreeBSD's built-in options, use the `BATCH_DELETE_OLD_FILES` flag.

```
# make -DBATCH_DELETE_OLD_FILES delete-old
```

I find yes(1) easier, though.

Obsolete Shared Libraries

Obsolete shared libraries require more care. Many third-party programs link against shared libraries. If you delete the shared library, the program won't run. This can be really, really annoying if you, say, delete the library required by your mission-critical application. The only way to restore service is to recompile the program or replace the shared library. We discuss shared libraries in Chapter 17. If none of your programs require the library, you can delete it. Identifying every program that requires a library is a royal pain, however.

For example, check the list of obsolete shared libraries above. One of the entries is libzfs.so.2. Looking in */lib*, I see that we now have libzfs.so.3. Perhaps I shouldn't need two different versions of the ZFS library. This host uses ZFS, though, and I have a whole bunch of ZFS utilities installed. If I

remove the old version of libzfs, there's a chance one of those programs won't work anymore. The presence of these obsolete library versions doesn't hurt anything in the short term; you can bring your system back on line with the old libraries in addition to the new ones and reinstall your add-on software in a more leisurely manner. We'll update your ports later in this chapter.

If you believe that none of the libraries listed as old are important and you can safely delete them, back up each before removing it. By just copying the library to an *old-libs* directory somewhere, you'll make recovery much simpler when you find out that your mission-critical software doesn't work anymore.

You can also copy old libraries into */usr/lib/compat* so that your programs will continue to run but the old libraries will be safely out of the way. The problem here is that we both know perfectly well that you're never going to go clean up those libraries.

I prefer to back up the libraries and then remove them from the live system. When I find a program doesn't work, I temporarily copy the missing library from the backup into a compat directory. The added annoyance ticks me off enough to solve the real problem, so I can delete the compat library.

```
# make check-old-libs | grep '^/' | tar zcv -T - -f /root/old-libs.tgz
```

Once you have the libraries backed up, delete them all. You can use the `BATCH_DELETE_OLD_FILES` option here, but once again, I find yes(1) easier to type.

```
# yes | make delete-old-libs
```

If by some chance removing these libraries breaks pkg(8), run `pkg-static install -f pkg` to fix pkg(8) itself or `pkg-static upgrade -f` to reinstall all packages.

Another option is to use the libchk package to identify programs linked against old libraries.

mergemaster Revisited

We're almost there! While we already updated the passwords and group information in */etc*, we must update the rest of the files. mergemaster has many special functions, all documented in its man page. I'm going to

specifically recommend the two that I find notably useful.

When a file is added to the base FreeBSD install, there's no need to compare it to anything. The `-i` option makes mergemaster automatically install new files in */etc*. I'll get a list of automatically installed files at the end of the mergemaster run.

Another set of files that I don't really care about are files that I haven't edited. For example, FreeBSD has dozens of startup scripts in */etc/rc.d*. If I haven't edited a startup script, I just want to install the newest version of the script. The `-U` flag tells mergemaster to automatically update any base system file that I haven't edited.

```
# mergemaster -iU
```

The mergemaster program examines every file in */etc* and compares it to that in the base distribution of FreeBSD. This works exactly the same way as in your preinstallation mergemaster run, so we're not going to walk through it here. You should be familiar with the customizations you've made to your system, so nothing should surprise you. If anything looks unfamiliar, refer to Chapter 14.

Reboot, and your base system is fully upgraded!

Customizing Mergemaster

Once you've run mergemaster a few times you'll realize that certain files always annoy you. Mergemaster will always complain about your customized */etc/motd* and */etc/printcap*. You'll probably wind up typing `-F` or `-U` every single time. You can set your preferred options in *$HOME/.mergemasterrc*, as documented in mergemaster(8). While you should read the man page for the complete list, here are the options I use most often.

Ignoring Files

Certain files you don't want mergemaster to even bother examining. Your organization's */etc/motd* will never match that in the FreeBSD distribution. Neither will your custom printer configuration, your *snmpd.conf*, or your tailored *sshd_config*. To have mergemaster skip these files, list them in `IGNORE_FILES`.

```
IGNORE_FILES='/etc/motd /etc/printcap'
```

I don't list the password or group file here because sometimes FreeBSD introduces new users.

Auto Install New Files

If you want mergemaster to automatically install files present in the new version of FreeBSD but not on the system yet, set `AUTO_INSTALL`.

```
AUTO_INSTALL=yes
```

This is equivalent to the `-i` flag.

Autoupdate Unchanged Files

Different versions of FreeBSD have similar files. Some files are almost identical, differing only by the line containing version control information. You can tell mergemaster to automatically update files that differ only by the version control information by using the `FREEBSD_ID` option.

```
FREEBSD_ID=yes
```

This is the same as the `-F` flag.

Autoupdate Unedited Files

You can tell mergemaster to update files that haven't been edited since they were installed. The FreeBSD team changes */etc/* files when it wants to change how the system behaves. While many of those changes might be irrelevant to you, a few might give you trouble. If you want to blindly update everything you haven't touched, set `AUTO_UPGRADE`.

```
AUTO_UPGRADE=yes
```

This is equivalent to the `-U` option.

Update Databases

FreeBSD builds databases from */etc/master.passwd*, */etc/services*, and so on. If you update these files, you also need to update the corresponding databases. Mergemaster asks you at the end of each run if you want to run these

updates. Tell mergemaster always to run the updates by setting `RUN_UPDATES`.

```
RUN_UPDATES=yes
```

You can find other options in mergemaster(8).

Upgrades and Single-User Mode

According to the instructions, several parts of the upgrade must be done in single-user mode. Many users consider this an annoyance or even a handicap. FreeBSD programs are just files on disk, aren't they? Common sense says that you can just copy them to the disk, reboot, and be done with it.

Here's yet another instance where your common sense is trying to ruin your month. On rare occasions, the FreeBSD team needs to make some low-level changes in the system that require running the install in single-user mode. You can have conflicts where vital programs won't run when installed in multiuser mode. This is rare, but if it happens with */bin/sh*, you're in a world of hurt. You have a very straightforward recovery route if that happens: remove the hard drive from the server, mount it in another machine, boot the other machine, and copy your data off the destroyed system before formatting and reinstalling. Or, you can boot from the installation media and demonstrate your amazing sysadmin skills.[5]

Running in multiuser mode can cause other upgrade problems, such as subtle races, symbol issues, and innumerable other headaches. You can choose to upgrade in multiuser mode, but don't complain if your system has a problem.

It's perfectly safe to build your new world in multiuser mode. You can even build and install your new kernel in multiuser mode. Once you start installing the userland, however, you *must* be in single-user mode and running on your upgraded kernel.

NFS AND UPGRADES

Have a lot of machines to update? Look at NFS, which we discussed in Chapter 13. Build world and all your kernels on a

central, fast machine, and then export */usr/src* and */usr/obj* from that system to your other clients. Running `make installkernel` and `make installworld` from those NFS exports saves build time on all your other machines and guarantees that you have the same binaries on all your FreeBSD boxes.

Shrinking FreeBSD

What's the point of having all this source code if you can't customize your operating system? FreeBSD not only gives you the source code; it provides ready-to-turn knobs to easily customize your FreeBSD build.

These options can be set in either */etc/make.conf* (see Chapter 16) or */etc/src.conf*. Settings in *src.conf* apply only to building the FreeBSD source, while *make.conf* 's settings apply to all source code building. The full list of *src.conf* options are documented in src.conf(5), but they all follow a standard pattern.

Each of these options starts with either `WITHOUT_` or, in a few cases, `WITH_` and then names a specific subsystem. For example, the `WITHOUT_BHYVE` option turns off building or installing the bhyve(8) hypervisor. The `WITHOUT_INETD` option turns off building the inetd(8) daemon (see Chapter 20). The `WITHOUT_INET6` option turns off IPv6. If you want to shrink your FreeBSD install, chop out everything you don't need.

The build system checks to see whether any of these variables are defined to any value at all. This means that even if you set one of these to `NO`, the mere presence of the option activates the option. Don't go copying all of these to *src.conf* and setting them all to `NO` because you'll disable building a great big bunch of the system.

In most cases, adding these `WITHOUT_` options includes the removed systems in the `make delete-old` checks. If you decide that your system doesn't need bhyve(8), for example, the upgrade not only doesn't build a new bhyve binary but also offers to remove the existing one from the installed system. If you're not building a piece of software, you're better off removing it entirely as opposed to leaving the old version lingering on the system.

Packages and System Upgrades

Operating system upgrades are great, except for the part where nobody cares.[6] Base operating system updates are necessary, but most people don't really care about using the base system. They care about using software that runs on the base system. And that software is prone to the same flaws as every other program. You need to upgrade it. Chapter 15 discusses upgrading packages in general, but let's talk about what happens when you upgrade the operating system underneath the packages.

The common issues with packages and system upgrades normally boil down to shared libraries. If you're upgrading FreeBSD major releases—say, from FreeBSD 12 to FreeBSD 13—you'll need to reinstall all of your packages.

Start by upgrading pkg(8) itself, using the `-f` flag to `pkg upgrade`. If the upgrade broke pkg(8) itself, you'll need to use pkg-static(8). This contains key pkg(8) functions, but it's a static binary.

```
# pkg-static upgrade -f pkg
```

This will bootstrap you into the current package tools. Now you can force a redownload and reinstall of all packages.

```
# pkg upgrade -f
```

Once you've upgraded everything you built from packages, rebuild anything you built from ports. I really hope you installed your ports via poudriere, though.

Updating Installed Ports

If you use portsnap to update your ports tree, anything you install from now on will be the latest version. But what about your previously installed applications? FreeBSD tracks all sorts of dependency information between add-on packages, and often updating one program will impact dozens of others. This is a royal pain to manage. Wouldn't it be nice to just say, "Update my Apache install," and have FreeBSD manage the dependencies for you? There's a few ways to solve this issue.

The best way is not to use the ports tree on a production host. Build a private repository with poudriere instead (see Chapter 16), and have all of your hosts pull from that. This is a change from traditional FreeBSD sysadmin practice.

Maybe you have the ports tree installed locally and use one or two custom ports atop a bunch of packages. If you have a single port installed, rebuild it, uninstall the package, and install the new port. This is terrible for complicated ports with many dependencies but works fine for hosts with one or two ports.

Some of us have been around a long time, though, and feel like we live between those solutions. Our hosts feel too small to run poudriere, but we need custom ports. FreeBSD doesn't include an official tool for updating a system managed largely by ports, but people have written add-on tools, such as portupgrade and portmaster. The problem with these tools is that they're maintained outside of FreeBSD. If they can't upgrade a port but the normal build process works, the tool users are responsible for fixing the problems. That's true of all parts of FreeBSD, but the base system has a wider base of users than any add-on tool.

As I write this, though, FreeBSD's ports infrastructure is changing to support multiple versions of a single package. These tools haven't been updated to accommodate the new infrastructure. I expect one of them will be, but I can't say which that will be. Chapter 16 recommended using only packages in production. This is why.

Now that you can update your system and installed software, let's look at some of FreeBSD's more interesting security features.

19
ADVANCED SECURITY FEATURES

FreeBSD includes a variety of tools for securing network traffic and users. Some of these tools are invisible to sysadmins but work behind the scenes to increase security, such as the sandboxing API capsicum(4). Packet filtering lets you control who can access your system. You can also use blacklisting to block network addresses that keep poking at your host. In addition, FreeBSD has a whole bunch of optional security features you can enable either in the installation process or later. In this chapter, we'll examine these tools and techniques, look at monitoring your system's security, and discuss how to react if you suffer an intrusion.

Let's start with a core security topic: unprivileged users.

Unprivileged Users

An *unprivileged user* is a specific user for a specific task. He has only the rights necessary to perform that limited task. Many programs run as unprivileged users or use unprivileged users to perform specific duties.

"Only the rights needed to perform its duties" sounds like every user account, doesn't it? That's true, but the account used by the least privileged human being still has more rights than many programs need. Anyone with shell access has a home directory. The normal user may create files in their home directory, run text editors, or process email. Your average shell user needs these minimal privileges, but programs do not. By having a program, particularly a network daemon, run as a very restricted user, you control the amount of damage an intruder can do to either the program or the user.

FreeBSD includes several unprivileged users. Take a look at */etc/passwd* and you'll see accounts like *audit*, *bind*, *uucp*, and *www*. These are all unprivileged accounts for use by specific server daemons. See what they have in common.

Unprivileged users don't have normal home directories. Many have a home directory of */nonexistent*, while others, such as *sshd*, have a special home directory such as */var/empty*. Having a home directory where you may not write or read files makes the account less flexible but good enough for a server daemon. These users do own files on the system, but they usually can't write to those files.

Similarly, nobody should ever log into these accounts. If the account *bind* is reserved for the DNS system, nobody should actually log into the system as that user! Such an account must have a user shell that specifically denies logging in, like */usr/sbin/nologin*. How does all this enhance system security? Let's look at an example.

Whatever web server you're using, it generally runs under the unprivileged account *www*. Suppose that an intruder discovered a security flaw in the version of the web server program you're using and can make the web server execute arbitrary code. This is among the worst types of security problems, where an intruder can make the server program do absolutely anything within its power. What *is* within this program's power?

The intruder probably wants a command prompt on the system. A command prompt on a Unix-like system is the door to so much more access, after all. The unprivileged user has an assigned shell that specifically disallows logins. This really annoys intruders and requires them to work much harder to reach that command prompt.

If she's really clever, though, the nologin shell won't stop the intruder. Let's assume that through clever trickery she makes the web server execute a simple shell, such as */bin/sh*, and offer her the prompt. She's in and can wreak untold damage . . . or can she?

She has no home directory and doesn't have permissions to create one. That means that any files she wants to store must go in a globally accessible directory, such as */tmp* or */var/tmp*, increasing her visibility. The Apache configuration file is owned by root or by your web server administration group, and the *www* user isn't part of that group. The intruder might have a path into the web server, but she can't reconfigure it. She can't change the

website files, as the *www* user doesn't own them. The *www* user doesn't have access to anything on the system except the web server itself. A sufficiently skilled intruder can make the web server serve up different pages or redirect to another site, at least until a reboot. Penetrating the application running on the server, or the host itself, requires another whole set of security breaches.

An unprivileged user doesn't solve all security problems, mind you. Our compromised *www* user can view web application source files. If your application is badly written or has database passwords hardcoded into hidden files, you're still in a lot of trouble. Still, if you've kept your system updated and all your packages up-to-date, an intruder will have a very hard time penetrating FreeBSD itself.

The nobody Account

For years, system administrators used the account *nobody* as a generic unprivileged user. They'd run web servers, proxy servers, and whatever else as *nobody*. This was better than running those programs as root, but not as good as having separate users for each daemon. If an intruder successfully penetrated one of these programs, he had access to them all. Our hypothetical web server intruder would abruptly have access not only to the web server but also to whatever other programs run as that same user! If you're using NFS, remember that NFS defaults to mapping remote root accounts to *nobody*. The whole point of using unprivileged users is to minimize the possible damage from a successful intrusion.

While you might test with the *nobody* account, never deploy production services with it. Use separate unprivileged accounts liberally.

A Sample Unprivileged User

Here are parameters useful for a generic unprivileged user:

Username Assign a username related to the user's function. For example, the default user for web servers is *www*.

Home directory Unprivileged users should deliberately not have a home directory, so use */nonexistent*. Do not create a */nonexistent* directory either; the whole point is that it doesn't exist!

Shell Unpriviliged users must not have a shell that can execute commands, so use */usr/sbin/nologin*.

UID/GID Choose a special range of user and group IDs for unprivileged users.

Full name Assign a name describing the user's function.

Password Use chpass(1) to assign the user a single asterisk as their encrypted password. This disables the account password. (Note that chpass(1) stands for *change password file*, not *change password*!)

These settings make your unprivileged user very unprivileged indeed. You can set all of this easily with adduser(8), giving the account no password, the correct home directory, and an appropriate shell.

Many ports and packages have assigned unprivileged users and groups, listed in */usr/ports/UIDs* and */usr/ports/GIDs*. Don't be afraid to add more. Use UIDs above 1,000, so as not to conflict with those assigned by packages and FreeBSD's core.

Network Traffic Control

Sysadminis must have the ability to control traffic to and from their systems. Unwanted visitors must be stopped while legitimate users get access. FreeBSD provides a variety of tools that allow you to control outside access to your systems, including TCP wrappers, packet filtering, and blacklisting.

The TCP wrappers, or simply *wrappers*, control access to network daemons. While the program must be written to support TCP wrappers, most modern software has supported wrappers for many years. Wrappers are fairly simple to configure and don't require much networking knowledge. As access controls go, however, wrappers are fairly limited. Wrappers do let you do interesting things with connections and with daemons offering connections, though, which is why we'll discuss it.

Packet filtering controls which traffic the system allows to pass through it and which traffic it rejects. Most firewalls are packet filters with a pretty GUI on top, but you can use FreeBSD packet filtering and proxy software to build a solid firewall in and of itself. A rejected connection request never reaches any userland program; it's blocked in the network stack. Packet filtering can control access to any program, service, or network port but does

require more networking knowledge.

Blacklisting is useful when you want a program to be able to decide to stop listening to a remote host. The most common tool for blacklisting is fail2ban (*https://www.fail2ban.org/*), which is flexible but requires much special configuration. FreeBSD includes blacklistd, an easier-to-configure blacklisting tool that requires integration with programs that use it.

Which should you use? For basic TCP/IP access control, I recommend always using a packet filter. Only use TCP wrappers if you need their specific features. I discuss blocking and allowing connections with TCP wrappers only as a prerequisite to those advanced features. If you want a service to block clients after a certain number of failed connection attempts, consider blacklistd.

With wrappers or packet filtering, you must decide whether you want a default accept or default deny traffic control policy.

Default Accept vs. Default Deny

One of the essential decisions in any security policy is between default accept and default deny. A *default accept* security stance means that you allow any type of connection except what you specifically disallow. A *default deny* stance means that you allow connections only from specified parts of the internet and/or to specified services and that you refuse all other connections. The default is used unless you make a specific rule dictating otherwise. Once you've chosen your default security stance, you create exceptions one way or another to either provide or block services as necessary. The choice is really between whether you offer services to the world (default accept) or only to a select few (default deny).

For example, company policy might dictate that the intranet web server must be accessible only from within the company. If so, adopt a default deny stance and explicitly list who may access the server. Alternatively, if you have a public website but want to block certain parts of the internet from accessing it for whatever reason, adopt a default accept stance.

I always recommend a default deny stance. If you don't make a choice, however, you've chosen default accept.

Choosing a default doesn't mean that the default must be implemented without exceptions. My public web servers have a default deny security

stance, but I specifically allow the world to access the websites. The machine rejects attempts to connect to other programs unless they come from one of a few specified IP addresses. This is a perfectly acceptable default deny stance.

Different security tools implement these stances in different ways. For example, with TCP wrappers, the *first* matching rule is applied. If your last rule denies everything, you've established a policy that says, "Unless I've specifically created a rule earlier to permit this traffic, block it." On the other hand, with the PF packet filter, the *last* matching rule applies. If your first rule says, "Block all traffic," you've implemented a policy that says, "Unless I specifically create a later rule to permit this traffic, block it."

Both default accept and default deny annoy the sysadmin. If you have a default accept policy, you'll spend your time continually plugging holes. If you choose a default deny policy, you'll spend your time opening access for people. You'll repeatedly apologize for either choice. With default deny, you'll say things like, "I've just activated service for you. I apologize for the inconvenience." With default accept, you'll say things like, ". . . and that's why the intruders were able to access our internal accounting database and why we lost millions of dollars." In the latter case, "I apologize for the inconvenience" *really* doesn't suffice.

TCP Wrappers

Remember from Chapter 7 that network connections are made to various programs that listen for connection requests. When a program is built with TCP wrappers support, the program checks the incoming request against the wrappers configuration. If the wrappers configuration says to reject the connection, the program immediately drops the request. Despite the name, TCP wrappers work with both TCP and UDP connections. Wrappers are a long-running Unix standard that have been incorporated into FreeBSD. Individual programs might or might not work with wrappers; while just about everything in the base FreeBSD system does, some third-party software doesn't.

TCP wrappers are implemented as a shared library, called *libwrap*. As seen in Chapter 17, shared libraries are small chunks of code that can be shared between programs. Any program that links with libwrap may use the

TCP wrappers functions.

Wrappers most commonly protect inetd(8), the super server that handles network requests for smaller programs. We'll discuss inetd in Chapter 20. While our examples cover inetd(8), you can protect any other program that supports wrappers in exactly the same way. While wrappers help protect inetd(8), make sure inetd(8) doesn't offer any unnecessary services, just as you do for the main system.

Configuring Wrappers

Wrappers check each incoming connection request against the rules in */etc/hosts.allow*, in order. The first matching rule is applied, and processing stops immediately. This makes rule order very important. Each rule is on a separate line and is made up of three parts separated by colons: a daemon name, a client list, and a list of options. Here's a sample rule:

```
ftpd : all : deny
```

The daemon name is `ftpd`; the client list is `all`, meaning all hosts; and the option is `deny`, telling wrappers to reject all connections. Nobody can connect to the FTP server on this host unless an earlier rule explicitly grants access.

In the early examples, I refer to only two options: `accept` and `deny`. They allow and reject connections, respectively. We'll discuss the additional options later.

Daemon Name

The daemon name is the program's name as it appears on the command line. For example, inetd(8) starts the ftpd(8) program when it receives an incoming FTP request. The Apache web server starts a program called *httpd*, so if your version of Apache supports wrappers, give the daemon name as `httpd`. (Note that Apache doesn't run out of inetd, but it can support wrappers anyway.) One special daemon name, `ALL`, matches all daemons that support wrappers.

If your system has multiple IP addresses, you can specify, as part of the daemon name, different wrapper rules for each IP address that a daemon listens on:

```
ftpd@203.0.113.1 : ALL : deny
```

```
ftpd@203.0.113.1 : ALL : deny
ftpd@203.0.113.2 : ALL : accept
```

In this example, we have two daemon names, `ftpd@203.0.113.1` and `ftpd@203.0.113.2`. Each has a separate TCP wrapper rule.

The Client List

The client list is a list of specific IP addresses, network address blocks, hostnames, domain names, and keywords, separated by spaces. Hostnames and IP addresses are simple: just list them.

```
ALL : netmanager.absolutefreebsd.com 203.0.113.5 : allow
```

With this rule at the top of */etc/hosts.allow*, wrappers allow my netmanager machine and any host with an IP address of 203.0.113.5 to connect to any service on this host. (I could block this access by other means, mind you.)

Specify network numbers in the client list with a slash between the IP address and the netmask, as discussed in Chapter 7. For example, if script kiddies attack your server from a bunch of addresses that begin with 192.0.2, you could block them like this:

```
ALL : 192.0.2.0/255.255.255.0 : deny
```

You can also use domain names in client lists by prefacing them with a dot. This works through reverse DNS, which means that anyone who controls the DNS server for a block of addresses can evade this restriction.

```
ALL : .mycompany.com : allow
```

If you have a long list of clients, you can even list them in a file and put the full path to the file in the client space in */etc/hosts.allow*. I've been on networks with large numbers of widely scattered hosts, such as an ISP or corporate network environment with network management workstations scattered across the world. Each workstation shared the same wrapper rules as every other workstation and appeared on half a dozen lines in *hosts.allow*. By maintaining a single file with a workstation list, I could centralize all changes.

In addition to specifically listing client addresses and names, wrappers provide several special client keywords to add groups of clients to your list. Table 19-1 shows the keywords and their usage.

Most of the client keywords listed in Table 19-1 require a working DNS server. If you use these keywords, you must have a very reliable DNS service, and you must remember the vital link between DNS and the rest of your programs. If your DNS server fails, daemons that use wrappers and those keywords can't identify any hosts. This means that everything matches your `UNKNOWN` rule, which probably denies the connection. Also, broken DNS on the client end can deny remote users access to your servers, as your DNS servers won't be able to get proper information from the client's servers. Finally, if you use DNS-based wrapping extensively, an intruder needs only to overload your nameserver or otherwise interrupt your nameserver to create a very effective denial-of-service attack against your network.

Table 19-1: TCP Wrapper Keywords

Keyword	Usage
`ALL`	This matches every possible host.
`LOCAL`	This matches every machine whose hostname does not include a dot. Generally, this means machines in the local domain. Machines on the other side of the world who happen to share your domain name are considered "local" under this rule.
`UNKNOWN`	This matches machines with unidentifiable hostnames or usernames. As a general rule, any host making an IP connection has a known IP address. Tracing hostnames, however, requires DNS, and tracking usernames requires identd(8). Be very careful using this option because transitory DNS issues can make even local hostnames unresolvable and most hosts don't run identd(8) by default. You don't want a service to become unusable just because your nameserver was misconfigured—especially if that machine *is* your nameserver!
`KNOWN`	This matches any host with a determinable hostname and IP address. Be very careful using this, as DNS outages can interrupt service.
`PARANOID`	This matches any host whose name does not match its IP address. You might receive a connection from a host with an IP address of 192.168.84.3 that claims to be called *mail.michaelwlucas.com*. Wrappers turn around and check the IP address of *mail.michaelwlucas.com*. If wrappers get a different IP address, the

host matches this rule. Sysadmins who do not have time to maintain their DNS are the most likely to have unpatched, insecure systems.

TCP wrappers provide additional keywords, but they're not as useful or secure as these. For example, it's possible to allow connections based on the username on the remote machine. You don't want to rely on a client username on a remote machine, however. For example, if I set up wrappers to allow only someone with a username of *mwlucas* to connect to my home system, someone could easily add an account of that name to his FreeBSD system and get right in. Also, this relies on the same rarely used identd(1) protocol that was mentioned earlier. You can find a few other obscure keywords of similar usefulness in hosts_access(5).

The ALL and ALL EXCEPT Keywords

Both daemon names and client lists can use the ALL and ALL EXCEPT keywords. The ALL keyword matches absolutely everything. For example, the default *hosts.allow* starts with a rule that permits all connections, from all locations, to any daemon:

```
ALL : ALL : accept
```

This matches all programs and all clients. You can limit this by giving a specific name to either the client list or the daemon list.

```
ALL : 203.0.113.87 : deny
```

In this example, we reject all connections from the host 203.0.113.87.

Categorically blocking access to all hosts isn't that great an idea, but remember that TCP wrappers follow rules in order and quit when they reach the first matching rule. The ALL keyword lets you set a default stance quite easily. Consider the following ruleset:

```
ALL : 192.168.8.3 192.168.8.4 : accept
ftpd : ALL : accept
ALL : ALL : deny
```

Our workstations 192.168.8.3 and 192.168.8.4 (probably the sysadmin's

workstations) may access anything they want. Anyone in the world may access the FTP server. Finally, we drop all other connections. This is a useful default deny stance.

Use the `ALL EXCEPT` keyword to compress rules. `ALL EXCEPT` allows you to list hosts by exclusion; what isn't listed matches. Here, we write the same rules with `ALL EXCEPT`:

```
ALL : 192.168.8.3 192.168.8.4 : accept
ALL EXCEPT ftpd : ALL : deny
```

Of course, this rule relies on having a default accept policy that permits the FTP connection later.

Some people find rules more clear when written with `ALL`, others prefer `ALL EXCEPT`. The important thing to remember is that the first matching rule ends the check, so be careful slinging `ALL` around.

It's a good idea to allow any connections from the local host; you're likely to discover a number of programs that break when they can't talk to the local machine. Put a rule like this early in your *hosts.allow*:

```
ALL : localhost : allow
```

Options

We've already seen two options: `allow` and `deny`. While `allow` permits the connection, `deny` blocks it. The first rule in the default *hosts.allow* applies to all daemons and clients, and it matches and allows all possible connections. This rule can't be first in your *hosts.allow* if you want to wrap your services, but it's a good final rule in a default accept security stance. Similarly, an `ALL:ALL:deny` rule is a good final rule in a default deny security stance. TCP wrappers support other options besides the simple `allow` and `deny`, however, giving you a great deal of flexibility.

LONG RULES

If you're using a lot of options, wrapper rules can get very long. To help keep rules readable, the *hosts.allow* file can use the backslash (\) followed by a return as a line-continuation

character.

Logging

Once you've decided to accept or reject the connection attempt, you can also log the connection. Suppose you want to permit but specifically log all incoming requests from a competitor. Similarly, you might want to know how many connections your server rejects because of DNS problems when using the `PARANOID` client keyword. Logging is good. More logging is better. Disk space is cheaper than your time.

The `severity` option sends a message to the system log, syslogd(8). You can configure syslogd to direct these messages to an arbitrary file based on the syslogd facility and level you choose (see Chapter 21).

```
sshd : ALL : severity local0.info : allow
```

This example permits all SSH connections but also logs them using the local0 facility.

Twisting

The `twist` option allows you to run arbitrary shell commands and scripts when someone attempts to connect to a wrapped TCP daemon and returns the output to the remote user. The `twist` option works properly only with TCP connections. (Remember, UDP is connectionless; there's no connection to return the response over, so you must jump through very sophisticated and annoying hoops to make `twist` work with UDP. Also, protocols that transmit over UDP frequently don't expect such a response and aren't usually equipped to receive or interpret it. Using `twist` with UDP isn't worth the trouble.) The `twist` option takes a shell command as an argument and acts as a deny-plus-do-this rule. You must know basic shell scripting to use `twist`; very complicated uses of `twist` are possible, but we'll stick with the simple ones.

The `twist` option is useful for a final rule in a default deny stance. Use `twist` to return an answer to the person attempting to connect as follows:

```
ALL : ALL : twist /bin/echo "You cannot use this service."
```

If you want to deny just a particular service to a particular host, you can use more specific daemon and client lists with `twist`:

```
sendmail : .spammer.com : twist /bin/echo \
    "You cannot use this service, spam-boy."
```

This isn't effective against spam, but it might make you feel better. Legit customers that encounter rude messages might trigger meetings, however.

If you're feeling friendly, you can tell people why you're rejecting their connection. The following `twist` rejects all connections from people whose hostname doesn't match their IP address and tells them why:

```
ALL : PARANOID : twist /bin/echo \
    "Your DNS is broken. When you fix it, try again."
```

Using `twist` holds the network connection open until the shell command finishes. If your command takes a long time to finish, you could find that you're holding open more connections than you like. This can impact system performance. A script kiddie can use `twist` to overload your system, creating a very simple DoS attack. Make `twist` simple and quick-finishing.

Spawning

Like `twist`, the `spawn` option denies the connection and runs a specified shell command. Unlike `twist`, `spawn` doesn't return the results to the client. Use `spawn` when you want your FreeBSD system to take an action upon a connection request but you don't want the client to know about it. Spawned commands run in the background. The following example allows the connection but logs the client's IP address to a file:

```
ALL : PARANOID : spawn (/bin/echo %a >> /var/log/misconfigured ) \
    : allow
```

Wait a minute—where did the `%a` come from? TCP wrappers support several variables for use in `twist` and `spawn` commands, so you can easily customize your responses. This particular variable, `%a`, stands for *client address*. It expands into the client's IP address in the shell command before the command is run. Table 19-2 lists other variables.

Table 19-2: Variables for `twist` and `spawn` Scripts

Variable	Description
`%a`	Client address.
`%A`	Server IP address.
`%c`	All available client information.
`%d`	Name of the daemon connected to.
`%h`	Client hostname, or IP address if hostname not available.
`%H`	Server hostname, or IP address if hostname not available.
`%n`	Client hostname, or `UNKNOWN` if no hostname is found. If the hostname and the IP address don't match, this returns `PARANOID`.
`%N`	Server hostname, but if no hostname is found, this returns either `UNKNOWN` or `PARANOID`.
`%p`	Daemon's process ID.
`%s`	All available server information.
`%u`	Client's username.
`%%`	A single `%` character.

Use these variables anywhere you'd use the information they represent in a shell script. For example, to log all available client information to a file whenever anyone connects to a wrapped program, you could use this:

```
ALL : ALL : spawn (/bin/echo %c >> /var/log/clients) : allow
```

Spaces and backslashes are illegal characters in shell commands and might cause problems. While neither appears in hostnames under normal circumstances, the internet is almost by definition not normal. TCP wrappers replace any character that might confuse the command shell with an underscore (_). Check for underscores in your logs; they might indicate possible intrusion attempts or just someone who doesn't know what they're doing.

Wrapping Up Wrappers

Let's take all the examples given so far in this section and build a complete

/etc/hosts.allow to protect a hypothetical network system. We must first inventory the network resources this system offers, the IP addresses we have on the network, and the remote systems we wish to allow to connect.

While these requirements are fairly complicated, they boil down to a very simple ruleset:

```
#reject all connections from hosts with invalid DNS and from our competitor
ALL : PARANOID 198.51.100.0/24 : deny
#localhost can talk to itself
ALL : localhost : allow
#our local network may access portmap, but no others
portmap : ALL EXCEPT 203.0.113.0/24 : allow
#allow SSH, pop3, and ftp, deny everything else
sshd, POP3, ftpd : ALL : allow
ALL : ALL : deny
```

You can find many more commented-out examples in */etc/hosts.allow* or in hosts_allow(5) and hosts_access(5).

Packet Filtering

To control access to networked programs that don't support TCP wrappers, or whenever your needs exceed what wrappers provide, use one of FreeBSD's kernel-level packet filtering tools. If you need a packet filter, it's best to entirely replace your TCP wrappers implementation with packet filtering. Using both tools at once on the same machine will simply confuse you.

A packet filter compares every network packet that enters the system to a list of rules. When a rule matches the packet, the kernel acts based upon that rule. Rules can tell the system to allow, drop, or alter the packet. You can't use the nifty options provided by TCP wrappers, however; instead of spitting a comparatively friendly rejection message back at the client, the connection is severed at the network level before the client even reaches the application.

While the idea of packet filtering is straightforward enough, your first implementation will be a complete nightmare—er, I mean, a "valuable learning experience." Be prepared to spend hours experimenting and don't be discouraged by failures. In my experience, it's ignorance of basic TCP/IP that causes grief with packet filtering, rather than the packet filter itself.

Trying to filter network traffic without understanding the network is frustrating and pointless. The only way to really understand TCP/IP is to do real work with it, however. Go study Chapter 7 again. If that doesn't suffice, dig into the books recommended there.

FreeBSD suffers from a wealth of packet filters: IPFW, IP Filter, and PF.

IPFW is the primordial FreeBSD packet filtering software. It's tightly integrated with FreeBSD; in fact, the generically named files */etc/rc.firewall* and */etc/rc.firewall6* are purely for IPFW. While quite powerful and very popular with more experienced FreeBSD administrators, it's a little difficult for a beginner.

The second packet filter, IP Filter, is not a FreeBSD-specific firewall program but is supported on several Unix-like operating systems. It's primarily the work of one individual, Darren Reed, who has by heroic effort developed the overwhelming majority of the code and ported it to all those operating systems. IP Filter is most useful if you want to share one firewall configuration among multiple operating systems.

We'll focus on the imaginatively named *PF*, or *packet filter*. PF originated in OpenBSD and was designed to be featureful, flexible, and easy to use. The average FreeBSD administrator can use PF to achieve almost any effect possible with the other two packet filters.

NOTE

For in-depth discussion of PF, you might check out Peter N. M. Hansteen's The Book of PF *(No Starch Press, 2014) or my book* Absolute OpenBSD *(No Starch Press, 2013), which contains several chapters about PF. You might also look at the online PF FAQ, but that has fewer haiku.*

Enabling PF

PF includes the packet filtering kernel module, *pf.ko*, and the userland program pfctl(8). Before using PF, you must load the kernel module. The simplest way is to enable PF in *rc.conf*:

```
pf_enable="YES"
```

PF defaults to the accept all stance, which means that you won't lock yourself out of your server merely by enabling the firewall.

Default Accept and Default Deny in Packet Filtering

The security stances (default accept and default deny) are critical in packet filtering. If you use the default accept stance and want to protect your system or network, you need numerous rules to block every possible attack. If you use the default deny stance, you must explicitly open holes for every little service you offer. In almost all cases, default deny is preferable; while it can be more difficult to manage, its increased security more than makes up for that difficulty.

When using a default deny stance, it's very easy to lock yourself out of remotely accessing your machine. When you have an SSH connection to a remote machine and accidentally break the rule that allows SSH access, you're in trouble. Everybody does this at least once, so don't be too embarrassed when it happens to you. The point is, it's best not to learn about packet filtering on a remote machine; start with a machine that you can console into so you can recover easily. I've cut my own access many times, generally because I'm not thinking straight when solving an unrelated packet filtering problem. Without a remote console or IPMI, the only fix is to kick myself as I climb into the car, drive to the remote location, and apologize profusely to the people I've inconvenienced as I fix the problem. Fortunately, as I grow older, this happens less and less.[1]

Still, in almost all circumstances, a default deny stance is correct. As a new administrator, the only way you can reasonably learn packet filtering is if you have convenient access to the system console. If you're not entirely confident in your configuration, don't set up a packet filtering system across the country unless you have remote console and power access, a competent local administrator, or a serial console.

Basic Packet Filtering and Stateful Inspection

Recall from Chapter 7 that a TCP connection can be in a variety of states, such as opening, open, closing, and so on. For example, every connection opens when the client sends a SYN packet to the server to request connection synchronization. If the server is listening on the requested port,

it responds with a SYN-ACK, meaning, "I've received your request, and here's basic information for our connection." The client acknowledges receipt of the information with an ACK packet, meaning, "I acknowledge receipt of the connection information." Each part of this three-way handshake must complete for a connection to occur. Your packet filtering ruleset must permit all parts of the handshake, as well as the actual data transmission, to occur. Allowing your server to receive incoming connection requests is useless if your packet filter rules don't permit transmitting that SYN-ACK.

In the early 1990s, packet filters checked each packet individually. If a packet matched a rule, it was allowed to pass. The system didn't record what it had previously passed and had no idea whether a packet was part of a legitimate transaction or not. For example, if a packet arrived marked SYN-ACK with a destination address inside the packet filter, the packet filter generally decided that the packet had to be the response to a packet it had previously approved. Such a packet *had* to be approved to complete the three-way handshake. As a result, intruders forged SYN-ACK packets and used them to circumvent seemingly secure devices. Since the packet filter didn't know who had previously sent a SYN packet, it couldn't reject illegitimate SYN-ACK packets. Once an intruder gets packets inside a network, he can usually trigger a response from a random device and start to worm his way in.

Modern packet filters use stateful inspection to counteract this problem. *Stateful inspection* means keeping track of every connection and its current condition. If an incoming SYN-ACK packet appears to be part of an ongoing connection, but nobody sent a corresponding SYN request, the packet is rejected. While this complicates the kernel, writing stateful inspection packet filter rules is easier than writing old-fashioned rules. The packet filter must track many, many possible states, so this is harder to program than it might seem—especially when you add in problems such as packet fragmentation, antispoofing, and so on.

PF performs stateful inspection by default. You don't need to specify it in a rule.

If you've started to think, "Hey, packet filtering sounds like a firewall," you're right, to a point. The word *firewall* is applied to a variety of network protection devices. Some of these devices are very sophisticated; some lose

intelligence contests to cinderblocks. These days, the term *firewall* is nothing more than a marketing buzzword with very little concrete meeting. The word *firewall* is like the word *car*: do you mean a rusty 1972 Gremlin with a 6-horsepower engine and an exhaust system that emits enough fumes to breach the Kyoto Accords, or a shiny Tesla Roadster with a 500-horsepower engine, a fancy tricolor paintjob, and the Stereo System of The Apocalypse? Both have their uses, but one is obviously designed for performance. While the Gremlins of firewalls might have their place, it's preferable to get the best you can afford.

Having said that, FreeBSD can be made as solid a firewall as you desire. Packet filtering is only the beginning. The packages collection contains a variety of application proxies that can let your FreeBSD system go up against Checkpoint or a PIX and come out on top—for tens of thousands of dollars less.

Configuring PF

Configure PF in */etc/pf.conf*. This file contains statements and rules whose formats vary with the features they configure. Not only is the rule order extremely important but also the order in which features are configured. If you try to do stateful inspection before you reassemble fragmented packets, for example, connections won't work properly.

The default */etc/pf.conf* has the sample rules in the proper order, but if you're in the slightest danger of becoming confused, I suggest that you put large comment markers between the sections, in capital letters if necessary. (Use hash marks to comment *pf.conf*.) The features must be entered in this exact order:

1. Macros
2. Tables
3. Options
4. Packet normalization
5. Bandwidth management
6. Translation
7. Redirection
8. Packet filtering

Yes, PF does more than just filter packets. It's a general-purpose TCP/IP manipulation tool. We won't cover all of its features here; go read Peter's book.

Macros

A macro lets you define variables to make writing and reading rules easier. For example, here are macros to define your network interface and your IP address:

```
interface="em0"
serveraddr="203.0.113.2"
```

Later in your rules, you may describe your network interface as `$interface` and your server's IP address as `$serveraddr`. This means that if you renumber your server or change your network card, making one change in your *pf.conf* fully updates your rules.

Sometimes you'll want a rule to refer to "all IP addresses currently on this interface." You don't care which address the traffic arrives at, you just want either to accept or reject traffic to that interface. PF provides shorthand for this. Enclose the interface name in parentheses, as we'll see later. (You can use the interface name without parentheses, but then PF won't notice any IP changes since the last reload or restart.)

Tables and Options

PF can store long lists of addresses through tables. That's a more sophisticated use of PF than we're going to use, but you should know the capability exists.

Similarly, PF has a variety of options that control network connection timing, table sizes, and other internal settings. The default settings are generally adequate for normal (and most abnormal) use.

Packet Normalization

TCP/IP packets can be broken up in transit, and processing these shards of data increases system load and the amount of work your server must do both to serve the request and filter the packets. A system must reassemble these fragments before handing them on to your client software, while deciding what to do with any other random crud that arrives. PF refers to this

reassembly as *scrubbing*. For example, to reassemble all fragments coming in your network interface, drop all fragments too small to possibly be legitimate, and otherwise sensibly sanitize your incoming data stream, use the following rule:

```
scrub in
```

This affects all packets entering the computer.

While scrubbing seems like a "nice to have," it's actually quite important since PF filters are based on whole packets. Fragments are much more difficult to filter and require special handling unless reassembled. Not scrubbing your traffic causes connectivity problems.

Bandwidth, Translation, and Redirection

PF includes other features vital for firewalls and performs other functions normally associated with network devices. Through queueing, PF can control how much traffic the host transmits on a per-IP or even per-port basis. PF includes a whole bunch of features to support Network Address Translation (NAT) and port redirection, two critical firewall features. The support exceeds that found in many commercial offerings.

All of this would fill another book. Literally. Peter Hansteen wrote *The Book of PF*. Go read that and build a firewall. Every sysadmin should build a firewall out of a raw operating system at least once in her life. Even if you revert to using a commercial offering, a little embedded device, or a product like pfSense or OPNsense, you'll learn a whole bunch.[2]

Small-Server PF Rule Sample

Here's a sample set of PF rules for protecting a small internet server. Start from here and edit this to match your server's requirements.

```
❶ ext_if="em1"
❷ set skip on lo0
❸ scrub in

❹ block in
❺ pass out
```

```
❻ pass in on $ext_if proto tcp from any to ($ext_if) port {22, 53, 80, 443}
❼ pass in on $ext_if proto udp to ($ext_if) port 53
❽ pass in on $ext_if inet proto icmp to ($ext_if) icmp-type { unreach, redir,
      timex, echoreq }
```

We start by defining a macro for our interface name ❶ so that if we change network cards, we won't need to rewrite all our rules.

The second line instructs PF not to filter on the lo0 interface ❷. The loopback interface is local to the machine. The only host that can communicate over it is the local machine.

Then, we scrub incoming traffic ❸, reassembling packets into a coherent whole and throwing away what can't be reassembled.

Now that we have a sensible stream of incoming data, we can filter it. This policy starts by blocking all incoming traffic ❹, setting a default deny policy. Everything not explicitly permitted is forbidden.

Outbound traffic gets a default allow policy ❺.

The final three rules in this policy address TCP, UDP, and ICMP. They have a similar format, which we'll dissect shortly.

First, we permit TCP traffic to ports 22, 53, 80, and 443 ❻.

Next, we permit UDP traffic to port 53 ❼. If this host offered more services than DNS, we'd have a longer list of ports.

The final rule allows vital ICMP traffic to our host and permits the host to respond ❽.

Let's take a closer look at the TCP rule.

```
❶pass in❷on $ext_if❸proto tcp❹from any❺to ($ext_if)❻port {22, 53, 80, 443}
```

This host has a default deny policy on inbound traffic, so with the `pass in` statement ❶, we're carving out an exception to that policy.

The next chunk of the rule specifies which interface this rule applies to ❷. This rule applies to the interface defined by the macro `$ext_if`, or em1.

Then, we specify a protocol ❸. This rule applies to TCP connections.

You can write PF rules that apply only to specific source or destination addresses. This rule applies to traffic from any host ❹. You can drop this

part of the rule if you're permitting any source address.

We then specify a destination address ❺. The destination is the interface name in parentheses, which means "any IP address on this interface."

Lastly, define the ports this rule applies to ❻. The braces allow you to group several entities together. The filter permits connections to port 22 (ssh), 53 (DNS), 80 (HTTP), and 443 (HTTPS). You could specify a port by its name (from */etc/services*), but I find numbers to be more reliable. Editing */etc/services* shouldn't break your firewall! Deploying a new TCP service on this host requires only adding a port to the list and reloading the firewall rules.

The UDP rule is very slightly different.

```
pass in on $ext_if❶proto udp to ($ext_if)❷port 53
```

The most obvious change is defining UDP protocol instead of TCP ❶. One less obvious change is that this rule drops the source address. It applies to packets from any address. This packet filter allows only one port, 53 ❷. Rules with a single port don't need braces.

The ICMP rule looks a little tricky, but it's really just the same.

```
pass in on $ext_if inet❶proto icmp to ($ext_if)❷icmp-type❸{ unreach, redir, timex
echoreq }
```

Specifying that this rule applies to ICMP is straightforward ❶. And this rule also doesn't list a source address, so it applies to traffic from anywhere.

Where the TCP and UDP rules specify a destination port, this ICMP rule lists an `icmp-type` ❷. ICMP doesn't have ports, but it does have different types of traffic. For our purposes, though, ICMP types are much like ports. Types have numerical codes, but the names are easier.

This rule specifies four different types of ICMP traffic ❸.

Taken as a whole, this rule permits ICMP traffic that's generally necessary for proper internet functioning. Your environment might need other ICMP types. Your organization's security policy might specify what ICMP you can and cannot pass. But these four are a reasonable combination for an internet-facing server.

This simple policy defines basic rules for communicating with our server. While it's not perfect, it can raise barriers for intruders. That jerk who broke into your web server and started a command prompt on port 10000? If your firewall rules don't allow incoming connections on that port, all their hard work will be wasted. Such a tragedy.

Managing PF

Manage PF with pfctl(8). If your rules have no errors, pfctl(8) runs silently; it produces output only when you have errors. You'll want to test, activate, view, and remove rules.

Testing Rules

As a firewall error can cause you much grief, it's best to check your rules before activating them. While a rule check only parses the file, checking for grammatical errors in the rules themselves, activating rules with grammatical errors either leaves your system unprotected, locks you out, or both. Use the `-n` flag to check a file for problems and `-f` to specify the PF rules file.

```
# pfctl -nf /etc/pf.conf
```

If you get errors, fix them and try again.

Activating Rules

Once your syntax check runs silently, activate the new rules by removing the `-n` flag.

```
# pfctl -f /etc/pf.conf
```

Changing PF configuration is very quick. This means you can have several PF configurations for different times or situations. Perhaps you want to allow access only to certain services at certain parts of the day; you could schedule a pfctl(8) run to install appropriate rules for those times. Or maybe you have separate rules for disaster situations and want to install a special ruleset when you lose your internet connection. Using pfctl(8) makes all these configurations simple.

View Rules

If you want to see the rules currently running on your firewall, use `pfctl -sr`.

```
# pfctl -sr
scrub in all fragment reassemble
block drop in all
pass in on em1 proto tcp from any to (em1) port = ssh flags S/SA keep state
pass in on em1 proto tcp from any to (em1) port = domain flags S/SA keep state
pass in on em1 proto tcp from any to (em1) port = http flags S/SA keep state
pass in on em1 proto tcp from any to (em1) port = https flags S/SA keep state
pass in on em1 proto udp from any to (em1) port = domain keep state
pass in on em1 inet proto icmp from any to (em1) icmp-type unreach keep state
pass in on em1 inet proto icmp from any to (em1) icmp-type redir keep state
pass in on em1 inet proto icmp from any to (em1) icmp-type timex keep state
pass in on em1 inet proto icmp from any to (em1) icmp-type echoreq keep state
pass out all flags S/SA keep state
```

You can write PF rules in exactly the format shown here.

Note that while we specified multiple TCP ports in the configuration file, in the packet filter each TCP and UDP port gets its own rule. Likewise, each ICMP type gets its own rule.

Removing Rules

Finally, remove all rules from your running configuration with the `-Fa` (flush all) flags. (You could use flags other than `a` to remove parts of your firewall config, but that can leave your system in an inconsistent state.)

```
# pfctl -Fa
```

You'll see PF systematically erase all rules, NAT configurations, and anything else in your configuration. Do not manually clear the configuration before loading a new configuration; just load the new rules file to erase the old rules.

PF is terribly powerful, very flexible, and can abuse TCP/IP in almost any way you like (and some ways you won't like). We've barely scratched the surface. Check out some of the resources listed at the start of "Packet Filtering" on page 462 to explore PF in depth.

Blacklistd(8)

Sometimes you want more thoughtful packet filtering than a simple allow or deny permits. I often have SSH servers open to the public internet so that I

can log in from anywhere. I do rather resent botnets thinking that I'd be sufficiently daft to permit logins without a password, though. That's where blacklistd(8) comes in.

Blacklistd lets a daemon report, "Hey, this IP address is bugging me." Once blacklistd receives a sufficient number of complaints about an address, it tells the firewall to block that address. Those bots eternally poking at your SSH server? They're history.

This sort of blacklisting is only marginally useful against distributed botnets like the Hail Mary Cloud, but even then, you might be able to configure sensitivity to block out the most annoying clients. It all depends on just how intrusive each botnet member is.

To use blacklistd, you must set up the packet filter to accept input from blacklistd, set tolerance levels for each service, and configure the service to use blacklistd.

PF and Blacklistd

PF handles dynamic rules through anchors. You can use pfctl(8) to edit an active anchor, letting you insert rules at a specific point in the policy. Add the blacklistd anchor to your rules right before your first `block` and `pass` statements. Using the policy from the previous section, your rules would look like this:

```
--snip--
anchor "blacklistd/*" in on $ext_if
block in
pass out
--snip--
```

You must include the quotes around the anchor name, and you must specify the interface.

The packet filter is now ready for dynamic blacklisting.

Configuring Blacklistd

Blacklistd gets its configuration from */etc/blacklistd.conf*. While most of its configuration goes in this file, you can also modify the service's behavior with command line options.

Start by enabling blacklistd in */etc/rc.conf*.

```
# sysrc blacklistd_enable=YES
blacklistd_enable: NO -> YES
```

The daemon won't start until you either reboot or start it manually, so you can configure it now.

/etc/blacklistd.conf

Blacklistd rules each support a single service, port, or group of addresses. Put your rules into */etc/blacklistd.conf*, one rule per line. Blacklistd rules come in two groups, local and remote.

Local blacklistd rules apply to items local to the machine running blacklistd. This is where you set rules for the local SSH service, or port 99, or anything else local. The section of local rules is prefaced with `[local]`.

Remote blacklistd rules apply to items not local to the machine. Here, you might define rules like "this block gets reduced tolerance" or "disable these addresses for shorter times" or "never block these addresses." The section of remote rules gets prefaced with `[remote]`. We'll talk about local rules first and then the additions supported by remote rules.

Here's a sample *blacklistd.conf* entry:

```
[local]
ssh             stream  *       *               *       3       24h
```

The first line is a `[local]` statement. Every rule that appears after this applies to the local machine, until we hit a `[remote]` entry.

Each rule has seven fields. The first four fields identify traffic to be blacklisted, while the last three fields define the blacklist behavior. An asterisk (`*`) is a wildcard, saying anything matches this field.

The first field is the *location*. For local rules, this gives the network port that this rule applies to. Entries like `ssh` and `ftp` are slightly deceiving. They don't apply to the programs named `sshd` and `ftpd`, but rather to the network ports listed in */etc/services*. While you can list a specific IP address and port in local rules, blacklistd ignores the address. Only the port applies. The sample rule blocks on `ssh`, or port 22.

The second field gives the socket type. TCP sockets use type `stream`, while UDP sockets need `dgram`. At this time, all services that support blacklistd use

TCP. You can safely use an asterisk here to say "any socket type." Our sample rule uses stream, so it's for TCP connections.

The third field defines the *protocol*. Supported options include `tcp`, `udp`, `tcp6`, `udp6`, or `numeric`, or you can just use a wildcard and say "any protocol." The only reason not to use a wildcard here is if you want to specifically match only one version of IP, such as using a different blacklist setting for TCP over IPv4 than for TCP over IPv6.

The fourth field gives the *owner* of the daemon complaining about the traffic. This can be a wildcard, a username, or a UID. Again, wildcards are the most common entry here. For blacklisting purposes, I don't care which user runs the server running on port 22; I care that it gets protected from random poking.

The fifth field, the *packet filter rule name*, is the first entry that determines how the block works. Blacklistd defaults to putting all blocks under an anchor called *blacklistd*, which we put into *pf.conf* in the previous section. If you want separate blacklists to use different anchors, you can define an anchor name in this field; otherwise, just use the wildcard for the default.

If you start a name with a hyphen (-), it means "use an anchor with the default name prepended."

```
ssh             stream  *       *               -ssh       3       24h
```

This entry adds any new blacklist rules to an anchor called *blacklistd-ssh*.

Using a slash (/) in the name field and the length of the netmask tells blacklistd to block entire subnets using prefix notation.

```
22              stream  tcp      *               */24    3       24h
```

When one host in a network misbehaves, we block everything in the adjoining /24. A /24 means very different things in IPv4 versus IPv6. Be sure to specify which protocol this rule applies to!

The sixth column, *nfail*, sets the number of login failures needed to blacklist the remote IP. Here, a wildcard means never. Our example rule sets a limit of 3, which is how many chances OpenSSH gives you to log in on one connection.

The last column, *disable*, says how long to blacklist the host for. The default unit is seconds, but you can use `m`, `h`, and `d` for minutes, hours, and

days, respectively. Our example rule is set to 24 hours.

So, with this rule in place, failing to authenticate to SSH three times will result in the client being blocked for 24 hours.

Once you have local rules set up, you can configure remote rules.

blacklistd.conf Remote Rules

Use remote rules to specify how blacklistd varies its behavior depending on the remote host. Each of the fields in a remote rule is the same as that in the local rules, but how blacklistd uses them changes. Here's a sample remote rule:

```
[remote]
203.0.113.128/25 *      *       *               =/25    =       48h
```

The *address* column is an IP (either IPv4 or IPv6) address, a port, or both. This lets you set special rules for a specific remote address range. Our sample rule applies to the address range 203.0.113.128/25.

The *type*, *protocol*, and *owner* columns are interpreted identical to the local rules.

The *name* column gets interesting. The equal sign in a remote rule means "use the value from the local rule you're matching." This rule says to take the firewall rule name entry and add the network prefix /25 (a 255.255.255.128 netmask) to it. If a connection from this address range gets blacklisted, it will affect the entire subnet. If you put a PF anchor name here, the blacklistd adds rules for this address block to the named anchor. A wildcard reverts to the default table.

The *nfail* column lets you set a custom number of failures for this address. Maybe you want to offer that one customer that just can't figure out how to type their password the first 30 times extra attempts to fail. Setting this column to an asterisk disables blocking.

The *disable* column lets you set a custom block time for this address block. Using a wildcard here disables blocking.

Remote rules let you enforce stricter limits on people you don't like, while telling blacklistd(8) never to blacklist your office.

You can now start blacklistd. It won't do anything, though, because programs don't know they should complain to it. But once you configure

them, it'll be ready.

Configuring Blacklistd Clients

FreeBSD includes a few blacklistd-aware clients. The two you're most likely to use are ftpd(8) and sshd(8).

To enable blacklistd in your SSH server, add the following line to */etc/ssh/sshd_config*.

```
UseBlacklist yes
```

Restart sshd.

Enable blacklisting in ftpd(8) with the `-B` command line option, either in */etc/inetd.conf* or in the standalone process's */etc/rc.conf* flags.

```
ftpd_flags="-B"
```

These programs will now whinge to blacklistd(8) any time someone fails to log in.

Managing Blacklistd

Blacklisting annoying clients that have no right to poke at your services cuts down on the amount of log analysis you need to do, but you'll probably want to see exactly what the blacklist is blocking. You want blacklistctl(8).

The blacklistctl(8) program has only one function: to display addresses and networks blocked by blacklistd. You always want the `blacklistctl dump` command.

By default, `blacklistctl dump` shows hosts that are in the list of candidates to be blocked but are not yet blocked. Add the `-b` flag to see all blocked hosts.

```
# blacklistctl dump -b
        address/ma:port id      nfail   last access
  203.0.113.128/25:22   OK      6/3     2018/08/28 16:30:09
```

Here, we see that the address range 203.0.113.128/25 attempted 6 out of 3 permitted login attempts. How did it achieve this? SSH lets a client try multiple logins on a single TCP/IP connection. Blacklisting doesn't stop a live connection. The last time the guilty host attempted to access this service

was at the date shown in `last access`.

You might find the time remaining more useful than the time of last access. Add the `-r` flag.

```
# blacklistctl dump -br
        address/ma:port id      nfail   remaining time
  203.0.113.128/25:22   OK      4/3     36s
```

Too soon, this subnet will be free to harass and harry my innocent SSH server. Maybe I need to increase the blacklist duration.

De-Blacklisting

Despite your best efforts, one day you'll need to pull an address from the blacklist before it expires naturally. The blacklistctl(8) program offers no way to do this: you must manually delete the address from the PF table. Doing so requires understanding how blacklistd manages addresses inside PF.

Each blocked port has a child anchor inside the blacklistd anchor. This anchor is named after the port. The child anchor that blocks port 22 would be called *blacklistd/22*. Inside that child anchor, you'll find a table containing the blocked addresses. The table is named *port*, followed by the port number. Hosts that can no longer connect to port 22 appear in a table called *port22*.

Here, I use the packet filter control program pfctl(8) to examine the contents of the port22 table inside child anchor blacklistd/22. I'm not going to explain all of this; just substitute your table and child anchor names. (Read Hansteen's *The Book of PF* to let anchors drag you under. Far, far under.)

```
# pfctl -a blacklistd/22 -t port22 -T show
--snip--
   203.0.113.128/25
--snip--
```

Yes, our problem address is in there. Removing it requires a fairly arcane pfctl(8) command.

```
# pfctl -a blacklistd/22 -t port22 -T delete 203.0.113.128/25
```

The blacklist is maintained in a database outside of PF, though, so the blacklisted address will still show up in blacklistctl(8). That database entry will eventually expire harmlessly. If the host misbehaves again, it will get

blocked again.

Public-Key Encryption

Many server daemons rely upon public-key encryption to ensure confidentiality, integrity, and authenticity of communications. Many different internet services also use public-key encryption. You need a basic grasp of public-key encryption to run services like secure websites (https) and secure POP3 mail (pop3ssl). If you're already familiar with public-key encryption, you can probably skip this section. If not, gird your loins for a highly compressed introduction to the topic.

Encryption systems use a key to transform messages between readable (cleartext) and encoded (ciphertext) versions. Although the words *cleartext* and *ciphertext* include the word *text*, they aren't restricted to text; they can also include graphics files, binaries, and any other data you might want to send.

All cryptosystems have three main purposes: integrity, confidentiality, and nonrepudiation. *Integrity* means that the message hasn't been tampered with. *Confidentiality* means that the message can be read only by the intended audience. And *nonrepudiation* means that the author can't later claim that he or she didn't write that message.

Older ciphers relied on a single key, and anyone with the key could both encrypt and decrypt messages. You might have had to do a lot of work to transform the message, as with the Enigma engine that drove the Allies nuts during World War II, but the key made the transformation possible. A typical example is any code that requires a key or password. The one-time message pads popular in spy novels are the ultimate single-key ciphers, impossible to break unless you have that exact key.

Unlike single-key ciphers, public-key (or asymmetric) encryption systems use two keys: a private key and a public key. Messages are encrypted with one key and decrypted with the other, and digital signatures ensure the message isn't tampered with en route. The math to explain this is really quite horrendous, but it does work—just accept that really, really large numbers behave really, really oddly. Generally, the key owner keeps the private key secret but hands the public key out to the world at large, for anyone's use. The key owner uses the private key, while everyone else uses the public key.

The key owner can encrypt messages that anyone can read, while anyone in the public can send a message that only the key owner can read.

Public-key cryptography fills our need for integrity, confidentiality, and nonrepudiation. If an author wants anyone to be able to read his message, while ensuring that it isn't tampered with, he can encrypt the message with his private key. Anyone with the public key (that is, the world) can read the message, but tampering with the message renders it illegible. (Depending on the use, he might choose to sign the message digitally instead.)

Encrypting messages this way also ensures that the author of the message has the private key. If someone wants to send a message that can be read only by a particular person, he can encrypt the message with the desired audience's public key. Only the person with the matching private key can read the message.

This works well so long as the private key is kept private. Once the private key is stolen, lost, or made public, the security is lost. A careless person who has his private key stolen could even find others signing documents for him. Be careful with your keys, unless you want to learn that someone used your private key to order half a million dollars' worth of high-end graphics workstations and have them overnighted to an abandoned-house maildrop in inner-city Detroit.[3]

The standard toolkit for all of these operations is OpenSSL.

WHY OPENSSL?

For many years, OpenSSL was the only choice for an encryption library. Today's newer alternatives, although probably more reliable, don't meet FreeBSD's long-term support model. The most obvious replacement, LibreSSL, supports each release for only one year. Until an encryption toolkit is both reliable and can be upgraded throughout the course of a FreeBSD release's lifespan, OpenSSL won't be replaced.

OpenSSL

FreeBSD includes the OpenSSL toolkit for handling public-key cryptography. OpenSSL lets you perform a full range of encryption operations. While many programs use OpenSSL functionality, the sysadmin doesn't need OpenSSL directly very often.

While OpenSSL works fine out of the box, I find it worthwhile to set a few defaults to make my life easier down the road. Configure OpenSSL with the file */etc/ssl/openssl.cnf*. Almost all of the settings in this file are correct as they are, and you shouldn't change them unless you're a cryptographer. The few things useful to change are the defaults for generating cryptographic signatures. Each default value is marked by the string `_default`. You'd be most interested in the following settings for common OpenSSL operations, which I've adjusted to fit my needs:

```
❶ countryName_default            = US
❷ stateOrProvinceName_default    = Michigan
❸ 0.organizationName_default     = Burke and Hare Word Mine, LLC
```

The `countryName_default` ❶ is the two-letter code for your nation—in my case, `US`. The `stateOrProvinceName_default` ❷ is the name of your local state and can be of any length. I would set this to `Michigan`. The `0.organizationName_default` field ❸ is your company name. If I'm buying a signed certificate, I'd put the same thing here that I want to appear on the certificate. If I'm just testing how programs work with SSL and don't have a real company name, I might use the name of the company I work for or something that I make up.

The following values don't show up in *openssl.cnf*, but if you set them, they appear as defaults in the OpenSSL command prompts. I find these useful, even though they change more frequently than the previous defaults—they remind me of the correct format of these answers, if nothing else.

```
❶ localityName_default           = Detroit
❷ organizationalUnitName_default = Pen-Monkey Division
❸ commonName_default             = www.michaelwlucas.com
❹ emailAddress_default           = mwlucas@michaelwlucas.com
```

The `localityName_default` ❶ is the name of your city. The

`organizationalUnitName_default` ❷ is the part of your company this certificate is for. One of the most commonly misunderstood values in OpenSSL, `commonName_default` ❸, is the hostname of the machine this certificate is for, as it appears in reverse DNS. Remember, reverse DNS isn't necessarily the same as the hostname! Your web server might have a nice friendly name, but the hosting company might assign it a totally different name in reverse DNS. Finally, `emailAddress_default` ❹ is the email address of the site administrator.

These values all show up in prompts in the OpenSSL command as default choices. Setting them in the configuration file will save you annoyance later.

Certificates

One interesting thing about public-key encryption is that the author and the audience don't have to be people. They can be programs. Secure Shell (SSH) and the Secure Sockets Layer (SSL) are two different ways programs can communicate without fear of intruders listening in. Public-key cryptography is a major component of the *digital certificates* used by secure websites and secure mail services. When you open Firefox to buy something online, you might not realize that the browser is frantically encrypting and decrypting web pages. This is why your computer might complain about "invalid certificates;" someone's public key has either expired or the certificate is self-signed. Today's protocols encrypt and decrypt with *Transport Layer Security (TLS)* and use *TLS certificates*.

SSL VS. TLS

You hear about SSL all the time, but it's most often incorrect. Today, Transport Layer Security (TLS) has mostly replaced SSL. Most uses of the term *SSL* are lingering remnants. Generally speaking, internet-facing sites should use TLS version 1.1 or better. TLS version 1.0 is only weakly protected. Traffic secured by any version of the SSL protocol isn't secured.

Many companies, such as VeriSign, provide a public-key signing service. These companies are called *Certificate Authorities (CAs)*, as they provide *TLS certificates*. Other companies that need a certificate signed provide proof of their identity, such as corporate papers and business records, and those public-key signing companies sign the applicant's certificate with their CA certificate. By signing the certificate, the CA says, "I have inspected this person's credentials and he, she, or it has proven their identity to my satisfaction." They're not guaranteeing anything else, however. A TLS certificate owner can use the certificate to run a website that sells fraudulent or dangerous products or use it to encrypt a ransom note. Signed TLS certificates guarantee certain types of technical security, not personal integrity or even unilateral technical security. Certificates don't magically apply security patches for you.

Web browsers and other certificate-using software include certificates for the major CAs. When the browser receives a certificate signed by a CA, it recognizes the certificate as legitimate. Essentially, the web browser says, "I trust the Certificate Authority, and the Certificate Authority trusts this company, so I will trust the company." So long as you trust the CA, everything works.

The package `ca_root_nss` contains the CA certificates recognized by the Mozilla Project. If a piece of software fails attempting to validate certificates, make sure you installed this package.

Most CAs are big commercial companies. No matter the size of your organization, though, I encourage you to investigate Let's Encrypt (*https://www.letsencrypt.org/*). Let's Encrypt is a CA that provides free, globally valid TLS certificates.

Using a certificate that's not signed by any CA is perfectly fine for testing. It might also suffice for applications within a company, where you can install the certificate in the client web browser or tell your users to trust the certificate. We'll look at both ways.

Both uses of the certificate require a host key.

TLS Host Key

Both signed and self-signed certificates require a private key for the host. The host key is just a carefully crafted random number. The following command creates a 2,048-bit host key and places it in the file *host.key*:

```
# openssl genrsa 2048 > host.key
```

You'll see a statement that OpenSSL is creating a host key and dots crossing the screen as key generation proceeds. In only a few seconds, you'll have a file containing a key. The key is a plaintext file that contains the words BEGIN RSA PRIVATE KEY and a bunch of random characters.

Protect your host key! Make it owned by root and readable only by root. Once you place your certificate in production, anyone who has that key can use it to eavesdrop on your private communications.

```
# chown root host.key
# chmod 400 host.key
```

Place this host key in a directory with the same permissions that we placed on the key file itself.

Create a Certificate Request

You need a certificate request for either a signed or self-signed certificate. We don't do much with OpenSSL, so we won't dissect this command. Go to the directory with your host key and enter this verbatim:

```
# openssl req -new -key host.key -out csr.pem
```

In response, you'll see instructions and then a series of questions. By hitting ENTER, you'll take the default answers. If you've configured OpenSSL, the default answers are correct.

```
❶ Country Name (2 letter code) [US]:
❷ State or Province Name (full name) [Michigan]:
❸ Locality Name (eg, city) [Detroit]:
❹ Organization Name (eg, company) [Burke and Hare Word Mine, LLC]:
❺ Organizational Unit Name (eg, section) [Pen-Monkey Division]:
❻ Common Name (eg, YOUR name) [www.michaelwlucas.com]:
❼ Email Address [mwlucas@michaelwlucas.com]:
```

The two-letter code for the country ❶ is defined in the ISO 3166 standard, so a quick web search will find this for you. If you don't know the state ❷ and city ❸ you live in, ask someone who occasionally leaves the

server room. The organization name ❹ is probably your company, and you list the department or division name ❺ as well. If you don't have a company, list your family name or some other way to uniquely identify yourself, and for a self-signed certificate, you can list anything you want. Different CAs have different standards for noncorporate entities, so check the CA's instructions.

The common name ❻ is frequently misunderstood. It's not your name; it's the name of the server as shown in reverse DNS. You must have a server name here, or the request will be useless.

I suggest using a generic email address ❼ rather than an individual's email address. In this case, I *am* michaelwlucas.com, so I might as well use my address. You don't want your organization's certificates tied to an individual who might leave the company for whatever reason.

```
   Please enter the following 'extra' attributes
   to be sent with your certificate request
❶ A challenge password []:
❷ An optional company name []:
```

The challenge password ❶ is also known as a *passphrase*. Again, keep this secret because anyone with the passphrase can use your certificate. Use of a certificate passphrase is optional, however. If you use one, you must type it when your server starts. That means that if your web server crashes, the website won't work until someone enters the passphrase. While passphrase use is highly desirable, this might be unacceptable. Hit ENTER to use a blank passphrase.

You've already entered quite a few company names, so a third ❷ is probably unnecessary.

Once you return to a command prompt, you'll see the file *csr.pem* in the current directory. It looks much like your host key, except that the top line says `BEGIN CERTIFICATE REQUEST` instead of `BEGIN RSA PRIVATE KEY`.

Submit *csr.pem* to your Certificate Authority, who will return the actual certificate. I recommend saving the certificate in a file named after the host, such as *www.mwl.io.crt*. This signed certificate is good for any TLS service, including web pages, pop3ssl, or any other TLS-capable daemon.

Some CAs require you use an intermediate certificate with your cert.

While most daemons have a configuration option to specify an intermediate certificate, if yours doesn't, you can append the signed certificate to the end of the intermediate cert.

Sign a Certificate Yourself

A self-signed certificate is technically identical to a signed certificate, but it's not submitted to a Certificate Authority. Instead, you provide the signature yourself. Most customers won't accept a self-signed certificate on a production service, but it's perfectly suitable for testing. To sign your own CSR, run the following:

```
# openssl x509 -req -days❶365 -in csr.pem -signkey host.key \
-out❷selfsigned.crt
Signature ok
subject=/C=US/ST=Michigan/L=Detroit/O=Burke and Hare Word Mine, LLC/OU=Pen-
Monkey Division/CN=michaelwlucas.com/emailAddress=mwlucas@michaelwlucas.com
Getting Private key
#
```

That's it! You now have a self-signed certificate good for 365 days ❶ in the file *selfsigned.crt* ❷. You can use this key exactly like a signed certificate, so long as you're willing to ignore the warnings your application displays.

If you sign your own certificates, client software generates warnings that the "certificate signer is unknown." This is expected—after all, people outside my office have no idea who Michael W. Lucas is or why he's signing web certificates. For some reason, people trust Symantec and other big-company CAs. I'm trusted by the people who know me,[4] but not trusted by the world at large. For this reason, don't use self-signed certificates anywhere the public will see them because the warnings will confuse, annoy, or even scare them away.

But before you go drop any amount of money on a CA certificate, definitely check out Let's Encrypt. It really will change your system administration practice.

TLS Trick: Connecting to TLS-Protected Ports

I said we wouldn't do much with OpenSSL, and that's correct. There's one facility the software offers that's too useful to pass up, however, and once

you know it, you'll use this one trick at least once a month and be glad you have it.

Throughout this book, we test network services by using telnet(1) to connect to the daemon running on that port and issuing commands. This works well for plaintext services such as SMTP, POP3, and HTTP. It doesn't work for encrypted services such as HTTPS. You need a program to manage the encryption for you when you connect to these services. OpenSSL includes the `openssl s_client` command, which is intended for exactly this sort of client debugging. While you'll see a lot of cryptographic information, you'll also get the ability to issue plaintext commands to the daemon and view its responses. Use the command `openssl s_client -connect` with a hostname and port number, separated by a colon. Here, we connect to the secure web server at *www.absolutefreebsd.com*:

```
# openssl s_client -connect www.michaelwlucas.com:443
CONNECTED(00000003)
depth=2 O = Digital Signature Trust Co., CN = DST Root CA X3
verify return:1
depth=1 C = US, O = Let's Encrypt, CN = Let's Encrypt Authority X3
--snip--
```

You'll see lots of stuff about chains of trust and limitations of liability, as well as lines and lines of the random-looking digital certificates. After all that, however, you'll see a blank line with no command prompt. You're speaking directly to the server daemon. As this is a web server, let's try an HTTP command:

```
GET /
```

The system responds with:

```
HTTP/1.1 400 Bad Request
```

The HTTP protocol has changed since the last time I tried this, I guess. But I'm definitely connected to the web server. The network works.

Some of you are probably wondering why we encrypt the service if it's so easy to talk to the encrypted service. The encryption doesn't protect the daemon; it protects the data stream between the client and the server. TLS encryption prevents someone from eavesdropping your network conversation in transit—it doesn't protect either the server or the client.

TLS can't save you if someone breaks into your desktop.

From this point on, I'll assume that you understand this OpenSSL command and what happens when we use it.

HARDWARE CRYPTOGRAPHIC SUPPORT

Most modern hardware has built-in encryption acceleration. Unfortunately, FreeBSD doesn't include it in the default configuration. Hardware crypto acceleration reduces load on the CPU and probably accelerates encryption. The aesni(4) kernel module activates access to Intel's hardware cryptographic accelerator. A driver for the new AMD accelerator is in development. In-kernel drivers affect only encryption that happens in the kernel, such as for encrypted disks and IPSec.

Global Security Settings

FreeBSD supports many optional security settings. These settings change basic FreeBSD behavior, making it differ from the common Unix experience. Some other operating systems provide these settings by default, however, so they're not unique to FreeBSD.

Should you turn all these features on in the name of improved security? There's no universally correct answer here. If restricting access to part of the system to the root account means that you'll need to give more people root access, maybe you shouldn't impose that restriction. A couple of these should be activated on all systems, though.

Install-Time Options

The FreeBSD installer provides an option for enabling each of these settings on first boot. You can enable and disable them later with the given sysctl setting.

Many of these features are especially useful on servers that don't have many users. If your application server doesn't have unprivileged users other than those used by applications, you should probably enable features that restrict unprivileged users. If you have unprivileged users, though, consider the situation more closely. Most of my unprivileged users[5] shouldn't be looking at server processes or other users, so I lock them down.

Hiding Other UIDs' Processes

Normally, commands like `ps -ax` display all processes running on the system. When you set the sysctl `security.bsd.see_other_uids` to 0, users can see only their own processes. Root can see all processes, no matter how you set this.

Hiding Other GIDs' Processes

Similarly, users can normally see processes owned by other groups. Disable that ability by setting the sysctl `security.bsd.see_other_gids` to 0. Again, root can see every process, no matter how this is set.

Hiding Jailed Processes

Users on a host can usually see all processes running in jails. By setting `security.bsd.see_jail_proc` to 0, unprivileged nonjailed users can't see jailed processes. This feature appeared in FreeBSD 12.

Hide Message Buffer

Unprivileged users can normally see the system message buffer, available through dmesg(8). Disable that access by setting the sysctl `security.bsd.unprivileged_read_msgbuf` to 0.

Disable Process Debugging

A debugger can tell users a whole bunch of useful information. Setting `security.bsd.unprivileged_proc_debug` to 0 disallows unprivileged users from using the debugger on processes.

Randomize Process IDs

Traditional Unix systems create process IDs in sequential order, allowing attackers a chance at guessing what the next PID will be. Randomize process IDs by setting the sysctl `kern.randompid` to a random large integer.

If you set it to 1, the kernel picks a fresh random number between 100 and 1,123 at each boot.

Clean /tmp

All sensible Unix-like systems clean */tmp* at boot to dispose of temporary files. Somewhere in the last few years, FreeBSD turned this behavior off by default. You might use tmpfs(5) for */tmp*, which gets destroyed at every power-down. If your */tmp* is on disk, though, well . . . as you're all sensible and wholesome sysadmins, always set `clear_tmp_enable` to `YES` in */etc/rc.conf*.

Disable Syslogd Networking

By default, syslogd(8) creates a half-open socket on UDP port 514. Nobody can connect to this socket; it's used only as a placeholder so nothing else binds to that port. Some people consider this half-open socket problematic. I'd say it's a feature; you don't want something else binding to port 514, claiming to be syslogd, and sending either worrisome or falsely soothing messages to your logging host. But to disable that half-open socket, set `syslogd_flags` to `-ss` in */etc/rc.conf*.

Disable Sendmail

A default FreeBSD install doesn't accept email from the network, but it does run a sendmail(8) daemon to sent outgoing messages. To completely disable sending mail from this host, set `sendmail_enable` to `NONE` in */etc/rc.conf*.

Disabling outbound mail won't prevent the daily, weekly, and monthly maintenance tasks from running. It'll prevent you from receiving the output of those messages unless you log directly onto the host, however. For people with multiple hosts, disabling outbound mail is unwise. Disabling Sendmail makes sense if you use an alternative mail agent, such as dma(8) (see Chapter 20).

Secure Console

Most Unix systems consider the physical console secure. Anyone who has access to the physical machine can do anything to the host that they want, including changing the root password. By changing all of the */etc/ttys* entries

that say `secure` to `insecure`, you tell FreeBSD to demand the root password even in single-user mode.[6] This won't prevent someone from physical access gaining access to your operating system, but it'll mean that they'll have to do slightly more work to subvert your machine. *Very* slightly more work.

Nonexecutable Stack and Stack Guard

One basic exploit mitigation technique is the nonexecutable stack. Once a program is loaded into memory, each page of memory allocated to that program should be either writable or executable, but not both.

A common exploit technique is to trick a program into writing information to memory and then executing that memory. An attacker might convince a program to write to a chunk of memory, but with the nonexecutable stack, the kernel won't execute it.

The stack defaults to nonexecutable on modern versions of FreeBSD. The only reason to disable this is if you have a badly written program that relies on executing and writing the same chunk of memory. Most such defective software has been rightfully purged from the open source ecosystem in the last 15 years. If you're very unlucky and can't avoid running a program that can't handle a nonexecutable stack, you can disable this by setting the sysctls `kern.elf32.nxstack` (for 32-bit programs) or `kern.elf64.nxstack` (for 64-bit programs) to 0.

Related to the nonexecutable stack, a stack guard page adds a random-sized shred of extra memory between parts of a program's memory allocation. This makes it harder for an attacker to guess memory addresses. FreeBSD allocates a stack guard page by default, but you can turn it off by setting the sysctl `security.bsd.stack_guard_page` to 0.

Other Security Settings

Most of FreeBSD's other kernel-level security settings are available in the `security.bsd` sysctl tree. More get added every few months. Run `sysctl -d security.bsd` to display your hosts' available options. I've described many of these earlier in this section, but you might find some of the others useful. Options include disabling the root account's privileges (`security.bsd.suser_enabled`), allowing nonroot users to set an idle priority (`security.bsd.unprivileged_idprio`), and blocking unprivileged users from using

mlock(2) (`security.bsd.unprivileged_mlock`). Take a look at the current options and see what might be useful.

Preparing for Intrusions with mtree(1)

One of the worst things to happen to a sysadmin is something that makes him think that his system could've been penetrated. If you find mysterious files in */tmp* or extra commands in */usr/local/sbin*, or if things "just don't feel right," you'll be left wondering whether someone has compromised your system. The worst thing about this feeling is that there's no way to prove it hasn't happened. A skilled attacker can replace system binaries with her own customized versions, so that her actions are never logged and your attempts to find her will fail. Having Sherlock Holmes examine your server with a magnifying glass is useless when the magnifying glass has been provided by the criminal and includes the special criminal-cloaking feature! People have even hijacked the system compiler so that freshly built binaries include the hijacker's backdoor.[7] What makes matters worse is that computers do weird things all the time. Operating systems are terribly complicated, and applications are worse. Maybe that weird file in */tmp* is something your text editor barfed up when you hit the keys too fast, or perhaps it's a leftover from a sloppy intruder.

The *only* way to recover a compromised system is to reinstall it from scratch, restore the data from backup, and hope that the security hole that led to the compromise is fixed. That's a thin hope, and doubt is so easy to acquire that many sysadmins eventually stop caring or lie to themselves rather than live with the constant worry.

Most intruders change files that already exist on the system. FreeBSD's mtree(1) can record the permissions, size, dates, and cryptographic checksums of files on your system. (While freebsd-update(8) includes similar features, and you don't have to gather data beforehand, it covers only the base system.) If you record these characteristics when your system is freshly installed, you have a record of what those files look like intact. When an intruder changes those files, a comparison will highlight the differences. When you have even the vaguest feeling you've been hacked, you can check that same information on the existing files to see whether any have changed.

Running mtree(1)

The following command runs mtree(1) across your root partition and stores SHA512 and SHA256 cryptographic checksums, placing them in a file for later analysis:

```
# mtree❶-x❷-ic❸-K sha512❹-K sha256❺-p /❻-X /home/mwlucas/mtree-exclude
>❼/tmp/mtree.out
```

While you can use mtree(1) across the entire server, most people use `-x` ❶ to run it once per partition. You don't want to record checksums on frequently changing files, such as the database partition on your database server. Collecting checksums on NFS mounts has the twin features of running really slowly and increasing network congestion. The `-ic` flag ❷ tells mtree to print its results to the screen, with each subsequent layer in the filesystem indented. This format matches the system mtree files in */etc/mtree*. The `-K` flag accepts several optional keywords; in this case, we want to generate SHA512 checksums ❸ and SHA256 checksums ❹. The `-p` flag ❺ tells mtree which partition to check. Almost every partition has files or directories that change on a regular basis and that you therefore don't want to record checksums for. Use `-X` ❻ to specify an *exclusion file*, a file containing a list of paths not to match. Finally, redirect the output of this command to the file */tmp/mtree.out* ❼.

mtree(1) Output: The Spec File

mtree(1)'s output is known as a specification, or *spec*. While this specification was originally intended for use in installing software, we're using it to verify a software install. Your spec starts with comments showing the user who ran the command, the machine the command ran on, the filesystem analyzed, and the date. The first real entry in the spec sets the defaults for this host and begins with `/set`.

```
/set type=file uid=0 gid=0 mode=0755 nlink=1 flags=uarch
```

The mtree(1) program picked these settings as defaults based on its analysis of the files in the partition. The default filesystem object is a file,

owned by UID 0 and GID 0, with permissions of 0755, with one hard link and the user archive flag. After that, every file and directory on the system has a separate entry. Here's the entry for the root directory:

```
❶.        ❷type=dir❸nlink=19❹time=1504101311.033742000
```

This file is the dot (.) ❶, or *the directory we're in right now*. It's a directory ❷, and it has 19 hard links ❸ to it. This directory was modified 1,504,101,311.033742000 seconds into Unix epochal time ❹. The Unix epoch began January 1, 1970.

EPOCHAL SECONDS AND REAL DATES

Don't feel like counting seconds since the epoch began? To convert epochal seconds into normal dates, run `date -r` seconds. Cut off the fraction at the end of mtree's time, however; date(1) likes only whole seconds.

In some ways, the entry for the directory is rather boring. An intruder can't realistically replace the directory itself, after all! Here's an entry for an actual file in the root directory:

```
.cshrc      mode=0644 nlink=2❶size=950 time=1499096179.000000000 \
❷ sha256digest=20d2a78c9773c159bac1df5585227c7b64b6aab6b77bccadbe4c65f1be474e8c \
❸
sha512=24d4330e327f75f10101cd7c0d6a5e59163336ade5b9eb04b0d96ea43d221c5eea4c71a89dfe85a.
```

We see the filename and the same mode, link, and time information as in the root directory, but also get the file size ❶. Additionally, there's the SHA256 ❷ and SHA512 ❸ cryptographic hashes computed from the files.

While it's theoretically possible for an intruder to craft a file that matches a particular cryptographic hash, and while cryptographers are constantly trying to find practical ways to create files that match arbitrary SHA256 and SHA512 checksums, it's extremely unlikely that an intruder can create a fake file that matches both checksums, contains his backdoor, and still functions

well enough that the system owner won't immediately notice a problem. By the time this happens, we will have additional checksum algorithms resistant to those methods and will switch to them.

The Exclusion File

The exclusion file (given with `-x`) lists filesystems you don't want mtree(1) to analyze. Lots of filesystems will change without malicious intervention. Log files and user home directories should change. Directories like */tmp* and */var/db/entropy better* change on a functional system. List each directory you don't want checked on its own line in the exclusion file, with a leading dot.

```
./tmp
./var/db/entropy
./var/log
./usr/home
```

Wait a day or so, and then run mtree(1) again to generate a new spec file. Differences between the two mtree files will let you improve your exclusion file. You'll do the exact same thing when you suspect a system intrusion.

Saving the Spec File

The spec file contains the information needed to verify the integrity of your system after a suspected intrusion. Leaving the spec file on the server you want to verify means that an intruder can edit the file and conceal his wrongdoing. You must not save the file on the system itself! Now and then someone will suggest that you checksum the mtree spec file but keep it on the server. That's not useful; if someone tampers with the mtree file and the checksum, how would you know? Or worse—if someone tampered with the spec file and you caught it, you couldn't tell what change had been made! Copy your spec file to a safe location, preferably on an offline media, such as a flash drive or an optical disk.

Finding System Differences

When something raises your suspicions and you begin to think that you might have suffered an intrusion, create a new mtree spec file and compare it with the "known good" spec file you stored offline. Use mtree(1) to check

for differences between spec files.

```
# mtree -f mtree.suspect -f mtree.good > mtree.differences
```

Every entry in the file is something that has changed. My exclusion file is finely tuned, eliminating files I expect to have changed. This particular run generates two lines of output.

```
                bin/sh file❶size=161672
❷ sha256digest=a4a85ca3563d8f3bda449711c6b591b37093e668fc136f8829eb188b955f56ab
❸
sha512=011793e3e6cacd99b4261e0a0f3a0b9bd6a6842f3ccd55da1ce2070b568e3c49ae7b0e51d33bb59e
                bin/sh file size=❹10489808
❺ sha256digest=45856525d4251b43d68df1429cf1fe0f4adb6640f06d7f995aace5b7ca0c03c2
❻
sha512=a4f0e83e5fb12d615721fd7d57cb6a120068d1aa71fc305b7b86927391f33bec822cf14ce8a8a9dt
```

The file */bin/sh* ❶ has changed size ❹ between mtree runs. This isn't good. Also, note the two different SHA256 hashes ❷ ❺ and the two different SHA512 hashes ❸ ❻. Don't hit the panic button yet, but start asking your fellow sysadmins pointed, hard questions. If you can't get a good answer as to why this binary changed, you might look for your installation media.

Or, perhaps you need to update your exclusion file. But if */bin/sh* changed, probably not.

Monitoring System Security

So, you think your server is secure. Maybe it is . . . for now.

Unfortunately, there's a class of intruders with nothing better to do than to keep up on the latest security holes and try them out on systems they think might be vulnerable. Even if you read *FreeBSD-security* religiously and apply every single patch, you still might get hacked one day. While there's no way to be absolutely sure you haven't been hacked, the following hints will help you find out when something does happen:

- Be familiar with your servers. Run `ps -axx` on them regularly, and learn what processes normally run on them. If you see a process you don't recognize, investigate.

- Examine your open network ports with `netstat -na` and `sockstat`. What TCP and UDP ports should your server be listening on? If you don't recognize an open port, investigate. Perhaps it's innocent, but it might be an intruder's backdoor.
- Unexplained system problems are hints. Many intruders are ham-fisted klutzes with poor sysadmin skills, who use click-and-drool attacks. They'll crash your system and think that they're the cyber incarnation of Samuel L. Jackson.
- Truly skilled intruders not only clean up after themselves but also ensure that the system has no problems that might alert you. Therefore, systems that are unusually stable are also suspicious.
- Unexplained reboots might indicate someone illicitly installing a new kernel. They might also be a sign of failing hardware or bad configuration, so investigate them anyway.
- FreeBSD sends you emails every day giving basic system status information. Read them. Save them. If something looks suspicious, investigate. Look at old messages to see when something has changed.

I particularly recommend the `lsof` package to increase your familiarity with your system. The `lsof` program lists all open files on your system. Reading lsof(8) output is an education in and of itself; you probably had no idea that your web server opened so much crud. Seeing strange files open indicates either that you're not sufficiently familiar with your system or that someone's doing something improper.

Package Security

The FreeBSD Project provides a database of security vulnerabilities in the ports and packages system. This database is made available in *Vulnerability and eXposure Markup Language (VuXML)*. When someone volunteers to maintain a port, they're also volunteering to watch out for security problems with that port.

An internet-connected FreeBSD host with pkg(8) installed downloads the latest VuXML file during the periodic(8) run (see Chapter 21) and stores it in */var/db/pkg/vuln.xml*. It then compares the installed packages with that database. If one of your packages has a vulnerability, you'll be notified in the

daily status email. (You are reading your daily status emails, right?)

If your packages are insecure, upgrade them as per Chapter 15.

If need be, you can set a different location to fetch the *vuln.xml* file with the `VULNXML_SITE` option in *pkg.conf*. You might do this if you maintain your own package repository and vulnerability databases.

If You're Hacked

After all this, what do you do if your system is hacked? There's no easy answer. Huge books are written on the subject. Here are a few general suggestions, however.

First and foremost: A hacked system can't be trusted. If someone has gained root access on your internet server, he could have replaced any program on the system. Even if you close the hole he broke in through, he could have installed a hacked version of login(8) that sends your username and password to an IRC channel somewhere every time you log in. Do not trust this system. An upgrade can't cleanse it, as even freebsd-update(8) and the compiler are suspect.

While rootkit-hunting software might help you verify the presence of intruders, nothing can verify that the intruder *isn't* there. Feel free to write *FreeBSD-security@FreeBSD.org* for advice. Describe what you're seeing and why you think you're hacked. Be prepared for the ugly answer, though: completely reinstall your computer from known secure media, and restore your data from backup. You did read Chapter 5, right?

Good security practices reduce your chances of being hacked, just as safe driving reduces your chances of being in a car wreck. Eventually you'll total your wheels anyway and wonder why you bothered. Good luck!

20
SMALL SYSTEM SERVICES

Even a server with a very narrowly defined role, such as a dedicated web server, needs a variety of small "helper" services to handle basic administrative issues. In this chapter, we'll discuss some of those services, such as time synchronization, sending mail, DHCP services, scheduling tasks, and so on. We'll start by securing your remote connections to your FreeBSD server with SSH.

Secure Shell

One of Unix's great strengths is its ease of remote administration. Whether the server is in front of you or in a remote, barricaded laboratory in a subterranean, maximum-security installation surrounded by vicious guard dogs mentored by a megalomaniacal weasel named Ivan, if you have network access to the machine, you can control it.

For many years, telnet(1) was the standard way to access a remote server. As a remote administration protocol, however, telnet has one crushing problem: everything sent over most versions of telnet is unencrypted. Anyone with a packet sniffer, attached anywhere along your connection, can steal your username, your password, and any information you view in your telnet session. When you use telnet, the best password-selection scheme in the world can't protect your username and password. Intruders place illicit packet sniffers anywhere they can; I've seen them on small local networks and global enterprise networks, in law firms handling sensitive government work, on home PCs, and on internet backbones. The only defense against a

packet sniffer is to handle your authentication credentials and data in such a way that a packet sniffer can't make sense of them. That's where SSH, or secure shell, comes in.

SSH behaves much like telnet in that it provides a highly configurable terminal window on a remote host. But unlike telnet, SSH encrypts everything you send across the network. SSH ensures not only that your passwords can't be sniffed but also that the commands you enter and their output are encrypted. While telnet does have a few minor advantages over SSH in that it requires less CPU time and is simpler to configure, SSH's security advantages utterly outweigh them. SSH also has many features that telnet doesn't have, such as the ability to tunnel arbitrary protocols through the encrypted session. SSH runs on every modern variant of Unix and even on Microsoft Windows.

SSH encrypts and authenticates remote connections via public-key cryptography. The SSH daemon offers the server's public key to clients and keeps the private key to itself. The client and server use the cryptographic key to negotiate a cryptographically secure channel between them. Since both public and private keys are necessary to complete this transaction, your data is secure; even if someone captures your SSH traffic, they can see only encrypted garbage.

To use SSH, you must run an SSH server on your FreeBSD machine and an SSH client on your workstation.

The SSH Server: sshd(8)

The sshd(8) daemon listens for SSH requests coming in from the network on TCP port 22. To enable sshd at boot, add the following line to */etc/rc.conf*:

```
sshd_enable="YES"
```

Once this is set, you can use the */etc/rc.d/sshd* script or `service sshd` subcommands to start and stop SSH. Stopping the SSH daemon doesn't terminate SSH sessions that are already in use; it only prevents the daemon from accepting new connections.

Unlike unencrypted protocols we look at, sshd is difficult to test by hand. One thing you can do is confirm that sshd is running by using nc(1) to

connect to the SSH TCP port.

```
# nc localhost 22
SSH-2.0-OpenSSH_7.2 FreeBSD-20160310
```

We connect to port 22, and get an SSH banner back. We can see that the daemon listening on this port calls itself SSH version 2, implemented in OpenSSH 7.2, on FreeBSD, version 20160310. You can get all this information from a simple nc(1) connection, but it's the last free information sshd offers. Unless you're capable of encrypting packets by hand, on the fly, this is about as far as you can go. Press CTRL-C to leave nc(1) and return to the command prompt.

SSH Keys and Fingerprints

The first time you start sshd(8), the program realizes that it has no encryption keys and automatically creates them. The initializing sshd process creates three pairs of keys: an RSA key, an ECDSA key, and an ED25519 key.

The key files ending in *.pub* contain the public keys for each type of key. These are the keys that sshd hands to connecting clients. This gives the connecting user the ability to verify that the server he's connecting to is really the server he thinks it is. (Intruders have tricked users into logging into bogus machines in order to capture their usernames and passwords.) Take a look at one of these public-key files; it's pretty long. Even when a user is offered the chance to confirm that the server is offering the correct key, it's so long that even the most paranoid users won't bother to verify every single character.

Fortunately, SSH allows you to generate a *key fingerprint*, which is a much shorter representation of a key. You can't encrypt traffic or negotiate connections with the fingerprint, but the chances of two unrelated keys having the same fingerprint are negligible. To generate a fingerprint for a public key, enter the command `ssh-keygen -lf` *`keyfile.pub`*.

```
# ssh-keygen -lf /etc/ssh/ssh_host_rsa_key.pub
2048 SHA256:tEcBfgXctTfaaEF9d5QK3oYUwr5Tb/cuIr3MNxV4wwE root@bert (RSA)
```

The first number, `2048`, shows the number of bits in the key. 2048 is standard for an RSA key in 2018, but as computing power increases, I expect

this number to increase. The string starting with `tEcB` and ending with `wwE` is the fingerprint of the public key. While it's long, it's much shorter and much more readable than the actual key. Copy this key fingerprint from the original server to a place where you can access it from your client machines. If a human needs to verify the fingerprint, try a web page or a paper list. If your SSH clients support SSHFP records and your DNS zones support DNSSEC, you can use DNS instead. Use this key to confirm your server's identity the first time you connect, or use one of the other key distribution methods.

Configuring the SSH Daemon

While sshd comes with a perfectly usable configuration, you might want to tweak the settings once you learn all the features sshd(8) offers. The configuration file */etc/ssh/sshd_config* lists all the default settings, commented out with a hash mark (`#`). If you want to change the value for a setting, uncomment the entry and change its value.

We won't discuss all the available sshd options; that would take a rather large book of its own. Moreover, OpenSSH advances quickly enough to make that book obsolete before it hits the shelves. Instead, we'll focus on some of the more common desirable configuration changes people make.

After changing the SSH daemon's configuration, restart the daemon with `/etc/rc.d/sshd restart` or `service sshd restart`.

VersionAddendum FreeBSD-20170902

The `VersionAddendum` appears in the server name when you connect to sshd's TCP port. Some people recommend changing this to disguise the operating system version. Identifying a computer's operating system is simple enough, however, by using fingerprinting techniques on packets exchanged with the host, so this isn't generally worth the time. (On the other hand, if changing `VersionAddendum` to `DrunkenBadgerSoftware` amuses you, proceed.)

Port 22

sshd(8) defaults to listening to TCP port 22. If you want, you can change this to a nonstandard port. If you want sshd to listen to multiple ports (for example, port 443 in addition to port 22), you can include multiple `Port`

entries on separate lines:

```
Port 22
Port 443
```

Changing the port isn't useful as a security measure. It can be useful to reduce log chatter. I freely admit to having a small SSH server that listens on a variety of popular TCP ports specifically to bypass useless network security devices. But it doesn't make SSH any more *secure*.

ListenAddress 0.0.0.0

sshd defaults to listening for incoming requests on all IP addresses on the machine. If you need to restrict the range of addresses to listen on (for example, on a jail server), you can specify it here:

```
ListenAddress 203.0.113.8
```

If you want sshd to listen on multiple addresses, use multiple `ListenAddress` lines.

SyslogFacility AUTH and LogLevel INFO

These two settings control how sshd(8) logs connection information. See Chapter 21 for more information on logging.

LoginGraceTime 2m

This controls how long a user has to log in after getting connected. If an incoming user connects but doesn't successfully log in within this time window, sshd drops the connection.

PermitRootLogin no

Do not let people log into your server as root. Instead, they should SSH in as a regular user and become root with su(1). Allowing direct root logins eliminates any hope you have of identifying who misconfigured your system and allows intruders to cover their tracks much more easily.

MaxAuthTries 6

This is the number of times a user may attempt to enter a password during a

single connection. After this number of unsuccessful attempts to log in, the user is disconnected.

AllowTcpForwarding yes

SSH allows users to forward arbitrary TCP/IP ports to a remote system. If your users have shell access, they can install their own port forwarders, so there's little reason to disable this.

X11Forwarding yes

Unix-like operating systems use the X11 (or X) protocol to display graphical programs. In X, the display is separated from the physical machine. You can run, say, a web browser on one machine and display the results on another.

As X has had a checkered security history, many admins reflexively disable X forwarding. Denying X forwarding over SSH doesn't disable X forwarding in general, however. Most users, if denied SSH-based X forwarding, just forward X over unencrypted TCP/IP using either X's built-in network awareness or a third-party forwarder, which in most circumstances is far worse than allowing X over SSH. If your sshd server has the X libraries and client programs installed, a user can forward X one way or another; it's best to let SSH handle the forwarding for you. If you don't have the X software installed, then `X11Forwarding` has no effect.

Banner /some/path

The banner is a message that's displayed before authentication occurs. The most common use for this option is to display legal warnings. The default is not to use a banner.

Subsystem sftp /usr/libexec/sftp-server

SSH allows you to securely copy files from one system to another with scp(1). While scp works well, it's not very user-friendly. The sftp server provides an FTP-like interface to file transfer, reducing the amount of time you must spend on user education but still maintaining solid security.

Managing SSH User Access

By default, anyone with a legitimate shell can log into the server. Using the

configuration variables `AllowGroups`, `DenyGroups`, `AllowUsers`, and `DenyUsers`, sshd(8) lets you define particular users and groups that may or may not access your machine.

When you explicitly list users who may SSH into a machine, any user who isn't listed can't SSH in.

For example, the `AllowGroups` option lets you restrict SSH access to users in specified groups defined in */etc/group* (see Chapter 9). If this option is set and a user isn't in any of the allowed groups, he can't log in. Separate multiple groups with spaces:

```
AllowGroups wheel webmaster dnsadmin
```

If you don't want to give a whole group SSH access, you can list individual users with `AllowUsers`. By using `AllowUsers`, you disallow SSH access for everyone except the listed users.

The `DenyGroups` list is the opposite of `AllowGroups`. Users in the specified system groups can't log in. The listed group must be their primary group, meaning it must be listed in */etc/master.passwd* and not just */etc/group*. This limitation makes `DenyGroups` less useful than it seems at first; you can't define a general group called *nossh* and just add users to it, unless you make it their primary group as well. Explicitly listing allowed groups is a much more useful policy.

Finally, the `DenyUsers` variable lists users who may not log in. You can use this to explicitly forbid certain users who are in a group that is otherwise allowed.

These four different settings make it possible for a user to be in multiple groups simultaneously. For example, one user might be in a group listed in `AllowGroups` and a group listed in `DenyGroups`. What then? The SSH daemon checks these values in the order: `DenyUsers`, `AllowUsers`, `DenyGroups`, and `AllowGroups`. The first rule that matches wins. For example, suppose Bert is a member of the wheel group. Here's a snippet of *sshd_config* :

```
DenyUsers: bert
AllowGroups: wheel
```

Bert can't SSH into this machine because `DenyUsers` is checked before `AllowGroups`.

SSH Clients

Of course, FreeBSD comes with the SSH client, as do most Unix-like operating systems. If possible, use the included SSH client—it's part of OpenSSH, developed by a subset of the OpenBSD team, and it's not only the most popular implementation but also the best. If you've been sentenced to run a Microsoft operating system, I recommend PuTTY, which is free for commercial or noncommercial purposes and has excellent terminal emulation. Microsoft is integrating a fork of OpenSSH into Windows, but it's still in beta as I write this.

This is a FreeBSD book, so we'll focus on FreeBSD's OpenSSH client. You can configure the client in a variety of ways, but the most common configuration choices available simply disable the functions offered by the server. If you're really interested in tweaking your client's behavior, read ssh_config(5).

To connect to another host with SSH, type `ssh` *`hostname`*. In response, you'll see something like this:

```
# ssh mwl.io
The authenticity of host 'mwl.io (203.0.113.221)' can't be established.
ECDSA key fingerprint is SHA256:ZxOWglg4oqcZKHOLv5tfqPlAwDW6UGVbiTvjfAjMc4E.
No matching host key fingerprint found in DNS.
Are you sure you want to continue connecting (yes/no)? yes
```

Your client immediately retrieves the public key from the host you're connecting to and checks its own internal list of SSH keys for a matching key for that host. If the key offered by the server matches the key the client has in its list, the client assumes you're talking to the correct host. If the client doesn't have the host key in its list of known hosts, it presents the key fingerprint for your approval.

The fingerprint presented by the SSH client should be identical to the fingerprint you generated on your server. If the fingerprint isn't identical, you're connecting to the wrong host and you need to immediately disconnect. If it matches, accept the key and continue. Once you accept the fingerprint, the key is saved under your home directory in *.ssh/known_hosts*.

If you're building a new server on your local network for your private use, perhaps you don't have to manually compare the key fingerprints. You should still copy the key fingerprint, however, since you'll eventually want to connect from a remote location and will need to verify the key. If many

people will connect to a server, it's generally okay to put the fingerprint on a web page. You must decide how much security you need. I strongly encourage you to err on the side of caution.

Accept the host key, and you'll be allowed to log into the server. While using a private key with a passphrase is preferable to using passwords, a password with SSH is still better than telnet.

Copying Files over SSH

The SSH client is fine for command line access, but what about moving files from one system to another? SSH includes two tools for moving files across the network: scp(1) and sftp(1).

scp(1) is "secure copy" and is ideal for moving individual files. scp takes two arguments: first, the file's current location; then, the desired location. The desired location is specified as *<username>@<hostname>:<filename>*. Suppose I want to copy the file *bookbackup.tgz* from my local system to the remote server *mwl.io*, giving the remote copy a different name. I'd run:

```
# scp bookbackup.tgz mwlucas@mwl.io:bookbackup-january.tgz
```

If you want to give the new copy the same name, you can leave off the filename in the second argument:

```
# scp bookbackup.tgz mwlucas@mwl.io:
```

scp(1) also lets you copy files from a remote system to your local system:

```
# scp mwlucas@mwl.io:bookbackup-january.tgz bookbackup.tgz
```

If you don't want to change the filename on the local system, you can use a single dot as the destination name:

```
# scp mwlucas@mwl.io:bookbackup.tgz .
```

Finally, if your username on the remote system is the same as your local username, you can delete the username and the `@` sign. For example, to back up my work, I just use:

```
# scp bookbackup.tgz mwl.io:
```

While this looks complicated, it's quite useful for quickly moving individual files around the network.

If you like interactive systems or if you don't know the precise name of the file you want to grab from a remote server, sftp(1) is your friend. sftp(1) takes a single argument, the username and server name, using scp's syntax for a remote server:

```
# sftp mwlucas@mwl.io
Connecting to bewilderbeast...
Password:
sftp> ls
```

The sftp(1) client looks much like a standard command line FTP client; it supports the usual FTP commands, such as `ls` (list), `cd` (change directory), `get` (download a file), and `put` (upload a file). One important difference is that sftp(1) doesn't require a choice between ASCII and binary transfers; it just transfers the file as is.

With SSH, scp, and sftp, you can completely eliminate cleartext passwords from your network.

OPENSSH PASSWORDS AND KEYS

To truly secure your system, use key-based SSH authentication. Creating keys isn't hard, but deploying them in a way that best suits your environment is more complicated than I can fit in here. Eliminating SSH passwords is the single greatest security improvement you can make in your network.

While SSH is the most common sysadmin tool, we've just brushed its surface. Time you spend mastering SSH will pay itself back several fold. You can find several good tutorials online and a few decent books, including my own *SSH Mastery* (Tilted Windmill Press, 2018).

Email

Running an email server has become vastly more complicated in the last few

years. Coping with the spam, viruses, and random crud that arrives on a mail server requires a specialized skill set, and the amount of that crud balloons every year. Think carefully before you deploy a mail server. Every host needs some sort of mail client, however. FreeBSD includes two software suites that can be used for managing local mail and forwarding mail to the mail server: Sendmail and the Dragonfly Mail Agent.

Sendmail is the great-granddaddy of mail programs. It can be a server, a client, a filter, and an arbitrary mail spindler. If you want to exchange mail with sites so isolated that they communicate once a day over UUCP over a dialup line, and also exchange mail with the latest commercial mail servers, Sendmail is a solid choice. For most of us, though, the Swiss Army Car Crusher of Email is overkill.

The *Dragonfly Mail Agent (DMA)* comes from Dragonfly BSD. It's a very minimal mail client that can deliver mail on the local host or forward it to a mail server. It's exactly what your average host needs to forward daily status mails to the minion tasked with reading them, to send reports from your application to the application administrator, and to forward all those annoying reports your WordPress security plugin wants you to read.

We'll spend some time with DMA. Before we can go there, though, let's talk about how FreeBSD copes with the world's multiplicity of mail servers.

mailwrapper(8)

For decades, Sendmail was the only mail server available for Unix-like systems. As such, huge amounts of software expects every server to have */usr/sbin/sendmail* and expects it to behave exactly like Sendmail. What makes matters worse, Sendmail behaves differently when called by different names. The program mailq(1) is a hard link to sendmail(8), but as it has a different name, it behaves differently. So do newaliases(1), send-mail(8), hoststat(8), and purgestat(8).[1]

As clients expect to find Sendmail, any replacement mail server must precisely emulate Sendmail, down to this multiname behavior. Using a different mail server isn't as easy as erasing the Sendmail binaries and replacing them with something else. But people try.

As a result, sysadmins exploring unfamiliar Unix systems might have no idea what */usr/sbin/sendmail* really is! If someone previously installed several

different mail servers in an effort to find something less ghastly than Sendmail, you'll have to resort to detective work and dogged persistence to identify your so-called sendmail(8).

FreeBSD does an end-run around all this confusion by using a separate mailwrapper(8) program. The mail wrapper directs requests for Sendmail to the preferred mail server, installed elsewhere.

Configure mailwrapper(8) in */etc/mail/mailer.conf*. This file contains a list of program names, along with the paths to the actual programs to be called. Here's the default *mailer.conf* directing everything to good old sendmail(8):

```
sendmail        /usr/libexec/sendmail/sendmail
send-mail       /usr/libexec/sendmail/sendmail
mailq           /usr/libexec/sendmail/sendmail
newaliases      /usr/libexec/sendmail/sendmail
hoststat        /usr/libexec/sendmail/sendmail
purgestat       /usr/libexec/sendmail/sendmail
```

Each of these six "programs" in the left column is a name that other programs might use for Sendmail. The right column gives the path to the program that should be called instead. Here, we see that Sendmail is installed as */usr/libexec/sendmail/sendmail*. If you use an alternative mailer, you must edit *mailer.conf* to point to the proper path to the mailer programs. Most alternative mailers use separate programs for each of these functions because the cost of disk space has plunged since Sendmail's birth. When you install an alternative mailer from a package or port, the post-install message usually provides instructions on exactly how to update *mailer.conf* for your installation. Follow those instructions if you want the new mail server to work. If you install a different mail server without using a package, you need to edit *mailer.conf* yourself.

The Dragonfly Mail Agent

The Dragonfly Mail Agent (DMA) can deliver mail locally and send mail to another server. It can't receive mail over the network. Where most mail servers bind to TCP port 25 on the local host, dma(8) does not. It delivers mail only for programs that can call */usr/sbin/sendmail* or one of its counterparts.

Before activating DMA, configure it in */etc/dma/dma.conf*. This file contains variables you can uncomment and set to a specific value. While

DMA has several configurable settings, you should leave most of them at the default.

Smart Host

A smart host is the actual mail server, the host this client should relay mail through. Use the hostname or IP address.

```
SMARTHOST=mail.mwl.io
```

TCP Port

If your mail administrator is a madman that runs the smart host's email on a nonstandard port, or if you're trying to evade your ISP's block port 25 outbound, set the TCP port here:

```
PORT 2025
```

If you don't set a smart host but do set a port, you'll break mail delivery.

False Hostname and Usernames

You might want your server to claim to be a different host when it sends mail. Maybe your cloud provider has given this system a hostname composed of random digits and numbers, but you want it to send mail as *www.example.com*. Use the MAILNAME to set a fake hostname.

```
MAILNAME www.mwl.io
```

If you give MAILNAME the full path to a file, dma(8) will use the first line of that file as the hostname.

Some mail servers very strictly inspect relayed mail and reject inadequately forged messages. For those hosts, you'll need to use the MASQUERADE option. Masquerading gives you a couple different options for changing messages. If you use an entire email address, all mail sent via dma(8) is rewritten so it comes from that address. If you use a username with an @ sign, such as `bert@`, all email appears to be coming from that user at the host. A hostname on its own leaves the sending username untouched but changes the hostname.

```
MASQUERADE bert@mwl.io
```

Any messages sent from this host appear to be from Bert. Any replies will go to him. All is as it should be.

Disable Local Delivery

Some hosts should never receive mail. No account on the host should ever get mail, not even from other local accounts. Totally disable local mail delivery by uncommenting the NULLCLIENT option.

Secure Transport

Over the decades, the email protocol has had a whole bunch of different security measures wedged into it. Your mail server might use any or all of them. Speak to your email administrator about what your smart host requires and supports.

Enable TLS (or SSL, if your mail server is notably awful) by uncommenting the SECURETRANSFER option. You don't need to set this to a value; its mere presence turns on TLS. If your mail server needs STARTTLS, also uncomment that option. If you want to send mail even if TLS negotiation fails, also uncomment OPPORTUNISTIC_TLS.

These three options all require the previous options. You can use SECURETRANSFER on its own, STARTTLS and SECURETRANSFER together, or all three. STARTTLS and SECURETRANSFER without their preceding options don't work.

If you need a local TLS certificate, set it with the CERTFILE option.

```
CERTFILE /etc/ssl/host.crt
```

These options should let you connect to just about any smart host.

Username and Password

Some smart hosts require clients authenticate with a username and password. Put authentication credentials in the file */etc/dma/auth.conf*. Each entry needs the format:

```
user|host:password
```

Suppose my smart host is *mail.mwl.io*. The username is *www1*, and the

password is *BatteryHorseStapleCorrect*. My *auth.conf* would contain:

```
www1|mail.mwl.io:BatteryHorseStapleCorrect
```

DMA will use this to log into your host.

If you want to use a username and password over an unencrypted connection, you must set the INSECURE variable. Sending unencrypted authentication information over the network is a bad idea, but many mail servers are full of bad ideas.

Enabling DMA

Using DMA requires shutting down any existing Sendmail processes and enabling dma(8) in *mailer.conf*.

Sendmail runs as a daemon even when it only handles local delivery. Shut down Sendmail with service(8) or the */etc/rc.d/sendmail* script.

```
# service sendmail stop
```

Make sure it never starts again.

```
# sysrc sendmail_enable=NONE
```

Now, go to */etc/mail/mailer.conf* and point every mail program to dma(8).

```
sendmail        /usr/libexec/dma
send-mail       /usr/libexec/dma
mailq           /usr/libexec/dma
newaliases      /usr/libexec/dma
rmail           /usr/libexec/dma
```

DMA has no persistent daemon, so it doesn't need a startup script.

Congratulations, you now have a small, simple, effective client mail agent.

The Aliases File and DMA

The */etc/mail/aliases* file contains redirections for email sent to specific accounts or usernames. Even mail clients and mail agents like DMA use the aliases file. Adding an entry to the aliases file is a good way to locally redirect email.

While the aliases file has a whole bunch of features, DMA can exercise

only a few of them. Features like redirecting email to an arbitrary file don't work. We'll discuss the basic functions.

Open up the aliases file and look around. Each line starts with an alias name or address, followed by a colon and a list of real users to send the email to. We'll illustrate how aliases work by example.

Forwarding Email from One User to Another

Someone should always read email sent to the root account. Rather than having that someone log onto every server to read the messages, forward all of root's email to another email address.

```
root: bert@mwl.io
```

I've assigned Bert the job of reading all the mail from all the machines.[2]

Many email addresses don't have accounts associated with them. For example, the required postmaster address often doesn't have an account. You can use an alias to forward this to a real account.

```
postmaster: root
```

So, postmaster forwards to root, which forwards to Bert. Bert gets all the email for these two addresses.

The default aliases file contains a variety of standard addresses for internet services, as well as aliases for all of the default FreeBSD service accounts. They all go to root by default. By defining a real address as a destination for your root email, you'll automatically get all system administration email.

Aliased Mailing Lists

You can list multiple users to create small mailing lists. This doesn't scale for dynamic lists, but it's sufficient for quick and dirty lists.

```
escalate: mwlucas@mwl.io, bert@mwl.io, helpdesk@mwl.io
```

The moment you find yourself creating an aliased mailing list is the moment you need to start considering which mailing list solution you're going to deploy. You'll need it sooner than you think.

Network Time

If a database starts entering dates three hours behind, or if emails arrive dated tomorrow, you'll hear about it pretty quickly. Time is *important*.[3] You have two tools to manage system time: tzsetup(8) to control the time zone and ntpd(8) to adjust the clock. Start by setting your time zone manually, and then use network time protocol.

Setting the Time Zone

Time zone is easy to manage with tzsetup(8), a menu-driven program that makes the appropriate changes on your system for each time zone. Global organizations might use the default of UTC (Universal Time Clock, previously known as Greenwich Mean Time, currently known as Coordinated Universal Time, soon to be known by Yet Another Name) on their systems, while others use their own local time. Enter `tzsetup`, follow the geographic prompts, and choose the appropriate time zone for your location. If you know your time zone's official name, you can set it at the command prompt without going through the prompts.

```
# tzsetup America/Detroit
```

The tzsetup(8) program copies the relevant time zone file from */usr/share/zoneinfo* to */etc/localtime*. This is a binary file, and you can't edit with your average text editor. If the characteristics of your time zone change —for example, the day Daylight Saving Time begins changes—you must upgrade FreeBSD to get the new time zone files and then rerun tzsetup(8) to correctly reconfigure time.

Users can use the `TZ` environment variable to set their personal time zone.

Network Time Protocol

Network time protocol (NTP) is a method to synchronize time across a network. You can make your local computer's clock match the atomic clock at your government's research lab or the time on your main server. Computers that offer time synchronization are called *time servers* and are roughly lumped into two groups: Tier 1 and Tier 2.

Tier 1 NTP servers are directly connected to a highly accurate

timekeeping device. If you really need this sort of accuracy, then what you really need is your own atomic clock. A USB radio clock such as that found on an inexpensive GPS might look very nice, but USB turns out to be a lousy medium for transferring timing data. Go price a dedicated non-USB GPS receiver, and then choose a Tier 1 NTP server instead.

Tier 2 NTP servers feed off the Tier 1 NTP servers, providing time service as a public service. Their service is accurate to within a fraction of a second and is sufficient for almost all non–life sustaining applications. Some digging will even lead you to Tier 3 time servers, which feed off of Tier 2 servers.

The best source of time servers is the list at *http://www.pool.ntp.org/*. This group has collected public NTP servers into round-robin DNS pools, allowing easy NTP configuration. These NTP servers are arranged first in a global list, then by continent, and then by country. For example, if you're in Canada, a brief search on that site leads you to *0.ca.pool.ntp.org*, *1.ca.pool.ntp.org*, and *2.ca.pool.ntp.org*. We'll use these servers in the following examples, but look up the proper servers for your country and use those instead when setting up your own time service.

Configuring ntpd(8)

ntpd(8) checks the system clock against a list of time servers. It takes a reasonable average of the times provided by the time servers, discarding any servers too far away from the consensus, and gradually adjusts the system time to match the average. This gives the most accurate system time possible, without demanding too much from any one server, and helps keep errant hardware in check. Configure NTP in */etc/ntpd.conf*. Here's a sample that uses Canadian time servers:

```
server 1.ca.pool.ntp.org
server 2.ca.pool.ntp.org
server 3.ca.pool.ntp.org
```

This system checks three time servers for updates. If you list only one server, ntpd(8) slaves its clock to that one server and shares any time problems that server experiences. Using two time servers guarantees that your system won't know what time it is; remember, NTP takes an average of its time servers but throws out any values too far out of range of the others. How can NTP decide whether one server is wrong when it has only two

values to choose from? Using three time servers is optimal; if one server runs amok, ntpd recognizes that the time offered by that server doesn't make sense against the time offered by the other two servers. (Think of this as a "tyranny of the majority"; the one guy whose opinion differs from the rest doesn't get any voice at all.)

ntpd(8) at Boot Time

To have ntpd perform a one-time clock synchronization at boot and then continually adjust the clock afterward, set the following in */etc/rc.conf*:

```
ntpd_enable="YES"
ntpd_sync_on_start="YES"
```

Ntpd will force correct time immediately on boot and then gently keep the clock synchronized.

Instant Time Correction

ntpd(8) is great at keeping the system clock accurate over time, but it adjusts the local clock only gradually. If your time is off by hours or days (which isn't unlikely at install time or after a long power outage), you probably want to set your clock correctly before letting any time-sensitive applications start. ntpd(8) includes that functionality as well, with `ntpd -q`.

To perform a single brute-force correction of your clock, use `ntpd -q`. This connects to your NTP servers, gets the correct time, sets your system clock, and exits.

```
# ntpd -q
ntpd: time set -76.976809s
```

This system's time was off by about 77 seconds but is now synchronized with the NTP servers.

Do not change the clock arbitrarily on a production system. Time-sensitive software, such as many database-driven applications, has problems if time suddenly moves forward or backward.

If you have really good hardware with an excellent oscillator, using `ntpd -q` at boot handles all of your time problems. Very few people have that sort of hardware, however. Most of us have to make do with commodity hardware with notoriously poor clocks. The best way to ensure you have accurate time

is to run ntpd(8) to gently adjust your clock on an ongoing basis.

Redistributing Time

While ntpd doesn't use a large amount of network bandwidth, having every server on your network query the public NTP servers is a waste of network resources—both yours and that of the time-server donors. It can also lead to very slight (subsecond) variances in time on your own network.

Reliable time servers aren't virtual machines. Tier 1 NTP servers are all run on real hardware specifically to avoid the clock jittering virtual machines can suffer.

I recommend setting up three authoritative time servers for your network. Have these servers synchronize their clock with the global NTP pool. Configure each server on your network to point to these servers for NTP updates. That way, every clock on your network will be perfectly synchronized. You won't have to trawl through NTP logs to try to determine whether a particular server in the global time server pool has somehow messed up your system clock. It's best to enforce this policy via firewall rules at your network border; allowing only your time server to communicate with outside NTP servers eliminates one common source of temporal chaos.

Name Service Switching

Any Unix-like system performs innumerable checks of many different name services. We've already talked about the Domain Name System that maps hostnames to IP addresses (see Chapter 7), but there's also a password entry lookup service, a TCP/IP port number and name lookup service, an IP protocol name and number lookup service, and so on. You can use */etc/nsswitch.conf* to configure how your FreeBSD system makes these queries and what information sources it uses through nsswitch (name service switching).

Each name service has an *nsswitch.conf* entry including the type of the service and the information sources it uses. We previously saw an example of name service switching in Chapter 8. Remember this entry for host lookups?

```
hosts: files dns
```

This means, "Look for IP addresses in the local files first, and then query DNS." The other information sources work similarly. FreeBSD, like most other Unix-like operating systems, supports name service switching for the information sources listed in Table 20-1.

Table 20-1: Lookups Supporting Name Service Switching

Lookup	Function
`groups`	Group membership checks (*/etc/group*)
`hosts`	Hostname and IP checks (DNS and */etc/hosts*)
`networks`	Network entries (*/etc/networks*)
`passwd`	Password entries (*/etc/passwd*)
`shells`	Checks for valid shells (*/etc/shells*)
`services`	TCP and UDP services (*/etc/services*)
`rpc`	Remote procedure calls (*/etc/rpc)*
`proto`	TCP/IP network protocols (*/etc/protocols*)

Most of these you don't want to muck with, unless you like breaking system functionality. If you have a Kerberos or an NIS domain, for example, you might want to have your FreeBSD box attach to them for user and group information—but if you don't, reconfiguring the password lookups would make your system slow at best or entirely stop working at worst!

For each name service, you must specify one or more sources of information. Many of these name services are very simple and default to having a single authoritative source of information—a file. Others, such as the host's name service, are more complicated and have multiple sources. A few are very complicated simply because of the vast array of information available and the many possible ways to get that information. As this book doesn't cover Kerberos, NIS, or any other enterprise-level user management systems, we won't cover changing password, group, and shell information sources. If you're in such an environment, read nsswitch.conf(5) for details.

Most common services have specific valid information sources. *Files* are the standard text files containing information for the service. For example, network protocols are traditionally stored in */etc/protocols*, network services in

/etc/services, and passwords in */etc/passwd* and friends. A source of *dns* means that the information is available on a DNS server, as is typical for the hosts service responsible for mapping hostnames to IP addresses. The password service often uses *compat*, which grants compatibility with */etc/passwd* and NIS but could also use *files*. You might add information sources to the system—for example, enabling LDAP authentication adds the *ldap* information source.

List each desired information source in the order you want them to be tried. Our `hosts` entry tells the name service lookup to try the local file first and then query the DNS server.

```
hosts: files dns
```

If you deploy a central authentication scheme like LDAP, you'll need to add an appropriate entry to tell the host to look up passwords and groups in LDAP. The important question is, should hosts use their local password file and then fall back to LDAP or start with LDAP and fall back to the password file?

```
passwd: ldap files
```

Here, we start with LDAP but fall back to the password file if LDAP isn't available.

inetd

The inetd(8) daemon handles incoming network connections for less frequently used network services. Most systems don't have a steady stream of incoming FTP requests, so why have the FTP daemon running all the time? Instead, inetd listens to the network for incoming FTP requests. When an FTP request arrives, inetd(8) starts the FTP server and hands off the request. Other common programs that rely on inetd are telnet, tftp, and POP3.

Inetd also handles functions so small and rarely used that they're easier to implement within inetd, rather than route them through a separate program. This includes discard (which dumps any data received into the black hole of */dev/null*), chargen (which pours out a stream of characters), and other

functions. These days, most of these services are not only not required but often considered harmful. The chargen service, for example, is mostly useful for denial-of-service attacks.

INETD SECURITY

Some sysadmins think of inetd as a single service with a monolithic security profile. Others say that inetd has a bad security history. Neither is exactly true. The inetd server itself is fairly secure, but it absorbs a certain amount of blame for the programs it forwards requests to. Some services that inetd can support, such as ftp, telnet, and so on, are inherently insecure, while others have had a troubled childhood and act out as a result (for example, popper). Treat inetd as you would any other network server program: do not run inetd unless you need it, and then confirm that it offers only trusted and secure programs!

/etc/inetd.conf

Take a look at */etc/inetd.conf*. Most daemons have separate IPv4 and IPv6 configurations, but if you're not running IPv6, you can ignore the IPv6 entries. Let's look at one entry, the FTP server configuration.

```
❶ftp    ❷stream    ❸tcp    ❹nowait    ❺root    ❻/usr/libexec/ftpd    ❼ftpd -
l
```

The first field is the service name ❶, which must match a name in */etc/services*. inetd performs a service name lookup to identify which TCP port it should listen to. If you want to change the TCP/IP port your FTP server runs on, change the port for FTP in */etc/services*. (You could also change the first field to match the service that runs on the desired port, but I find that this makes the entry slightly confusing.)

The socket type ❷ dictates what sort of connection this is. All TCP

connections are of type `stream`, while UDP connections are of type `dgram`. While you might find other possible values, if you're considering using them, either you're reading the documentation for a piece of software that tells you what to use, or you're just wrong.

The protocol ❸ is the layer 4 network protocol, either `tcp` (IPv4 TCP), `udp` (IPv4 UDP), `tcp6` (IPv6 TCP), or `udp6` (IPv6 UDP). If your server accepts both IPv4 and IPv6 connections, use the entries `tcp46` or `udp46`.

The next field indicates whether inetd should wait for the server program to close the connection or just start the program and go away ❹. As a general rule, TCP daemons use `nowait` while UDP daemons need `wait`. (There are exceptions to this, but they're rare.) inetd(8) starts a new instance of the network daemon for each incoming request. If a service uses `nowait`, you can control the maximum number of connections inetd accepts per second by adding a slash and a number directly after `nowait`, like this: `nowait/5`. One way intruders (usually script kiddies) try to knock servers off the internet is by opening more requests for a service than the server can handle. By rate-limiting incoming connections, you can stop this. On the other hand, this means that your intruder can stop other people from using the service at all. Choose your poison carefully!

We then have the user ❺ that the server daemon runs as. The FTP server ftpd(8) runs as root, as it must service requests for many system users, but other servers run as dedicated users.

The sixth field is the full path to the server program inetd runs when a connection request arrives ❻. Services integrated with inetd(8) appear as internal.

The last field gives the command to start the external program, including any desired command line arguments ❼.

Configuring inetd Servers

While */etc/inetd.conf* seems to use a lot of information, adding a program is actually pretty simple. The easiest way to learn about inetd(8) is to implement a simple service with it. For example, let's implement a Quote of the Day (qotd) service. When you connect to the qotd port, the server sends back a random quote and disconnects. FreeBSD includes a random quote

generator, fortune(1), in its games collection. This random quote generator is all we need to implement an inetd-based network program. We must specify a port number, a network protocol, a user, a path, and a command line.

port number

The */etc/services* file lists qotd on port 17.

network protocol

The qotd service requires that you connect to a network port and get something back, so it needs to run over TCP. Remember, UDP is connectionless—a reply isn't required. We must specify `tcp` in our inetd configuration, which means that we must specify `nowait` in the fourth field.

user

Best practice says to create an unprivileged user to run the qotd service, as discussed in Chapter 19. For this example, we'll just use the general unprivileged user nobody, but if you were implementing this in production, you'd want to create an unprivileged user qotd.

path

Find fortune at */usr/bin/fortune*.

Running the Command

fortune(6) doesn't require any command line arguments, but you can add them if you like.[4] On FreeBSD 11, believers in Murphy's Law can use `fortune murphy`, while *Star Trek* fans can get quotes with `fortune startrek`. (The latter correctly includes only the One True *Star Trek*, not any of the wannabe followups.) Those interested in education could use `fortune freebsd-tips`. FreeBSD 12 removes many of the fortune databases, sadly.

Sample inetd.conf Configuration

Putting this all together, the entry for qotd in */etc/inetd.conf* looks like this:

```
qotd    stream    tcp    nowait    nobody    /usr/bin/fortune    fortune
```

You might think this example trivial, but providing other services out of inetd(8) is no more difficult.

Starting inetd(8)

First, enable inetd(8) at boot by adding the following entry to */etc/rc.conf*:

```
inetd_enable=YES
```

With this set, start inetd by hand with `/etc/rc.d/inetd start`. Now that inetd is running, telnet to port 17 to test our new service:

```
   # telnet localhost 17
❶ Trying 127.0.0.1...
   Connected to localhost.
   Escape character is '^]'.
❷ It is difficult to produce a television documentary that is both
   incisive and probing when every twelve minutes one is interrupted by
   twelve dancing rabbits singing about toilet paper.
                  -- Rod Serling
   Connection closed by foreign host.
```

It works! We have the usual TCP/IP connection information ❶ and our random fortune ❷. (As an added bonus, you also know why I don't write for television.)

Changing inetd's Behavior

inetd behaves differently depending on the flags you set for it. The default flags turn on TCP wrappers, as configured in */etc/hosts.allow* (see Chapter 19). Table 20-2 lists some of the useful flags.

Table 20-2: inetd(8) Flags

Flag	Description
`-l`	Log every successful connection.
`-c`	Set the maximum number of connections per second that can be made to any service. By default, there's no limit. Note that "unlimited" isn't the same as "infinite"—your hardware only handles so many connections.
`-C`	Set the number of times one IP address can connect to a single service

Set the number of times one IP address can connect to a single service in one minute. This connection rate is unlimited by default, but using this can be useful against people trying to monopolize your bandwidth or resources.

`-R` Set the maximum number of times any one service can be started in one minute. The default is 256. If you use `-R 0`, you allow an unlimited number of connections to any one service.

`-a` Set the IP address inetd(8) attaches to. By default, inetd listens on all IP addresses attached to the system.

`-w` Use TCP wrappers for programs started by inetd(8), as per *hosts.allow* (see Chapter 19).

`-W` Use TCP wrappers for services integrated with inetd(8), as per *hosts.allow* (see Chapter 19).

As an extreme example, if you want to use TCP wrappers, allow only two connections per second from any single host, allow an unlimited number of service invocations per minute, and listen only on the IP address 203.0.113.2, then you'd set the following in */etc/rc.conf*:

```
inetd_flags="-Ww -c 2 -R 0 -a 203.0.113.2"
```

With inetd(8), almost anything can be a network service.

DHCP

Dynamic Host Configuration Protocol (DHCP) is the standard method for handing out IP addresses to client computers. While DHCP services aren't integrated with FreeBSD out of the box, they're commonly required to implement such services as diskless workstations. We'll cover the basics of DHCP configuration here so you can set up your own network.

These days, every firewall and embedded device has a DHCP server. Why would you need a separate DHCP server? Most of the embedded DHCP servers lack functions needed to run diskless clients, such as network-booted servers and VoIP phones. When they do support such functions, those DHCP servers are often difficult to manage. Services are meant to run on actual servers. We'll cover enough of DHCP to let you

configure your own network clients, including diskless hosts.

FreeBSD packages include several DHCP servers. The two I like are OpenBSD's dhcpd and ISC DHCP server. The *ISC DHCP server* is an industry standard and supports every feature you could possibly want. For small deployments, I recommend OpenBSD's *dhcpd*. The OpenBSD folks took ISC DHCP, ripped out all the rarely used features, and made a smaller, simpler server. The configuration file is still one-way compatible; you can run an OpenBSD dhcpd configuration on ISC's DHCP server without trouble. (The reverse is also true if you're not using any of the features OpenBSD ripped out of the server.) If you want to run diskless FreeBSD clients, or if you need LDAP integration, switching to the more complex ISC server is fairly straightforward. You can install only one of the two servers.

The package for either server includes dhcpd(8), the configuration file */usr/local/etc/dhcpd.conf*, and extensive man pages.

ROGUE DHCP SERVERS

Each network should have one and only one set of authoritative DHCP information. If you set up your own DHCP server on a network that already has one, such as in your company office, you'll probably break a whole bunch of clients and trigger a whole bunch of phone calls to the network team. Setting up a "rogue" DHCP server is a great way to have the network team ignore all of your help requests from now until forever.

How DHCP Works

DHCP can be terribly complicated in a large network where we are relaying DHCP requests between offices, but it's rather simple on a local Ethernet. Each DHCP client sends a broadcast across the local Ethernet asking for someone—anyone—to provide network configuration information. If your DHCP server is on that local Ethernet, it answers directly. If your DHCP

server is on another network segment, the router for that network segment needs to know which IP address to forward the DHCP request to. The DHCP server then loans configuration information to the client and tracks which clients have been assigned which IP addresses. A configuration issued to a client is called a *lease*. Like the lease you pay on a home or auto, DHCP leases expire and must be renewed occasionally.

The client can request certain features—for example, Microsoft clients ask for the IP address of the WINS server, while diskless systems ask where to find a kernel. You can set all these options as necessary.

Each client is uniquely identified by the MAC address of the network card used to connect to the network. ISC dhcpd tracks MAC and IP addresses, as well as leases, in the file */var/db/dhcpd.leases*. In this file, you can identify which hosts have which IP addresses. If a host disappears from the network for a time and returns, dhcpd(8) reissues the same IP to that client if that IP is still available.

Configuring dhcpd(8)

The file */usr/local/etc/dhcpd.conf* contains all the configuration for dhcpd. While ISC dhcpd(8) can and does fill entire books on its own, we'll focus on the functions needed for a basic small office as well as those used in the examples later in this book. The default *dhcpd.conf* is well commented and includes still more examples, while dhcpd.conf(5) is painfully exhaustive. We're going to assume that you're running a single DHCP server on your network, and that your server should answer all requests for DHCP services. (It's entirely possible to cluster dhcpd for fault tolerance, but that's beyond our scope here.)

Global Settings

Start your *dhcpd.conf* with a few general rules for client configuration. These rules apply to all DHCP clients unless specifically overridden.

```
❶ option domain-name "mwl.io";
❷ option domain-name-servers 198.51.100.2, 198.51.100.3;
❸ option subnet-mask 255.255.255.0;
❹ default-lease-time 600;
❺ max-lease-time 7200;
```

Each DHCP client registers its hostname with the DHCP server, but the client must learn the local domain name from the server. (It's also possible for the DHCP server to set the client's hostname.) Set this with the `domain-name` option ❶. You can give your DHCP clients any domain name you like; they don't need to share the server's domain name. You can include multiple domains if you separate them with spaces, but not all operating systems will recognize additional domain names.

Every TCP/IP client needs a DNS server or two. Specify them with the option `domain-name-servers` ❷. Separate multiple DNS servers with commas.

It's a good idea to set a default subnet mask ❸. Individual networks can override this, but a global default is useful.

The normal duration of a lease is given (in seconds) by the `default-lease-time` option ❹. After the lease time runs out, the client requests a new DHCP lease from the DHCP server. DHCP servers commonly default to a small number of minutes, but if your network is fairly stable you can extend this to hours or a couple days. If the client can't reach the DHCP server, it continues to use the old lease for a number of minutes equal to the maximum life of the lease, specified with `max-lease-time` ❺. You can think of the maximum lease time as "if my DHCP server fails, this is how long I have to replace it before the phone starts ringing." Give yourself time to fix the issue.[5]

Now define subnets.

Subnet Settings

Each subnet on your network needs a `subnet` statement to identify configuration information for DHCP clients on that subnet. For example, here's a network statement for a single small office network:

```
❶ subnet 198.51.100.0 netmask 255.255.255.0 {
  ❷ range 198.51.100.50 198.51.100.99;
  ❸ option routers 198.51.100.1;
  }
```

Each subnet declaration starts by identifying the network number and netmask ❶ of the subnet. Here, we have a subnet using the IP network

number 198.51.100.0 with the netmask 255.255.255.0, or the IP addresses 198.51.100.1 through 198.51.100.255. The information that follows in braces all pertains to hosts on that particular subnet.

The `range` keyword ❷ identifies the IP addresses that dhcpd(8) may issue to clients. In this example, we have 50 IP addresses available for clients. If 51 DHCP clients connect before any leases expire, the last host won't get an address.

Define a default route with the `routers` option ❸. Note that you can't define additional routes with dhcpd(8); instead, your local network router needs to have the proper routes to reach the destination. If you have multiple gateways on your local network, your gateway transmits an ICMP redirect to the DHCP client to give it an updated route. (If you have no idea what this means, that's all right. When you need it, you'll abruptly comprehend what I'm talking about, and if you never need it, you've just wasted the two seconds it took to read this aside.)

If you have multiple subnets, create multiple subnet statements. Some of those subnets might need settings different than the global defaults, such as a netmask or DNS servers. If so, use those same keywords to define those values for that subnet.

Dhcpd lets you set anything from the subnet mask, boot servers, and even WINS servers for antediluvian Windows clients. We'll use some of these less common settings to manage diskless clients in Chapter 23. See dhcpd.conf(5) for an exhaustive list.

Managing dhcpd(8)

Dhcpd defaults to listening to all network interfaces to catch DHCP request broadcasts. I've run many DHCP servers with multiple network cards, however, and usually want dhcpd to listen only to a single interface. Give the desired interface as a command line argument.

```
sysrc dhcpd_flags="em1"
```

Now enable dhcpd(8) itself.

```
sysrc dhcpd_enable=YES
```

You can now fire up dhcpd with `service dhcpd start`.

Congratulations, you're ready to go!

Printing and Print Servers

Printing on Unix-like operating systems is a topic that makes new sysadmins cry and seasoned sysadmins ramble on about the good old days when printers were TTY devices and about the younger generation not knowing how good they have it.[6] The most common printing situations are printers directly attached to a computer via a USB port and printers attached to a network print server.

If you have a printer attached directly to your FreeBSD machine, such as by a USB cable, I suggest using the *Common Unix Printing System (CUPS)*. This suite of software manages many popular consumer-grade and commercial printers, from lowly inkjets to web-scale laser printers. I'm not going into any detail about CUPS, as it's complicated and varies by printer model. Learn more about CUPS at *http://www.cups.org/*. Many brands of printers have special setup programs in CUPS, such as HP's `hp-setup`. If your printer supports a network connection, though, try to avoid CUPS and use network printing instead.

Accessing a remote print server or network printer via the *Line Printer Spooler Daemon (LPD)* is simple in comparison. LPD takes in PostScript and produces printouts. Most office print servers run LPD. The lpd(8) daemon manages LPD. Most modern networked printers also support LPD and can act as their own print server.

Test for LPD support by connecting to TCP port 515; if you get a connection, the device speaks LPD.

```
# nc -v color-printer 515
Connection to color-printer 515 port [tcp/printer] succeeded!
```

This device supports LPD. We can send print jobs to it by configuring */etc/printcap*.

/etc/printcap

Every printer your system knows about needs an entry in */etc/printcap*, the

printer capability database. This file is, by modern standards, in a rather obtuse format and will look very unfamiliar to anyone who hasn't previously worked with termcap(5). Fortunately, to access a print server you don't need to understand printcap(5); you just need to use the following template.

To connect to a printer on a print server, you must have the print server's hostname or IP address and its name for the printer you want to access. Make an entry in */etc/printcap* following this template. Pay special attention to the colons and backslashes—they're absolutely vital.

```
❶ lp|printername:\
      ❷ :sh=:\
      ❸ :rm=printservername:\
      ❹ :sd=/var/spool/output/lpd/printername:\
      ❺ :lf=/var/log/lpd-errs:\
      ❻ :rp=printername:
```

Our first line shows the printer's name ❶. If you print from LibreOffice or a graphical web browser, these names will show up as printer options. Each printer can have any number of names, separated by the pipe symbol (`|`). The default printer on any Unix-like system is called *lp*, so list that as one of the names for your preferred printer. One other name should be the name used by the print server for your printer (for example, *3rdFloorPrinter*). Be warned, Microsoft print servers frequently share one printer under several different names and use different names to handle printing differently. If you find this to be the case on your network, be sure to choose the PostScript name.[7]

By default, lpd(8) precedes each print job with a page listing the job name, number, host, and other information. Unless you're in an environment with a single massive shared printer, this is probably a waste of paper. The `:sh:\` entry ❷ suppresses this page.

The `rm` (remote machine) variable ❸ provides the hostname of the print server. You must be able to ping this server by the name you give here. If the print server is part of the printer, give the printer's hostname here.

Each printer requires a unique spool directory ❹, where the local print daemon can store documents in transit to the print server. This directory must be owned by user *root* and group *daemon*.

Unlike spool directories, which must be different, printers can share a common log file ❺.

Finally, specify the remote printer name ❻, as the print server identifies it. If you're connecting directly to a printer, not to a central print server, you can skip this entry—but you must get rid of the trailing slash on the previous line.

Be sure you end */etc/printcap* with a new line; don't just terminate the file immediately after the printer name. Also, note that unlike every other entry in this template, the last line doesn't require a trailing backslash.

Printers have dozens and dozens of options, from the cost per page to manually setting a string to feed a new sheet of paper. Most of these are obsolete today. If you have an older printer or special needs, though, consult printcap(5) for enough glorious detail to choke on.

Enabling LPD

Set `lpd_enable` to `YES` in */etc/rc.conf* to have lpd(8) start at boot. Any time you edit */etc/printcap* you must restart lpd(8). View the print queue with lpq(1) and watch for any problems in */var/log/lpd-errs*.

TFTP

Let's end our discussion of small network services with perhaps the smallest network service still used, the *Trivial File Transfer Protocol (TFTP)*. TFTP lets you transfer files from machine to machine without any authentication whatsoever. It's also much less flexible than file copy protocols, such as SCP or FTP. TFTP is still used by makers of embedded devices, such as Cisco, to load system configurations and operating system updates. We cover it here only because diskless clients use TFTP to download their operating system kernel and get their initial configuration information. Run tftpd(8) out of inetd(8) on TCP port 69.

TFTP SECURITY

TFTP isn't suitable for use on the public internet. Anyone

can read or write files on a TFTP server! Only use TFTP behind a firewall or at least protect it tightly with TCP wrappers (see Chapter 19).

Setting up a tftpd(8) server involves four steps: choosing a root directory for your server, creating files for the server, choosing an owner for your files, and running the server process.

Root Directory

The tftpd(8) daemon defaults to using the directory */tftpboot*. This might be suitable if you have only a couple files that you rarely access, but the root partition is best reserved for files that don't change often. You don't want a TFTP upload to crash your system by filling the root partition! If you're running ZFS, create a tftp dataset. On UFS, I usually put my tftpd(8) root directory in */var/tftpboot* and add a symlink to */tftpboot*:

```
# mkdir /var/tftpboot
# ln -s /var/tftpboot /tftpboot
```

Now you can create files for access via TFTP.

tftpd and Files

Users can both read and write files via TFTP. If you want tftpd(8) users to be able to read a file, the file must be world-readable:

```
# chmod +r /var/tftproot/filename
```

Similarly, tftpd(8) won't allow anyone to upload a file unless a file of that name already exists and is world-writable. Remember, programs and regular files have different permissions. A program must have execute permissions in addition to read and write permissions, so you must set permissions differently for programs and files. You can use touch(1) to precreate files that you'll want to upload via TFTP.

```
# chmod 666 /var/tftproot/filename
# chmod 777 /var/tftproot/programname
```

Yes, this means that anyone who knows a file's name can overwrite the contents of that file. Make vital files read-only.[8] This also means you don't have to worry about someone uploading a big file and filling your hard drive.

File Ownership

Files in a TFTP server should be owned by a user with the least possible privilege. If you run a TFTP server only intermittently, you can use the nobody user. For example, if you need the TFTP server only to perform the occasional embedded device upgrade, let the nobody user own your files and just turn tftpd(8) off when it's not needed. If you run a permanent TFTP server, however, it's best to have a dedicated tftp unprivileged user to own the files. The tftp user doesn't need to own the *tftproot* directory and, in fact, should have an entirely different home directory. He needs ownership only of the files available to users.

tftpd(8) Configuration

tftpd(8) is configured entirely through command line arguments, and there aren't many of them. For a full list, read tftpd(8), but here are the most commonly used ones.

If you create a user just to run tftpd(8), specify that user with the `-u` argument. If you don't specify a user, tftpd(8) runs as nobody. Create an unprivileged user.

I recommend logging all requests to your TFTP daemon. The `-l` argument turns on logging. tftpd(8) uses the `FTP` facility, which you must enable in *syslog.conf* (see Chapter 21).

Tftpd supports chrooting with the `-s` flag. This lets you confine tftpd(8) to your selected directory. You don't want users to TFTP world-readable files such as */etc/passwd*, or even */boot/kernel/kernel*, just on general principle! Always chroot your tftpd(8) installation.

You can chroot TFTP clients by IP address with the `-c` argument. In this case, you must create a directory for every client permitted to connect. For example, suppose the only host you want to give TFTP access to is your router, with the IP address of 192.168.1.1. You could create a directory */var/tftproot/192.168.1.1* and use `-c`. You must also use `-s` to define the base

directory of */var/tftproot*. This is a good compromise when you must offer TFTP to only one or two hosts, but you don't want the world to have access to your TFTP server.

You can choose to allow a client to write new files to your TFTP server. This is a bad idea because it lets remote users fill up your hard disks with arbitrary files. If you must have this functionality, use the `-w` flag.

For example, suppose you want to log all requests to tftpd, chroot to */var/tftpboot*, run the server as the user tftpd, and chroot clients by IP address. The command to run tftpd would look like this:

```
tftpd -l -u tftpd -c -s /var/tftpboot
```

Enter this into *inetd.conf* as described earlier this chapter, restart inetd(8), and you're in business!

Scheduling Tasks

The FreeBSD job scheduler, cron(8), allows the administrator to have the system run any command on a regular basis. Combined with the system maintenance scheduling system, periodic(8), you can schedule almost anything.

cron(8)

If you need to back up your database nightly or reload the nameserver four times a day, cron is your friend. cron(8) configuration files are called *crontabs* and are managed with crontab(1). Every user has a separate crontab stored in */var/cron/tabs*, and the global crontab file is */etc/crontab*. Global cron entries can also be placed in */etc/cron.d* and will be run as if they were part of */etc/crontab*.

User Crontabs vs. /etc/crontab

The purpose of */etc/crontab* is different from that of individual users' crontabs. With */etc/crontab*, root may specify which user will run a particular command. For example, in */etc/crontab*, the sysadmin can say, "Run this job at 10 PM Tuesdays as root, and run this other job at 7 AM as *www*." Other users can run jobs only as themselves. Of course, root can also edit a user's

crontab.

Also, any system user can view */etc/crontab*. If you have a scheduled job that you don't want users to know about, place it in a user crontab. For example, if you have an unprivileged user for your database, use that unprivileged user's crontab to run database maintenance jobs.

/etc/crontab is considered a FreeBSD system file. Don't overwrite it when you upgrade! One way to simplify upgrading */etc/crontab* is to set your custom entries at the end of the file, marked off with a few lines of hash marks (`#`). The */etc/crontab* file must end with a new line, or the last line won't get parsed and run. That's fine if your last entry is a comment, but not so good if it's a command.

Finally, while you edit */etc/crontab* with a text editor, edit a user crontab with `crontab -e`.

cron and Environment

Crontabs run in a shell, and programs might require environment variables to run correctly. You can also specify environment variables on the command line for each command you run from cron. cron doesn't inherit any environment variables from anywhere; any environment variables a program needs must be specified in the crontab. For example, here's the environment from */etc/crontab* on a FreeBSD 12 system:

```
SHELL=/bin/sh
PATH=/etc:/bin:/sbin:/usr/bin:/usr/sbin
```

Yes, this is extremely minimal! Feel free to add environment variables as needed to user crontabs, but be conservative when changing */etc/crontab*. If you need a custom environment variable, it's safest to use a user crontab rather than */etc/crontab* because many of the commands in */etc/crontab* are for core system maintenance.

Crontab Format

Beneath the environment statements, a user crontab is divided into six columns. The first five columns represent the time the command should run, as minute, hour, day of the month, month of the year, and day of the week, in that order. An asterisk (`*`) in any column means *every one*, while a number means *at this exact time*. Minutes, hours, and days of the week begin

with 0, and days of the month and months begin with 1. Also, thanks to an ancient disagreement between AT&T and BSD, Sunday can be represented by either 7 or 0. After the time, list the command to be run at that time.

The */etc/crontab* file, and files under */etc/cron.d*, have one extra column: the user under which to run the command. It goes between the time specification and the command itself. Check out the many examples in */etc/crontab* if you like.

Sample Crontabs

Assume that we're editing the crontab of an unprivileged user to schedule maintenance of a program. As */etc/crontab* has column headings at the top, we'll demonstrate user crontabs here. (To use these examples in */etc/crontab*, just add the user before the command.) Here, we want to run the program */usr/local/bin/maintenance.sh* at 55 minutes after each hour, every single hour:

```
55    *    *    *    *    /usr/local/bin/maintenance.sh
```

Asterisks tell cron to run this job every hour, on every day of the month, every month, and on every weekday. The 55 tells cron to run this job only at minute 55.

To run the same job at 1:55 PM every day, use the following:

```
55    13    *    *    *    /usr/local/bin/maintenance.sh
```

Here, 13 represents 1:00 PM on the 24-hour clock, and 55 is the number of minutes past that hour.

One common mistake people make when using cron is specifying a large unit of time but missing the small one. For example, suppose you want to run the job every day at 8 AM:

```
*    8    *    *    *    /usr/local/bin/maintenance.sh
```

This is wrong. Yes, the job will run at 8:00 AM. It will also run at 8:01, 8:02, 8:03, and so on, until 9 AM. If your job takes more than one minute to run, you'll quickly bring your system to its knees. The correct way to specify 8:00 AM, and only 8:00 AM, is this:

```
0    8    *    *    *    /usr/local/bin/maintenance.sh
```

To specify ranges of time, such as running the program once an hour, every hour, between 8 AM and 6 PM, Monday through Friday, use something like this:

```
55    8-18    *    *    1-5    /usr/local/bin/maintenance.sh
```

To specify multiple exact times, separate them with commas:

```
55    8,10,12,14,16    *    *    *    /usr/local/bin/maintenance.sh
```

More interestingly, you can specify fractions of time, or *steps*. For example, to run a program every 5 minutes, use:

```
*/5    *    *    *    *    /usr/local/bin/maintenance.sh
```

You can combine ranges with steps. To run the program every 5 minutes, but 1 minute after the previous example, use this:

```
1-56/5    *    *    *    *    /usr/local/bin/maintenance.sh
```

Control the day a job runs with two fields: the day of the month and the day of the week. If you specify both, the job will run whenever *either* condition is met. For example, tell cron to run a job on the 1st and the 15th of every month, plus every Monday, as follows:

```
55    13    *    1,15    1    /usr/local/bin/maintenance.sh
```

If your job has a nonstandard environment, set the environment on the command line just as you would in the shell. For example, if your program requires a `LD_LIBRARY_PATH` environment variable, you can set it thus:

```
55    *    *    *    *    LD_LIBRARY_PATH=/usr/local/mylibs ;
/usr/local/bin/maintenance.sh
```

cron also supports special scheduling, such as *annually* or *daily*, with the `@` symbol. Most of these terms are best not used, as they can be ambiguous. While the machine knows exactly what they mean, humans tend to misunderstand! One useful crontab entry is for *whenever the system boots*, which is `@reboot`. This lets an unprivileged user run jobs when the system boots. Use the `@reboot` label instead of the time fields:

```
@reboot    /usr/local/bin/maintenance.sh
```

Crontabs and cron(8) let you schedule your work any way you like, eliminating the human being from many routine maintenance tasks.

periodic(8)

Some system maintenance jobs should be run only on particular systems, but the way they should be run is identical across all hosts. That's where periodic(8) comes in.

The periodic(8) command runs system functions on schedule, as cron(8) determines. Periodic checks a directory for a set of scripts to run. FreeBSD includes several directories for periodic tasks: */etc/periodic/daily*, */etc/periodic/weekly*, */etc/periodic/monthly*, and */etc/periodic/security*. Depending on which packages you install, you might have corresponding directories in */usr/local/etc/periodic*. When cron runs, say, periodic daily, periodic(8) checks each script in each *periodic/daily* directory to see whether it should be run.

When you have spare time, I recommend perusing the periodic(8) scripts. You might find disabled maintenance scripts useful for your environment.

Which scripts should be run? The default settings are listed in */etc/defaults/periodic.conf*, but you can override them in */etc/periodic.conf*.

Once periodic(8) runs, it mails the results of the scripts to root on the local machine. Forward root's mail to someone who will actually read it.

Why use periodic(8)? It's all for system maintenance. */etc/crontab* is for configuring your own system administration jobs. Using separate scripts allows the system upgrade process to replace tasks and packages to add and remove them.

All periodic(8) jobs run as root, though. If you have scheduled jobs that should be run by less privileged users, run them from the user's crontab.

Now that you have a decent understanding of the common small services provided by FreeBSD, let's go on to performance.

21
SYSTEM PERFORMANCE AND MONITORING

Even if "it's slow!" isn't the most dreaded phrase a system administrator can hear, it's pretty far up on the list. The user doesn't know why the system is slow and probably can't even quantify or qualify the problem any further than that. It just *feels* slow. Usually there's no test case, no set of reproducible steps, and nothing particularly wrong. A slowness complaint can cause hours of work as you dig through the system trying to find problems that might or might not even exist.

One phrase is more dreadful still, especially after you've invested those hours of work: "it's still slow."

An inexperienced sysadmin accelerates slow systems by buying faster hardware. This exchanges "speed problems" for costly parts and even more expensive time. Upgrades just let you conceal problems without actually using the hardware you already own, and sometimes they don't even solve the problem at all.

You can frequently solve performance problems by tweaking the software that's causing the problems. Your WordPress site is slow? Investigate running PHP under memcached or another PHP accelerator. FreeBSD is only one layer of your application stack, so be sure to give the other layers proper attention.

FreeBSD includes many tools designed to help you examine system performance and provide the information necessary to learn what's actually slowing things down. Some of them, such as dtrace(1), are highly complicated and require extensive knowledge of the system, the software, and a book of their own. Once you understand where a problem is,

identifying the solution to the problem becomes much simpler. You might actually need faster hardware, but sometimes shifting system load or reconfiguring software might solve the problem at much less expense. In either case, the first step is understanding the problem.

Computer Resources

Performance problems are usually caused by running more tasks than the computer can handle. That seems obvious, but think about it a moment. What does that really mean?

A computer has four basic resources: input/output, network bandwidth, memory, and CPU. If any one of them is filled to capacity, the others can't be used to their maximum. For example, your CPU might very well be waiting for a disk to deliver data or for a network packet to arrive. If you upgrade your CPU to make your system faster, you'll be disappointed. Buying a whole new server might fix the problem, but only by expanding the existing bottleneck. The new system probably has more memory, faster disks, a better network card, and faster processors than the old one. You have deferred the problem until the performance reaches some new limit. However, by identifying where your system falls short and addressing that particular need, you can stretch your existing hardware much further. After all, why purchase a whole new system when a few gigabytes of relatively inexpensive memory would fix the problem? (Of course, if your goal is to retire this "slow" system to make it your new desktop, that's another matter.)

Input/output is a common bottleneck. System busses have a maximum throughput, and while you might not be pushing your disk or your network to their limits, you might be saturating the bus by continually bombarding both.

One common cause of system slowdowns is running multiple large programs simultaneously. Not only does disk I/O become saturated, but the processors might spend the majority of their time waiting to swap data between the on-CPU cache and the memory. For example, I once thoughtlessly scheduled a massive database log rotation that moved and compressed gigabytes of data at the same time as the daily periodic(8) run. Since the job required shutting down the main database and caused application downtime, speed was crucial. Both the database job and the

periodic(8) run slowed unbearably. Rescheduling one of them made both jobs go more quickly.

FreeBSD has some features that improve performance. Doing lots of cryptographic operations? Use the aesni(4) kernel module. Database is disk bound? Consider the filesystem block size. ZFS pool slow? Maybe you need an add-on cache. Identifying what you should change requires a hard look at the system, however.

We're going to look at several FreeBSD tools for examining system performance. Armed with that information, we'll consider how to fix performance issues. Each potential bottleneck can be evaluated with the proper tools. FreeBSD changes continually, so later systems might have new tuning options and performance features. Read tuning(7) on your system for current performance tips.

WHAT IS NORMAL?

One word you'll keep tripping over in this chapter is *abnormal*. As the sysadmin, you're supposed to know what's normal for your system. It's somewhat like art; you might not be able to define *normal*, but you need to recognize *abnormal* when you see it. Use these tools regularly when the system is behaving itself so you can have a good idea of which results are out of whack during system slowdowns. Pay attention to your hardware!

Checking the Network

If you're concerned about network performance, measure it. Consult `netstat -m` and `netstat -s`, and look for errors or places where you're out of memory or buffers. These are instantaneous snapshots, but for the network, you really need to evaluate congestion and latency over minutes, hours, and even days. The network team probably has a tool like Cacti, Zabbix, or Graphite to observe long-term performance.[1] Ask them for information. Combine what these tools provide with your instantaneous snapshots. If the average

throughput per minute on your 10-gig Ethernet is only 5 gigabit a second, but your instantaneous measurements show frequent spikes up to the full 10 gigabit, you probably have really bursty connectivity.

Some network cards can better handle a full network in *polling mode*. Polling tells the network card to stop sending frames up to the operating system as they arrive and instead let the operating system visit every so often to collect the frames. Check your network card's man page to see whether it supports polling. Enable and disable polling with ifconfig(8).

A heavily loaded network might benefit from a different congestion control algorithm. FreeBSD provides several TCP congestion control algorithms. Look for files beginning with *cc_* in */boot/kernel*; these are congestion control modules. Each has a man page.

View the currently loaded congestion control algorithms with the sysctl `net.inet.tcp.cc.available`.

```
# sysctl net.inet.tcp.cc
net.inet.tcp.cc.available: newreno
```

New Reno is the traditional congestion control algorithm. The congestion control kernel modules on this system include CDG, CHD, CUBIC, DCTCP, HD, H-TCP, and Vegas. The H-TCP algorithm is specifically designed for long-distance, high-bandwidth applications. Let's enable it.

```
# kldload /boot/kernel/cc_htcp.ko
# sysctl net.inet.tcp.cc.available
net.inet.tcp.cc.available: newreno, htcp
```

We now have H-TCP available in the kernel. Enable it with the `net.inet.tcp.cc.algorithm` sysctl.

```
# sysctl net.inet.tcp.cc.algorithm=htcp
net.inet.tcp.cc.algorithm: newreno -> htcp
```

Ultimately, you can't fit 10 pounds of bandwidth in a 5-pound circuit. If your saturated Ethernet is crippling your applications, turn off unnecessary network services or add more bandwidth.

Other system conditions are much more complicated. Start by checking where the problem lies with vmstat(8).

General Bottleneck Analysis with vmstat(8)

FreeBSD includes several programs for examining system performance. Among these are vmstat(8), iostat(8), and systat(1). We'll discuss vmstat(8) because I find it most helpful; iostat(8) is similar to vmstat(8), and systat(1) provides the same information in an ASCII graphical format.

Use vmstat(8) to see the system's current virtual memory statistics. While the output takes getting used to, vmstat(8) is very good at showing large amounts of data in a small space. Type **vmstat** at the command prompt and follow along.

```
# vmstat
procs  memory      page                    disks        faults     cpu
r b w  avm   fre   flt  re  pi  po    fr   sr ad0 ad1   in    sy    cs us sy id
8 0 0 1.3G   26G   157   0   1   0   172    1   0   0   12   212   149  0  0 100
```

The vmstat divides its display into six sections: process (`procs`), `memory`, paging (`page`), `disks`, `faults`, and `cpu`. We'll look at all of them quickly and then discuss in detail those parts that are the most important for investigating performance issues. This single line represents the average values for the whole time the system has been running. We'll get more real-time data in the next section.

Processes

vmstat(8) has three columns under the `procs` heading. Technically, vmstat counts threads rather than processes. Unthreaded applications have one thread per process, but your multithreaded application could have far, far more.

r The number of runnable threads that are waiting for CPU time, including all running processes. One thread per CPU is fine; it means your hardware is fully utilized. More than that means your CPU is a bottleneck. Some programs demand all the processor the host has and more, though; check that you're not running such a remorseless compute suck.

b The number of threads that are blocked waiting for system input or output—generally, waiting for disk access. These threads will run as soon as they get their data. If this number is high, your disk is the bottleneck.

`w` The number of threads that are runnable but are entirely swapped out. If you regularly have processes swapped out, the system's memory is inadequate for the host's workload.

This host has averaged eight runnable threads since boot, but zero waiting on I/O or memory. If you're getting complaints that this host is slow, the first place to check is processor utilization. Is someone, say, building FreeBSD from source just to generate interesting output for a book's performance chapter, while real people are attempting to do their jobs on the same system?

Memory

FreeBSD breaks memory up into uniform-sized chunks called *pages*. When a program requests memory, it gets assigned a number of pages. The size of a page is hardware- and OS-dependent but appears in the `hw.pagesize` sysctl. On FreeBSD's i386 and amd64 platforms, a page is 4KB. The system treats each page as a whole—if FreeBSD must shift memory into swap, for example, it does that on a page-by-page basis. The kernel thread that manages memory is called the *pagedaemon*. The `memory` section has two columns.

`avm` The average number of pages of virtual memory that are in use. If this value is abnormally high or increasing, your system is actively consuming swap space.

`fre` The number of memory pages available for use. If this value is abnormally low, you have a memory shortage.

Our sample output is using 1.3GB of RAM and has 26GB free. Memory isn't an issue.

Paging

The `page` section shows how hard the virtual memory system is working. The inner workings of the virtual memory system are an arcane science that I won't describe in detail here.[2]

`flt` The number of page faults, where information needed wasn't in real memory and had to be fetched from swap space or disk.

re The number of pages that have been reclaimed or reused from cache.

pi Short for *pages in*; this is the number of pages being moved from real memory to swap.

po Short for *pages out*; this is the number of pages being moved from swap to real memory.

fr How many pages are freed per second.

sr How many pages are scanned per second.

Moving memory into swap isn't bad, but consistently recovering paged-out memory indicates a memory shortage. Having high `fr` and `flt` values can indicate lots of short-lived processes—for example, a script that starts many other processes or a cron job scheduled too frequently. Or perhaps someone's been running `make -j16 buildworld`. A high `sr` probably means you don't have enough memory, as the pagedaemon is constantly trying to free memory. The paging daemon normally runs once a minute or so, but a high `sr` count means you're probably trying to do more work than your RAM can hold.

Disks

The `disks` section shows each of your disks by device name. The number shown is the number of disk operations per second, a valuable clue to determining how well your disks are handling their load. You should divide your disk operations between different disks whenever possible and arrange them on different buses when you can. If one disk is obviously busier than the others, and the system has operations waiting for disk access, consider moving some frequently accessed files from one disk to another. One common cause of high disk load is a coredumping program that can restart itself. For example, a faulty CGI script that dumps core every time someone clicks on a link will greatly increase your disk load.

If you have a lot of disks, you might notice that they don't all appear on the vmstat display. Designed for an 80-column display, vmstat(8) can't list every disk on a large system. If, however, you have a wider display and don't mind exceeding the 80-column limit, use the `-n` flag to set the number of drives you want to display.

Faults

Faults aren't inherently bad; they're just received system traps and interrupts. An abnormally large number of faults is bad, of course—but before you tackle this problem, you need to know what's normal for your system.

The first line of vmstat output shows the average faults per second since system boot.

in The number of system interrupts (IRQ requests) received

sy The number of system calls

cs The number of context switches in the last second, or a per-second average since the last update. (For example, if you have vmstat update its display every five seconds, this column displays the average number of context switches per second over the last five seconds.)

This host has averages 12 system calls and 212 context switches per second since boot. How does that compare to what you saw when the system was working normally?

CPU

Finally, the `cpu` section shows how much time the system spent doing user tasks (`us`), system tasks (`sy`), and how much time it spent idle (`id`). top(1) presents this same information in a friendlier format, but only for the current time, whereas vmstat lets you view system utilization over time.

Using vmstat

So, how do you use all this information? Start by checking the first three columns to see what the system is waiting for. If you're waiting for CPU access (the `r` column), then you're short on CPU horsepower. If you're waiting for disk access (the `b` column), then your disks are the bottleneck. If you're swapping (the `w` column), you're short on memory. Use the other columns to explore these three types of shortages in more detail.

Continuous vmstat

You're probably more interested in what's happening over time, rather than in a brief snapshot of system performance. Use the `-w` flag and a number to run it as an ongoing display updating every so many seconds. FreeBSD shows average values since the last update, updating counters continuously:

```
# vmstat -w 5
procs  memory       page                    disks     faults         cpu
r b w  avm   fre   flt  re  pi  po    fr   sr ad0 ad1   in    sy    cs us sy id
8 0 0 1.6G   25G   415   0   1   0   432    6   0   0   12   281   157  1  0 99
8 0 0 2.4G   24G 53089   0   7   0 11188  561  11   8   45  8789   994 96  4  0
8 0 0 2.5G   24G 44600   0   3   0 38703  741  10   9   49  8806  1032 96  3  1
8 0 0 2.2G   24G 42841   0  15   0 58044  717  11   9   52 10271  1103 96  4  0
--snip--
```

The first line still shows the averages since boot. Every five seconds, however, an updated line appears at the end. You can sit there and watch how your system's performance changes when scheduled jobs kick off or when you start particular programs. Hit CTRL-C when you're done. In this example, processes are always waiting for CPU time (as shown by the stack of `8`s in the `r` column), and we frequently have something waiting for disk access.

An occasional wait for a system resource doesn't mean you must upgrade your hardware; if performance is acceptable, don't worry about it. Otherwise, however, you must look further. The most common culprit is the storage system.

Disk I/O

Disk speed is a common performance bottleneck, especially with spinning disks, but even flash-based storage can get slow. Programs that must repeatedly wait for disk activity to complete run more slowly. This is commonly called *blocking on disk*, meaning that the disk is preventing program activity. The only real solution for this is to use faster disks, install more disks, or reschedule the load.

While FreeBSD provides several tools to check disk activity, my favorite is gstat(8), so we'll use that. You can run `gstat` without arguments for a display of all of your disks and partitions that updates every second or so. If you have many disks this can generate a whole bunch of zeros, though. I always use the `-a` flag, so that gstat(8) displays only disks with activity. The `-p`

flag is also useful, to view entire disks, but I prefer a per-partition view.

```
# gstat -a
dT: 1.002s  w: 1.000s
 L(q)  ops/s    r/s   kBps ❶ms/r    w/s   kBps ❷ms/w   %busy Name
   0    120      0      0    0.0    118    331    0.1   12.1| ada1
   0    120      0      0    0.0    118    331    0.1   12.1| ada1p1
   0     21      0      0    0.0     19    351    0.4    8.2| da1
   0     20      0      0    0.0     18    331    0.1   12.1| gpt/zfs4
   0     21      0      0    0.0     19    351    0.4    8.2| da1p1
   0     21      0      0    0.0     19    351    0.4    8.2| gpt/zfs7
```

We get a line for each disk device, slice, and partition, and various information for each. gstat(8) shows all sorts of good stuff, such as the number of reads per second (`r/s`), writes per second (`w/s`), the kilobytes per second of reading and writing, as well as a friendly-looking `%busy` column.

Ignore most of these. Some of these, such as the percent busy column, use sloppy measuring methods. The FreeBSD developers chose disk performance over accuracy of statistical measurements. What does matter, however, are the ms/r (milliseconds per read) ❶ and ms/w (milliseconds per write) ❷. These numbers are accurate. Measure and monitor them. If one disk has really high activity, but another is idle, consider dividing what's on that disk between multiple disks or using striped storage. Or, if it's your laptop, consider accepting that this is as fast as your storage system gets.

Once you identify the scarce system resource, you need to figure out what program's draining that resource. We'll need other tools for that.

CPU, Memory, and I/O with top(1)

The top(1) tool provides a decent overview of system status, displaying information about CPU, memory, and disk usage. Just type `top` to get a full-screen display of system performance data. The display updates every two seconds, so you have a close to real-time system view. Even if you update the update interval to one second, you can still miss short-lived, resource-sucking processes.

The output of top(1) is split into two halves. The upper portion gives basic system information, while the bottom gives per-process data.

```
❶last pid: 84111;  ❷load averages:  0.09,  0.21,  0.20                ❸up
```

```
7+07:58:00  14:41:09
❹28 processes:  2 running, 26 sleeping
❺CPU:  0.0% user,  0.0% nice,  0.9% system,  0.0% interrupt, 99.1% idle
❻Mem: 80M Active, 642M Inact, 124M Laundry, 222M Wired, 17M Free
❼Swap: 1024M Total, 83M Used, 941M Free, 8% Inuse

  PID USERNAME       THR PRI NICE   SIZE    RES STATE    TIME    WCPU COMMAND
  479 bind             4  20    0 99444K 35956K kqread   6:55   0.00% named
  586 root             1  20    0   154M 33768K select   4:54   0.00% perl
  562 root             1  20    0 22036K 13948K select   1:27   0.00% ntpd
--snip--
```

Very tightly packed, isn't it? The top(1) tool crams as much data as possible into a standard 80 × 25 terminal window or X terminal. Let's take this apart and learn how to read it. We'll start with the upper part, which can look a little different depending on whether you're using UFS or ZFS.

UFS and top(1)

The per-host information at in the upper part of top(1) varies slightly between ZFS and UFS hosts, but we'll start with UFS and then explain the differences.

PID Values

Every process on a Unix machine has a unique process ID (PID). Whenever a new process starts, the kernel assigns it a PID one greater than the previous process. The last PID value is the last process ID assigned by the system. In the previous example, our last PID is 84,111 ❶. The next process created will be 84,112, then 84,113, and so on. Watch this number to see how fast the system changes. If the system is running through PIDs more quickly than usual, it might indicate a process forking beyond control or something crashing and restarting.

Load Average

The *load average* ❷ is a somewhat vague number that offers a rough impression of the amount of CPU load on the system. The load average is the average number of threads waiting for CPU time. (Other operating systems have different load average calculation methods.) An acceptable load average depends on your system. If the numbers are abnormally high, you

need to investigate. Some hosts feel bogged down at a load average of 3, while some modern systems are still snappy with what look like ridiculously high load averages. Again, what's normal for *this* host?

You'll see three load averages. The first (0.09 here) is the load average over the last minute, the second (0.21) is for the last five minutes, and the last (0.20) is for the last 15 minutes. If your 15-minute load average is high, but the 1-minute average is low, you had a major activity spike that has since subsided. On the other hand, if the 15-minute value is low but the 1-minute average is high, something happened within the last 60 seconds and might still be going on now. If all of the load averages are high, the condition has persisted for at least 15 minutes.

Uptime

The last entry on the first line is the *uptime* ❸, or how long the system has been running. This system has been running for 7 days, 7 hours, and 58 minutes, and the current time is 14:41:09. I'll leave it up to you to calculate what time I booted this system.

Process Counts

On the second line, you'll find information about the processes currently running on the system ❹. Running processes are actually doing work—they're answering user requests, processing mail, or doing whatever your system does. Sleeping processes are waiting for input from one source or another; they're just fine. You should expect a fairly large number of sleeping processes at any time. Processes in other states are usually waiting for a resource to become available or are hung in some way. Large numbers of nonsleeping, nonrunning processes hint at trouble. The ps(1) command can show the state of all processes.

Process Types

The `CPU states` line ❺ indicates what percentage of available CPU time the system spends handling different types of processes. It shows five different process types: `user`, `nice`, `system`, `interrupt`, and `idle`.

The `user` processes are average everyday programs—perhaps daemons run by root, or commands run by regular users, or whatever. If it shows up in `ps`

`-ax`, it's a user process.

The `nice` processes are user processes whose priority has been deliberately manipulated. We'll look at this in detail in "Reprioritizing with Niceness" on page 543.

The `system` value gives the total percentage of CPU time spent by FreeBSD running kernel processes and the userland processes in the kernel. These include things such as virtual memory handling, networking, writing to disk, debugging with INVARIANTS and WITNESS, and so on.

The `interrupt` value shows how much time the system spends handling interrupt requests (IRQs).

Last, the `idle` entry shows how much time the system spends doing nothing. If your CPU regularly has a very low idle time, you might want to think about rescheduling jobs or getting a faster processor.

TOP AND SMP

On an SMP system, top(1) displays the average use among all the processors. You might have one processor completely tied up compiling something, but if the other processor is idle, top(1) shows the CPU usage of only 50 percent. Use the `-p` flag to view per-CPU stats.

Memory

The `Mem` line ❻ represents the usage of physical RAM. FreeBSD breaks memory usage into several different categories.

Active memory is the total amount of memory in use by user processes. When a program ends, the memory it had used is placed into *inactive memory*. If the system runs this program again, it can retrieve the software from memory instead of disk.

Free memory is totally unused. It might be memory that has never been accessed, or it might be memory released by a process. This system has 17MB of free RAM. If you have a server that's been up for months, and it still has free memory, you might consider putting some of that RAM in a

machine that's hurting for memory.

Memory in the *Laundry* is queued to be synchronized with other storage, such as disk.

FreeBSD 11 shuffles memory between the inactive, laundry, and free categories as needed to maintain a pool of available memory. Memory in the inactive is most easily transferred to the free pool. When cache memory gets low and FreeBSD needs still more free memory, it picks pages from the inactive pool, verifies that it can use them as free memory, and moves them to the free pool. FreeBSD tries to keep the total number of free pages above the sysctl `vm.v_free_target`.

FreeBSD 12 has no cache and handles low memory situations a little differently. When free memory gets low, the pagedaemon picks pages from the inactive pool. If that inactive page needs to be synced to disk, it's placed on the laundry queue, and the pagedaemon tries another inactive page. One way to test whether a host needs more RAM is if the pagedaemon is accumulating CPU time from all this testing.

On either FreeBSD version, having free memory doesn't mean that your system has enough memory. If vmstat(8) shows that you're swapping at all, you're using more physical memory than you have. You might have a program that releases memory on a regular basis. Also, FreeBSD will move some pages from inactive to free in an effort to maintain a certain level of free memory.

FreeBSD uses *wired* memory for in-kernel data structures, as well as for system calls that must have a particular piece of memory immediately available. Wired memory is never swapped or paged. All memory used by ZFS is wired.

Swap

The `Swap` line ❼ gives the total swap available on the system and how much is in use. Swapping is using the disk drive as additional memory. We'll look at swap in more detail later in the chapter.

ZFS and top(1)

The output of top(1) on a ZFS system looks superficially different, but the per-host handling of memory has important differences.

```
  last pid: 53202;  load averages:  0.26,  0.28,  0.30      up 1+15:41:48
  13:50:54
  120 processes: 1 running, 119 sleeping
  CPU:  0.1% user,  0.0% nice,  0.0% system,  0.0% interrupt, 99.9% idle
❶ Mem: 288M Active, 205M Inact, 3299M Wired, 137M Free
❷ ARC: 2312M Total, 458M MFU, 1626M MRU, 420K Anon, 38M Header, 189M Other
❸      1918M Compressed, 8885M Uncompressed, 4.63:1 Ratio
  Swap: 2048M Total, 126M Used, 1922M Free, 6% Inuse

    PID USERNAME    THR PRI NICE   SIZE    RES STATE   C   TIME    WCPU COMMAND
  53202 mwlucas       1  20    0 20124K  3388K CPU0    0   0:00   0.08% top
    835 mysql        26  23    0   629M   219M select  1  62:42   0.03% mysqld
  53151 www           1  20    0   237M 12924K select  2   0:00   0.03% httpd
    863 nobody        7  20    0 34928K  4960K kqread  0   0:31   0.02% memcached
  53058 www           1  20    0   239M 13296K lockf   0   0:00   0.01% httpd
    852 root          1  20    0   166M 11716K kqread  2   0:05   0.01% php-fpm
  --snip--
```

The Mem section ❶ lists `Active`, `Inactive`, `Laundry`, `Wired`, and `Free` memory familiar from UFS output.

The ARC line ❷ represents ZFS's *Advanced Replacement Cache*. The `Total` field shows the amount of memory the entire ARC uses. Within the 2,312MB used by the cache, 458MB are in the Most Frequently Used (`MFU`) cache, while 1,626MB are in the Most Recently Used (`MRU`) cache. You'll also see much smaller entries for ZFS internal data structures, such as anonymous buffers (`Anon`), ZFS headers (`Header`), and the ever-useful `Other`.

ZFS compresses the ARC ❸, exchanging plentiful CPU time for scarce memory. You can see the amount of space used by compressed and uncompressed cached data.

ZFS is greedy for memory, provided nothing else wants it. ZFS aggressively caches data read from and written to disk. This host has 4,096MB of RAM, and ZFS has claimed 2,312MB of that. You'll see that this host has only 137MB free. If a program requests memory and the system doesn't have it available, ZFS will release some of its cache back to the system. If you see a high wired memory level, remind yourself that all memory claimed by ZFS goes into the "wired" bucket.

This is a long-winded way of saying, "Don't let apparent high ZFS memory usage worry you." Worry only if the host starts paging and swapping.

More interesting is the list of processes that are using that memory.

Process List

Finally, top(1) lists the processes on the system and their basic characteristics. The table format is designed to present as much information as possible in as little space as possible. Every process has its own line.

PID First, we have the process ID number, or PID. Every running process has its own unique PID. When you use kill(1), specify the process by its PID. (If you don't know the PID of a process, you can use pkill(1) to kill the process by its name.)

Username Next is the username of the user running the process. If multiple processes consume large amounts of CPU or memory, and they're all owned by the same user, you know whom to talk to.

Priority and niceness The `PRI` (priority) and `NICE` columns are interrelated and indicate how much precedence the system gives each process. We'll talk about priority and niceness a little later in this chapter.

Size `SIZE` gives the amount of memory that the process has requested.

Resident memory The `RES` column shows how much of a program is actually in memory at the moment. A program might request a huge amount of memory but use only a small fraction of that at any time. The kernel is smart enough to give programs what they need rather than what they ask for.

State The `STATE` column shows what a process is doing at the moment. A process can be in a variety of states—waiting for input, sleeping until something wakes it, actively running, and so on. You can see the name of the event a process is waiting on, such as `select`, `pause`, or `ttyin`. On an SMP system, when a process runs, you'll see the CPU it's running on.

Time The `TIME` column shows the total amount of CPU time the process has consumed.

WCPU The weighted CPU (WCPU) usage shows the percentage of CPU time that the process uses, adjusted for the process's priority and niceness.

Command Finally, we have the name of the program that's running.

Looking at top(1)'s output gives you an idea of where the system is

spending its time.

Not every process on a host is actively engaged in work. You might have dozens or hundreds of daemons sitting idle. Enter `i` on a running top(1) display to toggle displaying idle processes, or use the `-i` command line flag. To show individual threads, either toggle `H` or add the `-H` flag.

By default, top sorts its output by weighted CPU usage. You can also sort output by priority, size, and resident memory. Enter `o` at a running top display. Enter the name of the column you want to sort by. This will help identify self-important programs or those using too much memory.

top(1) and I/O

In addition to the standard CPU display, top(1) has an I/O mode that displays which processes are using the disk most actively. While top(1) is running, hit `m` to enter the I/O mode. The upper portion of the display still shows memory, swap, and CPU status, but the lower portion changes considerably.

```
  PID USERNAME     VCSW  IVCSW   READ  WRITE  FAULT  TOTAL PERCENT COMMAND
 3064 root           89      0     89      0      0     89 100.00% tcsh
  767 root            0      0      0      0      0      0   0.00% nfsd
 1082 mwlucas         2      1      0      0      0      0   0.00% sshd
 1092 root            0      0      0      0      0      0   0.00% tcsh
  904 root            0      0      0      0      0      0   0.00% sendmail
--snip--
```

The `PID` is the process ID, of course, and the `USERNAME` column shows who is running the process.

`VCSW` stands for *voluntary context switches*; this is the number of times this process has surrendered the system to other processes. `IVCSW` means *involuntary context switches* and shows how often the kernel has told the process, "You're done now. It's time to let someone else run for a while."

Similarly, `READ` and `WRITE` show how many times the system has read from disk and written to disk. The `FAULT` column shows how often this process has had to pull memory pages from disk, which makes for another sort of disk read. These last three columns are aggregated in the `TOTAL` column.

The `PERCENT` column shows what percent of disk activity this process is using. Unlike gstat(8), top(1) displays each process's utilization as a percentage of the actual disk activity, rather than the possible disk activity. If

you have only one process accessing the disk, top(1) displays that process as using 100 percent of disk activity, even if it's sending only a trickle of data. While gstat(8) tells you how busy the disk is, top(1) tells you what's generating that disk activity and where to place the blame. Here, we see that process ID 3064 is generating all of our disk activity. It's a tcsh(1) process, also known as "some user's shell." Let's track down the miscreant.

MORE TOP FEATURES

The top(1) tool can alter its display in many ways. You can view processes for a particular user, include or exclude kernel threads, exclude idle processes, and so on. Read the man page for details.

Following Processes

On any Unix-like system, every userland process has a parent-child relationship with other processes. When FreeBSD boots, it creates a single process by starting init(8) and assigning it PID 1. This process starts other processes, such as the */etc/rc* startup script and the getty(8) program that handles your login request. These processes are children of process ID 1. When you log in, getty(8) starts login(8), which fires up a new shell for you, making your shell a child of the login(8) process. Commands you run are either children of your shell process or part of your shell. You can view these parent-child relationships with ps(1) using the `-ajx` flags (among others).

```
# ps -ajx
USER      PID  PPID  PGID  SID JOBC STAT TT         TIME COMMAND
root        0     0     0    0    0 DLs   -      6:26.23 [kernel]
root        1     0     1    1    0 ILs   -      0:00.09 /sbin/init --
root        2     0     0    0    0 DL    -      0:00.00 [crypto]
--snip--
root      845     1   845  845    0 Is    -      0:00.00 /usr/sbin/sshd
root      849     1   849  849    0 Ss    -      0:01.05 /usr/sbin/cron -s
root     8632   845  8632 8632    0 Is    -      0:00.09 sshd: mwlucas [priv] (sshd)
mwlucas  8634  8632  8632 8632    0 S     -      0:00.53 sshd: mwlucas@pts/0,pts/1
(sshd)
mwlucas  8687     1  8687 8687    0 Ss    -      0:25.90 tmux: server (/tmp/tmux-
1001/default) (tmux)
```

```
--snip--
```

At the far left, we have the username of the process owner and then the PID and parent PID (PPID) of the process. This is the most useful thing we see here, but we'll briefly cover the other fields.

The PGID is the process group ID number, which is normally inherited from its parent process. A program can start a new process group, and that new process group will have a PGID equal to the process ID. Process groups are used for signal processing and job control. A session ID, or SID, is a grouping of PGIDs, usually started by a single user or daemon. Processes may not migrate from one SID to another. JOBC gives the job control count, indicating whether the process is running under job control (that is, in the background).

STAT shows the process state—exactly what the process is doing at the moment you run ps(1). Process state is very useful as it tells you whether a process is idle, what it's waiting for, and so on. I highly recommend reading the section on process state from ps(1).

TT lists the process's controlling terminal. This column shows only the end of the terminal name, such as v0 for ttyv0 or p0 for ttyp0. Processes without a controlling terminal are indicated by ??.

The TIME column shows how much processor time the process has used, both in userland and in the kernel.

Finally, we see the COMMAND name, as it was called by the parent process. Processes in square brackets are actually kernel threads, not real processes. FreeBSD runs a whole bunch of kernel threads.

So, how can this help us track a questionable process? In our top(1) I/O example, we saw that process 3064 was generating almost all of our disk activity. Run **ps -ajx** to look for this process:

```
   USER      PID  PPID  PGID  SID JOBC STAT TT         TIME COMMAND
   --snip--
   root     3035  3034  3035  2969    1 S+    p0    0:00.03 _su -m (tcsh)
❶  bert     2981  2980  2981  2981    0 Is    p1    0:00.03 -tcsh (tcsh)
❷  root     2989  2981  2989  2981    1 I     p1    0:00.01 su -m
❸  root     2990  2989  2990  2981    1 D     p1    0:00.05 _su -m (tcsh)
❹  root     3064  2990  3064  2981    1 DV+   p1    0:00.15 _su -m (tcsh)
   mwlucas  2996  2995  2996  2996    0 Is    p2    0:00.02 -tcsh (tcsh)
   --snip--
```

Our process of interest is owned by root and is a tcsh(1) instance ❹, just as top's I/O mode said. The command is running under su(1), however. Check this process's parent process ID with the PPID column, and you'll see that process 3064 is a child of process 2990 ❸, which is a child of process 2989 ❷, both of which are owned by root. Process 2989 is a child of 2981 ❶, however, which is a shell run by a real user. You might also note that these processes are all parts of session 2981, showing that they're probably all run in the same login session. The `TT` column shows `p1`, which means that the user is logged in on */dev/ttyp1*, the second virtual terminal on this machine. Investigating that SID would illuminate just what Bert thought he was doing.

Now that you know how process parent-child operations work, you can cheat. Add the `-d` flag, as in, `ps -ajxd`, to present processes arranged in a tree with their parents. You'll want a wide terminal.

It's normal for a system to experience brief periods of total utilization. If nobody else is using the system and nobody's complaining about performance, why not let this user run his job? If this process is causing problems for other users, however, we can either deprioritize it, use our root privileges to kill the job, or show up at the user's cubicle with a baseball bat.

Paging and Swapping

Using swap space isn't bad in and of itself. Swap space is much slower than chip memory, but it does work, and many programs don't need to have everything in RAM in order to run. The old rule of thumb says that a typical program spends 80 percent of its time running 20 percent of its code. Much of the rest of its code covers startup and shutdown, error handling, and so on. You can safely let those bits go out of RAM with minimal performance impact.

Swap caches data that it has handled. Once a process uses swap, that swap remains in use until the process either exits or calls the memory back from swap.

Swap usage occurs through *paging* and *swapping*. Paging is all right; swapping is not so good, but it's better than crashing.

Paging

Paging occurs when FreeBSD moves a portion of a running program into swap space. Paging can actually improve performance on a heavily loaded system because unused bits can be stored on disk until they're needed—if ever. FreeBSD can then use the real memory for actual running code. Does it really matter whether your system puts your database startup code to swap once the database is up and running?

Swapping

If the computer doesn't have enough physical memory to store a process that isn't being run at that particular microsecond, the system can move the entire process to swap. When the scheduler starts that process again, FreeBSD fetches the entire process from swap and runs it, probably consigning some other process to swap.

The problem with swapping is that disk I/O activity goes through the roof and performance drops dramatically. Since requests take longer to handle, there are more requests on the system at any one time. Logging in to check the problem only makes the situation worse because your login is just one more process. Some systems can handle certain amounts of swapping, while on others, the situation quickly degenerates into a death spiral.

When your CPU is overloaded, the system is slow. When your disks are a bottleneck, the system is slow. Memory shortages can actually crash your computer. If you're swapping, you *must* buy more memory or resign yourself to appalling performance. If you're trapped into this hardware and can't buy more memory, you might get a really fast SSD and use it for swap.

The output of vmstat(8) shows the number of processes swapped out at any one time.

Performance Tuning

FreeBSD caches recently accessed data in memory because a surprising amount of information is read from the disk time and time again. Information cached in physical memory can be accessed very quickly. If the system needs more memory, it dumps the oldest cached chunks in favor of

new data. UFS and ZFS use different methods to decide what to cache, but the principle generally applies.

When I booted my desktop this morning, I started Firefox so I could check my RSS feeds. The disk worked for a moment or two to read in the program. I then shut the browser off so I could focus on my work, but FreeBSD left Firefox in the cache. If I restart Firefox, FreeBSD will pull it from memory instead of troubling the disk, which dramatically reduces its startup time. Had I started a process that demanded a whole bunch of memory, though, FreeBSD would have dumped the web browser from the cache to support the new program.

If your system is operating well, you'll have at least a few megabytes of free memory. The sysctls `vm.v_free_target` and `hw.pagesize` tell you how much free memory FreeBSD thinks it needs on your system. If you consistently have more free memory than these two sysctls multiplied, your system isn't being used to its full potential. For example, on my mail server I have:

```
# sysctl vm.v_free_target
vm.v_free_target: 5350
# sysctl hw.pagesize
hw.pagesize: 4096
```

My system wants to have at least 5,350 × 4,096 = 21,913,600 bytes, or about 22MB, of free memory. I could lose a gigabyte of RAM from my desktop without flinching, if it wasn't for the fact that I suffer deep-seated emotional trauma about insufficient RAM.[3]

Memory Usage

If a host has a lot of memory in cache or buffer, or the ARC has eaten all its RAM, it doesn't have a memory shortage. You might make good use of more memory, but it isn't strictly necessary. If you have low free memory, but a lot of active and non-ZFS wired memory, your system is devouring RAM. Adding memory would let you take advantage of the buffer cache.

If the pagedaemon keeps running, incrementing the sr field in your vmstat output, the kernel is working hard to provide memory. The host might well have a memory shortage. Once the host start to use swap, though, this memory shortage is no longer hypothetical. It might not be bad, but it's not theoretical.

Swap Space Usage

Swap space helps briefly cover RAM shortages. For example, if you're untarring a huge file you might easily consume all of your physical memory and start using virtual memory. It's not worth buying more RAM for such occasional tasks when swap suffices. If a memory-starved server runs a daemon that doesn't ever get called, that daemon will eventually get mostly or entirely swapped out in favor of processes that are performing work.

Only worry about swap space use when the system constantly pages data in and out of swap.

In short, swap space is like wine. A glass or two now and then won't hurt you and might even be a good choice. Hitting the bottle constantly is a problem. If you have to swap constantly, consider a really fast but durable SSD.

CPU Usage

A processor can do only so many things a second. If you run more tasks than your CPU can handle, requests will start to back up, you'll develop a processor backlog, and the system will slow down. That's CPU usage in a nutshell. If performance is unacceptable and top(1) shows your CPU hovering around 100 percent all the time, CPU utilization is probably your problem. While new hardware is certainly an option, you do have other choices. For example, investigate the processes running on your system to see whether they're all necessary. Did some junior sysadmin install a SETI@Home client to hunt for aliens with your spare CPU cycles? How about a Bitcoin miner? Is anything running that was important at one time, but not any longer? Find and shut down those unnecessary processes, and make sure that they won't start the next time the system boots.

If you have very specific needs, such as dedicating certain processors to specific tasks, consider cpuset(1). It's overkill for most users, but a high-performance application might make good use of dedicated processors.

Once that's done, evaluate your system performance again. If you still have problems, try rescheduling or reprioritizing.

Rescheduling

Rescheduling is easier than reprioritizing; it's a relatively simple way to balance system processes so that they don't monopolize system resources. As discussed in Chapter 20, you and your users can schedule programs to run at specific times with cron(8). If you have users who are running massive jobs at particular times, you might consider using cron(1) to run them in off hours. Frequently, jobs such as the monthly billing database search can run between 6 PM and 6 AM and nobody will care—Finance just wants the data on hand at 8 AM on the first day of the month so they can close out last month's accounting. Similarly, you can schedule your `make buildworld && make buildkernel` at 1 AM.

Reprioritizing with Niceness

If rescheduling won't work, you're left with reprioritizing, which can be a little trickier. When reprioritizing, you tell FreeBSD to change the importance of a given process. For example, you can have a program run during busy hours, but only when nothing else wants to run. You've just told that program to be *nice* and step aside for other programs.

The nicer a process is, the less CPU time it demands. The default niceness is `0`, but niceness runs from `20` (very nice) to `-20` (not nice at all). This might seem backward; you could argue that a higher number should mean a higher priority. That would lead to a language problem, however; calling this factor "selfishness" or "crankiness" instead of "niceness" didn't seem like a good idea at the time.[4]

The top(1) tool displays a `PRI` column for process priority. FreeBSD calculates a process's priority from a variety of factors, including niceness, and runs high-priority processes first whenever possible. Niceness affects priority, but you can't directly edit priority.

If you know that your system is running at or near capacity, you can choose to run a command with nice(1) to assign the process a niceness. Specify niceness with `nice -n` and the nice value in front of the command. For example, to start a very selfish `make buildworld` at nice 15, you'd run:

```
# nice -n 15 make buildworld
```

Only root can assign a negative niceness to a program, as in `nice -n -5`. For example, if you want to abuse your superuser privileges to make a compile

finish as quickly as possible, use a negative niceness:

```
# nice -n -20 make
```

NICE VS. TCSH

The tcsh(1) shell has a nice command built in. That built-in nice uses the renice(8) syntax, which is different from nice(1). I'm sure there's a reason for that other than annoying tcsh users, but that rationale escapes me at the moment. To use nice(1), use the full path */usr/bin/nice*.

Usually, you don't have the luxury of telling a command to be nice when you start it but instead have to change its niceness when you learn that it's absorbing all of your system capacity. You can use renice(8) to reprioritize running processes by their process IDs or owners. To change the niceness of a process, run `renice` with the new niceness and the PID as arguments.

In my career, I've run several logging hosts. In addition to general syslog services, they usually also run several instances of flow-capture, Nagios, and other critical network awareness systems. I'll often use a web interface to all of this and allow other people to access my logs. If I find that intermittent load on the web server is interfering with my network monitoring or my syslogd(8) server, I must take action. Renicing the web server makes clients run more slowly, but that's better than slowing down monitoring. Use pgrep(1) to find the web server's PID:

```
# pgrep httpd
993
# renice 10 993
993: old priority 0, new priority 10
```

Boom! FreeBSD now serves web requests after other processes. This greatly annoys the users of that service, but since it's my server and I'm already annoyed, that's all right.

To renice every process owned by a user, use the `-u` flag. For example, to make my processes more important than anyone else's, I could do this:

```
# renice -5 -u mwlucas
1001: old priority 0, new priority -5
```

The 1001 is my user ID on this system. Again, presumably I have a very good reason for doing this, beyond my need for personal power.[5] Similarly, if that user who gobbled up all my processor time insists on being difficult, I could make his processes very, very nice, which would probably solve other users' complaints. If you have a big background database job, having the user running that job run nicely can let the foreground work proceed normally.

Niceness only affects CPU usage. It has no impact on disk or network activity.

THE BOTTLENECK SHUFFLE

Every system has bottlenecks. If you eliminate one bottleneck, performance will increase until another bottleneck is hit. The system's performance is bound by the slowest component in the computer. For example, a web server is frequently network-bound because the slowest part of the system is the internet connection. If you upgrade your gigabit uplink to a 2.4Gb/s OC-48, the system will hand out its sites as fast as its other components allow. The hypothetical "eliminating bottlenecks" that management often demands is really a case of "eliminating bottlenecks that interfere with your usual workload."

Now that you can look at system problems, let's learn how to hear what the system is trying to tell you.

Status Mail

FreeBSD runs maintenance jobs every day, week, and month, via periodic(8). These jobs perform basic system checks and notify the administrators of changes, items requiring attention, and potential security issues. The output

of each scheduled job is mailed daily to the root account on the local system. The simplest way to find out what your system is doing is to read this mail; many very busy sysadmins just like you have collaborated to make these messages useful. While you might get a lot of these messages, with a little experience, you'll learn how to skim the reports looking for critical or unusual changes only.

The configuration of the daily, weekly, and monthly reports is controlled in *periodic.conf*, as discussed in Chapter 20.

You probably don't want to log in as root on all of your servers every day just to read email, so forward root's mail from every server to a centralized mailbox. Make this change in */etc/mail/aliases*, as discussed in Chapter 20.

The only place where I recommend disabling these jobs is on embedded systems, which should be managed and monitored through some other means, such as your network monitoring system. On such a system, disable the periodic(8) checks in */etc/crontab*.

While these daily reports are useful, they don't tell the whole story. Logs give a much more complete picture.

Logging with syslogd

The FreeBSD logging system is terribly useful. Any Unix-like operating system allows you to log almost anything at almost any level of detail. While you'll find default system logging hooks for the most common system resources, you can choose a logging configuration that meets your needs. Almost all programs integrate with the logging daemon, syslogd(8).

The syslog protocol works through messages. Programs send individual messages, which the syslog daemon syslogd(8) catches and processes. syslogd(8) handles each message according to its facility and priority level, both of which client programs assign to messages. You must understand both facilities and levels to manage system logs.

Facilities

A *facility* is a tag indicating the source of a log entry. This is an arbitrary label, just a text string used to sort one program from another. In most cases, each program that needs a unique log uses a unique facility. Many programs

or protocols have facilities dedicated to them—for example, FTP is such a common protocol that syslogd(8) has a special facility just for it. syslogd also supports a variety of generic facilities that you can assign to any program.

Here are the standard facilities and the types of information they're used for.

`auth` Public information about user authorization, such as when people logged in or used su(1).

`authpriv` Private information about user authorization, accessible only to root.

`console` Messages normally printed to the system console.

`cron` Messages from the system process scheduler.

`daemon` A catch-all for all system daemons without other explicit handlers.

`ftp` Messages from FTP and TFTP servers.

`kern` Messages from the kernel.

`lpr` Messages from the printing system.

`mail` Mail system messages.

`mark` This facility puts an entry into the log every 20 minutes. This is useful when combined with another log.

`news` Messages from the Usenet News daemons.

`ntp` Network Time Protocol messages.

`security` Messages from security programs, such as pfctl(8).

`syslog` Messages from the log system about the log system itself. Don't log when you log, however, as that just makes you dizzy.

`user` The catch-all message facility. If a userland program doesn't specify a logging facility, it uses this.

`uucp` Messages from the Unix-to-Unix Copy Protocol. This is a piece of pre-internet Unix history that you'll probably never encounter.

`local0` through `local7` These are provided for the sysadmin. Many programs have an option to set a logging facility; choose one of these if at all possible. For example, you might tell your customer service system to log to `local0`.

While most programs have sensible defaults, it's your job as the sysadmin

to manage which programs log to which facility.

Levels

A log message's *level* represents its relative importance. While programs send all of their logging data to syslogd, most systems record only the important stuff that syslogd receives and discard the rest. Of course, one person's trivia is another's vital data, and that's where levels come in.

The syslog protocol offers eight levels. Use these levels to tell syslogd what to record and what to discard. The levels are, in order from most to least important:

`emerg` System panic. Messages flash on every terminal. The computer is basically hosed. You don't even have to reboot—the system is doing it for you.

`crit` Critical errors include things such as bad blocks on a hard drive or serious software issues. You can continue to run as is, if you're brave.

`alert` This is bad, but not an emergency. The system can continue to function, but this error should be attended to immediately.

`err` These are errors that require attention at some point, but they won't destroy the system.

`warning` These are miscellaneous warnings that probably won't stop the program that issued them from working just as it always has.

`notice` This includes general information that probably doesn't require action on your part, such as daemon startup and shutdown.

`info` This includes program information, such as individual transactions in a mail server.

`debug` This level is usually only of use to programmers and occasionally to sysadmins who are trying to figure out why a program behaves as it does. Debugging logs can contain whatever information the programmer considered necessary to debug the code, which might include information that violates user privacy.

`none` This means, "Don't log anything from this facility." It's most commonly used to exclude information from wildcard entries, as we'll see shortly.

By combining level with priority, you can categorize messages quite narrowly and treat each according to your needs.

Processing Messages with syslogd(8)

The syslogd(8) daemon catches messages from the network and compares them with entries in */etc/syslog.conf* or files in */etc/syslog.d/*. Files in */etc/syslog.d/* are for your own entries and add-on programs, while */etc/syslog.conf* is for integrated system programs. Syslogd only reads */etc/syslog.d/* files ending in *.conf*. Both files have the same format, but I'll refer to */etc/syslog.conf* for clarity. That file has two columns; the first describes the log message, either by facility and level, or by program name. The second tells syslogd(8) what to do when a log message matches the description. For example, look at this entry from the default *syslog.conf*:

```
mail.info                                   /var/log/maillog
```

This tells syslogd(8) that when it receives a message from the `mail` facility with a level of `info` or higher, the message should be appended to */var/log/maillog*.

The logger won't log to a nonexistent file. Use touch(1) to create the log file before restarting syslogd(8).

Wildcards

You can also use wildcards as an information source. For example, this line logs every message from the `mail` facility:

```
mail.*                                      /var/log/maillog
```

To log everything from everywhere, uncomment the `all.log` entry and create the file */var/log/all.log*:

```
*.*                                         /var/log/all.log
```

This works, but I find it too informative to be of any real use. You'll find yourself using complex grep(1) statements daisy-chained together to find even the simplest information. Also, this would include all sorts of private data.

Excluding Information

Use the `none` level to exclude information from a log. For example, here, we exclude `authpriv` information from our all-inclusive log. The semicolon allows you to combine entries on a single line:

```
*.*; authpriv.none        /var/log/most.log
```

Comparison

You can also use the comparison operators `<` (less than), `=` (equal), and `>` (greater than) in *syslog.conf* rules. While syslogd defaults to recording all messages at the specified level or above, you might want to include only a range of levels. For example, you could log everything of `info` level and above to the main log file while logging the rest to the debug file:

```
mail.info                 /var/log/maillog
mail.=debug               /var/log/maillog.debug
```

The `mail.info` entry matches all log messages sent to the `mail` facility at `info` level and above. The second line matches only the messages that have a level of precisely `debug`. You can't use a simple `mail.debug` because the debugging log will then duplicate the content of the previous log. This way, you don't have to sort through debugging information for basic mail logs, and you don't have to sort through mail transmission information to get your debugging output.

Local Facilities

Many programs offer to log via syslog. Most of these can be set to a facility of your choice. The various `local` facilities are reserved for these programs. For example, by default, dhcpd(8) (see Chapter 20) logs to the facility `local7`. Here, we catch these messages and send them to their own file:

```
local7.*                  /var/log/dhcpd
```

If you run out of `local` facilities, you can use other facilities that the system isn't using. For example, I've once used the `uucp` facility on a busy log server on a network that had no `uucp` services.

Logging by Program Name

If you're out of facilities, you can use the program's name as a matching term. An entry for a name requires two lines: the first line contains the program name with a leading exclamation mark and the second line sets up logging. For example, FreeBSD uses this to log ppp(8) information:

```
!ppp
*.*                     /var/log/ppp.log
```

The first line specifies the program name and the second one uses wildcards to tell syslogd(8) to append absolutely everything to a file.

The `!programname` syntax affects all lines after it, so you must put it last in *syslogd.conf*. You can safely use it in an */etc/syslog.d* file without worrying about affecting other entries.

Logging to User Sessions

When you log to a user, any messages that arrive appear on that user's screen. To log to a user session, list usernames separated by commas as the destination. To write a message to all users' terminals, use an asterisk (`*`). For example, the default *syslog.conf* includes this line:

```
*.emerg                 *
```

This says that any message of emergency level will appear on all users' terminals. Since these messages usually say "goodbye" in one way or another, that's appropriate.

Sending Log Messages to Programs

To direct log messages to a program, use a pipe symbol (`|`):

```
mail.*                  |/usr/local/bin/mailstats.pl
```

Logging to a Logging Host

My networks habitually have a single logging host that handles not only the FreeBSD boxes but also Cisco routers and switches, other Unix boxes, and any syslog-speaking appliances. This greatly reduces system maintenance and saves disk space. Each log message includes the hostname, so you can easily sort them out later.

Use the *at* symbol (@) to send messages to another host. For example, the following line dumps everything your local syslog receives to the logging host on my network:

```
*.*                 @loghost.blackhelicopters.org
```

The *syslog.conf* on the destination host determines the final destination for those messages.

On the logging host, you can separate logs by the host where the log message originated. Use the plus (+) symbol and the hostname to indicate that the rules that follow apply to this host:

```
+dhcpserver
local7.*            /var/log/dhcpd
+ns1
local7.*            /var/log/named
```

Put your generic rules at the top of *syslog.conf*. Per-host rules should go near the bottom or in separate *syslog.d* files.

Logging Overlap

The logging daemon doesn't log on a first-match or last-match basis; instead, it logs according to every matching rule. This means you can easily have one log message in several different logs. Consider the following snippet of log configuration.

```
*.notice;authpriv.none        /var/log/messages
local7.*                      /var/log/dhcp
```

Almost every message of level `notice` or more is logged to */var/log/messages*. Anything with a facility of `authpriv` is deliberately excluded from this log, though. We have our DHCP server logging to */var/log/dhcp*. This means that any DHCP messages of `notice` level or above will be logged to both */var/log/messages* and */var/log/dhcpd*. I don't like this; I want my DHCP messages only in */var/log/dhcpd*. I can follow the `authpriv` example to deliberately exclude DHCP messages from */var/log/messages* by using the `none` facility:

```
*.notice;authpriv.none;local7.none        /var/log/messages
```

My */var/log/messages* syslog configuration frequently grows quite long as I incrementally exclude every local facility from it, but that's all right.

SPACES AND TABS

Traditional Unix-like operating systems require tabs between the columns in *syslog.conf*, but FreeBSD permits you to use spaces. Be sure to use only tabs if you share the same *syslog.conf* between different operating systems.

syslogd Customization

FreeBSD runs syslogd by default, and out of the box it can be used as a logging host. You can customize how it works through the use of command line flags. You can specify flags either on the command line or in *rc.conf* as `syslogd_flags`.

Allowed Log Senders

You can specify exactly which hosts syslogd(8) accepts log messages from. This can be useful so you don't wind up accepting logs from random people on the internet. While sending you lots of logs could be used to fill your hard drive as a preparation for an attack, it's more likely to be the result of a misconfiguration. Your log server should be protected by a firewall in any case. Use the `-a` flag to specify either the IP addresses or the network of hosts that can send you log messages, as these two (mutually exclusive) examples show:

```
syslogd_flags="-a 192.168.1.9"
syslogd_flags="-a 192.168.1.0/24"
```

While syslogd(8) would also accept DNS hostnames and domain names for this restriction, DNS is a completely unsuitable access control mechanism.

You can entirely disable accepting messages from remote hosts by specifying the `-s` flag, FreeBSD's default. If you use `-ss` instead, syslogd(8) also disables sending log messages to remote hosts. Using `-ss` removes

syslogd(8) from the list of network-aware processes that show up in sockstat(1) and netstat(1). While this half-open UDP socket is harmless, some people feel better if syslogd(8) doesn't appear attached to the network at all.

Attach to a Single Address

syslogd(8) defaults to attaching to UDP port 514 on every IP address the system has. Your jail server needs syslogd, but a jail machine can run only daemons that bind to a single address. Use the `-b` flag to force syslogd(8) to attach to a single IP:

```
syslogd_flags="-b 192.168.1.1"
```

Additional Log Sockets

syslogd(8) can accept log messages via Unix domain sockets as well as over the network. The standard location for this is */var/run/log*. No chrooted processes on your system can access this location, however. If you want those chrooted processes to run, you must either configure them to log over the network or provide an additional logging socket for them. Use the `-l` flag for this and specify the full path to the additional logging socket:

```
syslogd_flags="-l /var/named/var/run/log"
```

The named(8) and ntpd(8) programs come with FreeBSD and are commonly chrooted. The */etc/rc.d/syslogd* is smart enough to add the appropriate syslogd sockets if you chroot these programs through *rc.conf*.

Verbose Logging

Logging with verbose mode (`-v`) prints the numeric facility and level of each message written in the local log. Using doubly verbose logging prints the name of the facility and level instead of the number:

```
syslogd_flags="-vv"
```

These are the flags I consider most commonly. Read syslogd(8) for the complete list of options.

Log File Management

Log files grow, and you must decide how large they can grow before you trim them. The standard way to do this is through *log rotation*. When logs are rotated, the oldest log is deleted, the current log file is closed up and given a new name, and a new log file is created for new data. FreeBSD includes a basic log file processor, newsyslog(8), which also compresses files, restarts daemons, and in general handles all the routine tasks of log file shuffling. cron(1) runs newsyslog(8) once per hour.

When newsyslog(8) runs, it reads */etc/newsyslog.conf* and the files in */etc/newsyslog.conf.d/*. The */etc/newsyslog.conf* file is for core system functions, while files in */etc/newsyslog.conf.d/* are for add-on software. The newsyslog program attempts to parse any files in */etc/newsyslog.conf.d/* as newsyslog configurations. Both use the same format, so we'll refer to *newsyslog.conf* for clarity. Each line in *newsyslog.conf* gives the condition for rotating one log file. If the conditions for rotating the log are met, the log is rotated and other actions are taken as appropriate. */etc/newsyslog.conf* uses one line per log file; each line has seven fields, like this:

```
/var/log/ppp.log       root:network    640  3     100  *     JC
```

Let's examine each field in turn.

Log File Path

The first entry on each line (`/var/log/ppp.log` in the example) is the full path to the log file to be processed.

Owner and Group

The second field (`root:network` in our example) lists the rotated file's owner and group, separated by a colon. This field is optional and isn't present in many of the standard entries.

newsyslog(8) can change the owner and group of old log files. By default, log files are owned by the root user and the wheel group. While it's not common to change the owner, you might need this ability on multiuser machines.

You can also choose to change only the owner or only the group. In these

cases, you use a colon with a name on only one side of it. For example, `:www` changes the group to www, while `mwlucas:` gives me ownership of the file.

Permissions

The third field (`640` in our example) gives the permissions mode in standard Unix three-digit notation.

Count

This field specifies the oldest rotated log file that newsyslog(8) should keep. newsyslog(8) numbers archived logs from newest to oldest, starting with the newest as log 0. For example, with the default `count` of `5` for */var/log/messages*, you'll find the following message logs:

```
messages
messages.0.bz
messages.1.bz
messages.2.bz
messages.3.bz
messages.4.bz
messages.5.bz
```

Those of you who can count will recognize that this makes six archives, not five, plus the current log file, for a week of logs. As a rule, it's better to have too many logs than too few; however, if you're tight on disk space, deleting an extra log or two might buy you time.

Size

The fifth field (`100` in our example) is the file size in kilobytes. When newsyslog(8) runs, it compares the size listed here with the size of the file. If the file is larger than the size given here, newsyslog(8) rotates the file. If you don't want the file size to affect when the file is rotated, put an asterisk here.

Time

So far this seems easy, right? The sixth field, rotation time, changes that. The time field has four different legitimate types of value: an asterisk, a number, and two different date formats.

If you rotate based on log size rather than age, put an asterisk here.

If you put a plain naked number in this field, newsyslog(8) rotates the log after that many hours have passed. For example, if you want the log to rotate every 24 hours but don't care about the exact time when that happens, put `24` here.

The date formats are a little more complicated.

ISO 8601 Time Format

Any entry beginning with an `@` symbol is in the restricted ISO 8601 time format. This is a standard used by newsyslog(8) on most Unix-like systems; it was the time format used in MIT's primordial newsyslog(8). Restricted ISO 8601 is a bit obtuse, but every Unix-like operating system supports it.

A full date in the restricted ISO 8601 format is 14 digits with a `T` in the middle. The first four digits are the year, the next two the month, the next two the day of the month. The `T` is inserted in the middle as a sort of decimal point, separating whole days from fractions of a day. The next two digits are hours, the next two minutes, the last two seconds. For example, the date of March 2, 2008, 9:15 and 8 seconds PM is expressed in restricted ISO 8601 as `20080302T211508`.

While complete dates in restricted ISO 8601 are fairly straightforward, confusion arises when you don't list the entire date. You can choose to specify only fields near the `T`, leaving fields further away as blank. Blank fields are wildcards. For example, `1T` matches the 1st day of every month. `4T00` matches midnight of the 4th day of every month. `T23` matches the 23rd hour, or 11 PM, of every day. With a *newsyslog.conf* time of `@T23`, the log rotates every day at 11 PM.

As with cron(1), you must specify time units in detail. For example, `@7T`, the seventh day of the month, rotates the log once an hour, every hour, on the seventh day of the month. After all, it matches all day long! A time of `@7T01` would rotate the log at 1 AM on the 7th day of the month, which is probably more desirable. You don't need more detail than an hour, however, as newsyslog(8) runs only once an hour.

FreeBSD-Specific Time

The restricted ISO 8601 time system doesn't allow you to easily designate

weekly jobs, and it's impossible to specify the last day of the month. That's why FreeBSD includes a time format that lets you easily perform these common tasks. Any entry with a leading cash sign ($) is written in the FreeBSD-specific *month week day* format.

This format uses three identifiers: `M` (day of month), `W` (day of week), and `H` (hour of day). Each identifier is followed by a number indicating a particular time. Hours range from 0 to 23, while days run from 0 (Sunday) to 6 (Saturday). Days of the month start at 1 and go up, with `L` representing the last day of the month. For example, to rotate a log on the fifth of each month at noon I could use `$M5H12`. To start the month-end log accounting at 10 PM on the last day of the month, use `$MLH22`.

ROTATING ON SIZE AND TIME

You can rotate logs at a given time, when they reach a certain size, or both. If you specify both size and time, the log rotates whenever either condition is met.

Flags

The `flags` field dictates any special actions to be taken when the log is rotated. This most commonly tells newsyslog(8) how to compress the log file, but you can also signal processes when their log is rotated out from under them.

Log File Format and Compression

Logs can be either text or binary files.

Binary files can be written to only in a very specific manner. newsyslog(8) starts each new log with a "logfile turned over" message, but adding this text to a binary file would damage it. The `B` flag tells newsyslog(8) that this is a binary file and that it doesn't need this header.

Other log files are written in plain old ASCII text, and newsyslog(8) can and should add a timestamped message to the top of the file indicating when the log was rotated. If you're using UFS, compressing old log files can save

considerable space. The `-J` flag tells newsyslog(8) to compress archives with bzip(1); the `-Z` flag specifies gzip compression; the `-X` flag, xz(1); and the `-Y` flag, the new hotness in compression, zstd(1).

If you're using ZFS, though, text log files get compressed at the dataset layer along with every other compressible file. You can compress the log files in the traditional manner anyway, but there's no advantage to doing so. Plus, you'll need to manually decompress the files before you can view them. Let ZFS handle compression for you.

Special Log File Handling

When it creates and rotates log files, newsyslog(8) can perform a few special tasks. Here are the most common; you can read about the others in newsyslog.conf(5).

Perhaps you have many similar log files that you want to treat identically. The `-G` flag tells newsyslog that the log file name at the beginning of the line is actually a shell glob, and that all log files that match the expression are to be rotated in this manner. To learn about shell expressions, read glob(3). Bring aspirin.

You might want newsyslog to create a file if it doesn't exist. Use the `-C` flag for this. The syslogd program won't log to a nonexistent file.

The `-N` flag explicitly tells newsyslog not to send a signal when rotating this log.

Finally, use a hyphen (-) as a placeholder when you don't need any of these flags. It creates a column here so that you can have, say, a pidfile path.

Pidfile

The next field is a pidfile path (not shown in our example, but look at */etc/newsyslog.conf* for a couple of samples). A pidfile records a program's process ID so that other programs can easily view it. If you list the full path to a pidfile, newsyslog(8) sends a `kill -HUP` to that program when it rotates the log. This signals the process to close its logfiles and restart itself. Not all processes have pidfiles, and not all programs need this sort of special care when rotating their logs.

Signal

Most programs perform logfile rotation on a SIGHUP, but some programs need a specific signal when their logs are rotated. You can list the exact signal necessary in the last field, after the pidfile.

Sample newsyslog.conf Entry

Let's slap all this together into a worst-case, you've-*got*-to-be-kidding example. A database log file needs rotation at 11 PM on the last day of the month. The database documentation says that you must send the server an interrupt signal (SIGINT, or signal number 2) on rotation. You want the archived logs to be owned by the user *dbadmin* and viewable only to that user. You need six months of logs. What's more, the logs are binary files. Your *newsyslog.conf* line would look like this:

```
/var/log/database    dbadmin:   400   6   *   $MLH23   B   /var/run/db.pid   2
```

This is a deliberately vile example; in most cases, you just slap in the filename and the rotation condition, and you're done.

FreeBSD and SNMP

Emailed reports are nice but general, and logs are difficult to analyze for long-term trends. The industry standard for network, server, and service management is *Simple Network Management Protocol (SNMP)*. Many different vendors support SNMP as a protocol for gathering information from many different devices across the network. FreeBSD includes an SNMP agent, bsnmpd(8), that not only provides standard SNMP functions but also gives visibility to FreeBSD-specific features.

FreeBSD's *bsnmpd* (short for *Begemot SNMPD*) is a minimalist SNMP agent specifically designed to be extensible. All actual functionality is provided via external modules. FreeBSD includes the bsnmpd modules for standard network SNMP functions and modules for specific FreeBSD features, such as PF and netgraph(4). Rather than trying to be all things to all people, bsnmpd(8) offers a foundation where everyone can build an SNMP implementation that does only what they need, no more and no less.

SNMP 101

SNMP works on a classic client-server model. The SNMP client, usually some kind of management workstation or monitoring server, sends a request across the network to an SNMP server. The SNMP server, also called an *agent*, gathers information from the local system and returns it to the client. FreeBSD's SNMP agent is bsnmpd(8).

An SNMP agent can also send a request to make changes to the SNMP server. If the system is properly (or improperly, depending on your point of view) configured, you can issue commands via SNMP. This "write" configuration is most commonly used in routers, switches, and other embedded network devices. Most Unix-like operating systems have a command line management system and don't usually accept instruction via SNMP. Writing system configuration or issuing commands via SNMP requires careful setup and raises all sorts of security issues; it's an excellent topic for an entire book. No sysadmin I know is comfortable managing their system via SNMP. With all of this in mind, we're going to focus specifically on read-only SNMP.

In addition to having an SNMP server answer requests from an SNMP client, the client can transmit SNMP *traps* to a trap receiver elsewhere on the network. An SNMP agent generates these traps in response to particular events on the server. SNMP traps are much like syslogd(8) messages, except that they follow the very specific format required by SNMP. FreeBSD doesn't include an SNMP trap receiver at this time; if you need one, check out snmptrapd(8) from net-snmp (*net-mgmt/net-snmp*).

SNMP MIBs

SNMP manages information via a management information base (MIB), a tree-like structure containing hierarchical information in ASN.1 format. We've seen an example of an MIB tree before: the sysctl(8) interface discussed in Chapter 6.

Each SNMP server has a list of information it can extract from the local computer. The server arranges these bits of information into a hierarchical tree. Each SNMP MIB tree has very general main categories: network, physical, programs, and so on, with more specific subdivisions in each. Think of the tree as a well-organized filing cabinet, where individual drawers

hold specific information and files within drawers hold particular facts. Similarly, the uppermost MIB contains a list of MIBs beneath it.

MIBs can be referred to by name or by number. For example, here's an MIB pulled off a sample system:

```
interfaces.ifTable.ifEntry.ifDescr.1 = STRING: "em0"
```

The first term in this MIB, `interfaces`, shows us that we're looking at this machine's network interfaces. If this machine had no interfaces, this first category wouldn't even exist. The `ifTable` is the interface table, or a list of all the interfaces on the system. `ifEntry` shows one particular interface, and `ifDescr` means that we're looking at a description of this interface. This MIB can be summarized as, "Interface number 1 on this machine is called em0."

MIBs can be expressed as numbers, and most SNMP tools do their work natively in numerical MIBs. Most people prefer words, but your poor brain must be capable of working with either. An MIB browser can translate between the numerical and word forms of an SNMP MIB for you, or you could install *net-mgmt/net-snmp* and use snmptranslate(1), but for now, just trust me. The preceding example can be translated to:

```
.1.3.6.1.2.1.2.2.1.2.1
```

Expressed in words, this MIB has 5 terms separated by dots. Expressed in numbers, the MIB has 11 parts. That doesn't look quite right if they're supposed to be the same thing. What gives?

The numerical MIB is longer because it includes the default `.1.3.6.1.2.1`, which means *.iso.org.dod.internet.mgmt.mib-2*. This is the standard subset of MIBs used on the internet. The vast majority of SNMP MIBs (but not all) have this leading string in front of them, so nobody bothers writing it down any more.

If you're in one of those difficult moods, you can even mix words and numbers:

```
.1.org.6.1.mgmt.1.interfaces.ifTable.1.2.1
```

At this point, international treaties permit your coworkers to drive you from the building with pitchforks and flaming torches. Pick one method of expressing MIBs and stick to it.

MIB Definitions and MIB Browsers

MIBs are defined according to a very strict syntax and are documented in *MIB files*. Every SNMP agent has its own MIB files; bsnmpd's are in */usr/share/snmp*. These files are very formal plaintext. While you can read and interpret them with nothing more than your brain, I highly recommend copying them to a workstation and installing an MIB browser so that you can comprehend them more easily.

MIB browsers interpret MIB files and present them in their full tree-like glory, complete with definitions of each part of the tree and descriptions of each individual MIB. Generally speaking, an MIB browser lets you enter a particular MIB and displays both the numerical and word definitions of that MIB, along with querying an SNMP agent for the status of that MIB.

If you have FreeBSD (or a lesser Unix) on your workstation, use mbrowse (*net-mgmt/mbrowse*) for MIB browsing. If you don't want to use a graphical interface for SNMP work, check out net-snmp (*net-mgmt/net-snmp*) for a full assortment of command line SNMP client tools.

SNMP Security

Many security experts state that SNMP really stands for "Security: Not My Problem!" This is rather unkind but very true. SNMP needs to be used only behind firewalls on trusted networks. If you must use SNMP on the naked internet, use packet filtering to keep the public from querying your SNMP service. SNMP agents run on UDP port 161.

The more common SNMP versions, 1 and 2c, provide no encryption. This means that anyone with a packet sniffer can capture your SNMP community name, so be absolutely certain you're using SNMP only on a private network. Making unencrypted SNMP queries over an untrusted network is a great way to have strangers poking at your system management. SNMP version 3 uses encryption to protect data on the wire.

SNMP provides basic security through *communities*. If you go looking around, you'll find all sorts of explanations for why a community isn't the same thing as a password, but a community *is* a password. Most SNMP agents have two communities by default: public (read-only access) and private (read-write access). Yes, there's a default that provides read-write access. Your first task whenever you provision an SNMP agent on any host, on any OS, is to disable those default community names and replace them

with ones that haven't been widely documented for decades.

FreeBSD's bsnmpd(8) defaults to SNMPv2c but can do SNMPv3. SNMPv3 is a more complicated protocol, so we're not going to cover it here. If you understand the SNMPv3 protocol and the basics of configuring FreeBSD's bsnmpd, you won't have any trouble enabling SNMPv3 in bsnmpd.

Configuring bsnmpd

Before you can use SNMP to monitor your system, you must configure the SNMP daemon. Configure bsnmpd(8) in */etc/snmpd.config*. In addition to including the default communities of public and private, the default configuration doesn't enable any of the FreeBSD-specific features that make bsnmpd(8) desirable.

bsnmpd Variables

bsnmpd uses variables to assign values to configuration statements. Most high-visibility variables are set at the top of the configuration file, as you'll see here:

```
location := "Room 200"
contact := "sysmeister@example.com"
system := 1     # FreeBSD
traphost := localhost
trapport := 162
```

These top variables define values for MIBs that should be set on every SNMP agent. The location describes the physical location of the machine. Every system needs a legitimate email contact. bsnmpd(8) runs on operating systems other than FreeBSD, so you have the option of setting a particular operating system here. Lastly, if you have a trap host, you can set the server name and port here.

Further down the file, you can set the SNMP community names:

```
# Change this!
read := "public"
# Uncomment begemotSnmpdCommunityString.0.2 below that sets the community
# string to enable write access.
write := "geheim"
trap := "mytrap"
```

The `read` string defines the read-only community of this SNMP agent. The default configuration file advises you to change it. Take that advice. The `write` string is the read-write community name, which is disabled by default further down in the configuration file. You can also set the community name for SNMP traps sent by this agent.

With only this configuration, bsnmpd(8) will start, run, and provide basic SNMP data for your network management system. Just set `bsnmpd_enable="YES"` in */etc/rc.conf* to start bsnmpd at boot. You won't get any special FreeBSD functionality, however. Let's go on and see how to manage this.

Detailed bsnmpd Configuration

bsnmpd(8) uses the variables you set at the top of the configuration file to assign values to different MIBs later in the configuration. For example, at the top of the file you set the variable `read` to `public`. Later in the configuration file, you'll find this statement:

```
begemotSnmpdCommunityString.0.1 = $(read)
```

This sets the MIB `begemotSnmpdCommunityString.0.1` equal to the value of the `read` variable.

Why not just set these values directly? bsnmpd(8) is specifically designed to be extensible and configurable. Setting a few variables at the top of the file is much easier than directly editing the rules further down the file.

Let's go back to this `begemotSnmpdCommunityString` MIB set here. Why are we setting this? Search for the string in your MIB browser, and you'll see that this is the MIB that defines an SNMP community name. You probably could have guessed that from the assignment of the `read` variable, but it's nice to confirm that.

Similarly, you'll find an entry like this:

```
begemotSnmpdPortStatus.0.0.0.0.161 = 1
```

Checking the MIB browser shows that this dictates the IP address and the UDP port that bsnmpd(8) binds to (in this case, all available addresses, on port 161). All MIB configuration is done in this manner.

Loading bsnmpd Modules

Most interesting bsnmpd(8) features are configured through modules. Enable modules in the configuration file by giving the `begemotSnmpdModulePath` MIB a class that the module handles and the full path to the shared library that implements support for that feature. For example, in the default configuration, you'll see a commented-out entry for the PF bsnmpd(8) module:

```
begemotSnmpdModulePath."pf"    = "/usr/lib/snmp_pf.so"
```

This enables support for PF MIBs. Your network management software will be able to see directly into PF when you enable this, letting you track everything from dropped packets to the size of the state table.

As of this writing, FreeBSD's bsnmpd(8) ships with the following modules included but disabled. Some are FreeBSD-specific, while others support industry standards. Enable these by uncommenting their configuration file entries and restarting bsnmpd.

`lm75` Provides data from the lm75(4) temperature sensor via SNMP.

`Netgraph` Provides visibility into all Netgraph-based network features, documented in snmp_netgraph(3).

`PF` Provides visibility into the PF packet filter.

`Hostres` Implements the Host Resources SNMP MIB, snmp_hostres(3).

`bridge` Provides visibility into bridging functions, documented in snmp_bridge(3).

`wlan` Accesses information on wireless networking.

Restart bsnmpd(8) after enabling any of these in the configuration file. If the program won't start, check */var/log/messages* for errors.

With bsnmpd(8), syslogd(8), status emails, and a wide variety of performance analysis tools, you can make your FreeBSD system the best-monitored device on the network. Now that you can see everything your system offers, grab a flashlight as we explore a few of FreeBSD's darker corners.

22
JAILS

Virtualization separates an operating system instance from the underlying hardware. Virtualization allows you to move operating system installs from one piece of hardware to another by copying a file. Virtualization needs an operating system installed on the hardware, but that install is normally very minimal, has no public-facing services, and is easily reproduced on new hardware. It's perhaps the biggest change in system administration in decades.

Virtualization is something like a client-server environment. The hardware and its core operating system instance is the *host*, while the *clients* are all virtualized operating system instances. The clients rely on the host to provide basic services, such as storage, processor power, and memory. Changes to the host can be reflected in the virtualized clients, but changes on the client have no effect on the host beyond consuming resources.

FreeBSD supports two types of virtualization: jails and bhyve.

Jails are a lightweight virtualization method, sometimes called *OS-level virtualization*. A jail normally contains a complete operating system userland that runs on top of an existing FreeBSD system. The jail relies on the host's filesystem but is limited to a subset of the directory tree. It might even have a chunk of dedicated space in a ZFS pool. A jail doesn't have its own kernel and instead runs in a restricted portion of the host's kernel. The host can manage jailed processes without entering the jail, or it can run processes inside the jail if preferred. Jails don't get a graphical console. Use jails to virtualize FreeBSD installs of the same version or older, or to run simple virtual Linux systems.

Bhyve is a heavier virtualization system. Rather than using the host's kernel and filesystem, bhyve simulates hardware. The host provides a chunk of disk space for the virtual machine to use as a disk. A bhyve virtual machine must bring along its own filesystem, kernel, and supporting infrastructure. Bhyve virtual machines require more resources than jails, but they also offer a console via virtual network computing (VNC) and can run truly foreign operating systems, like Microsoft Windows. Bhyve is changing rapidly thanks to its rapid development, so this book doesn't cover it. I'll write about bhyve once it stabilizes.

Before considering bhyve, see whether a jail will meet your needs.

Jail Basics

A jail is a supercharged chroot that applies not only to the filesystem but also to processes and the network stack. A jailed system can access only a narrow part of the filesystem and can't see processes outside the jail. Traditionally, each jail is assigned a dedicated IP address, and the jail can view only traffic to that particular IP. Each jail even has its own user accounts. The root account in a jail completely controls that jail but has no access to anything beyond the jail.

To a user given root access to a jail, the jail looks like a nearly complete FreeBSD system, missing only a few device nodes. The user can install whatever software she likes without interfering with the host or other jails. All processes running in the jail can affect only the jail's files and processes. The jailed user has no visibility of anything beyond the jail; she's confined. If the jail is hacked, the intruder is also confined to the jail.

Jails can use a virtual network stack, based on vnet(9). That's an advanced use we won't cover here, but if you need to provide a jail with its own routing table, that's how you do it.

As of FreeBSD 9, multiple jails can share a single IP address, but the sysadmin needs to configure each jail to use unique TCP/IP ports for every network service. You can't run multiple SSH instances on port 22 of a single IP! For simplicity, the following examples use a single IP for each jail, but remember that you have other options.

Many people put all of their services in jails, even when a host is dedicated to a particular purpose. A ZFS snapshot of the jail dataset, or a

tarball of the directory tree, is a complete backup of the jail. Restoration after a failed software upgrade becomes a simple matter of extracting a tarball or rolling back to the snapshot.

A jail is also useful for software development and testing. Deploying a new service often requires installing and testing quite a few packages. Doing the testing in a jail before selecting a solution and proceeding to production prevents polluting the host with abandoned files and unneeded software.

Depending on your hardware and the system load, a single FreeBSD host can support dozens or even hundreds of jails. If you want to seriously run that many jails, though, make sure your host has two separate network interfaces. Dedicate one to jails and the other to managing the host.

Everything starts with configuring your jail host.

Jail Host Server Setup

A server meant as a jail host must work within a few annoying constraints. Configure your host correctly before building your first jail.

The jail system has its own sysctl tree, `security.jail`. You can change these sysctls only from the host system. Some sysctls affect all jails running on the host. Sysctls that begin with `security.jail.param` can be set on a per-jail basis. We'll touch on these throughout this chapter.

Jail Host Storage

I strongly advise you to use your jail host only for the purpose of running jails and to put all services inside a jail. Start by configuring the host's storage to separate jails and the host operating system.

Many hosts intended for virtualization include SATA DOM flash drives on the mainboard for the operating system. These drives are usually less than 100GB in size, but a base install of FreeBSD fits in much less than a gigabyte. If you have a SATA DOM or similar, use it for the host operating system. If you have multiple sets of redundant hard drives, use a pair to mirror the operating system and dedicate everything else to jails.

If you don't have such hardware, dedicate space to the host operating system. Use either a partition for UFS filesystems or a dataset reservation for ZFS. In either case, 10GB of space should be sufficient. If you need

additional space for an emergency, you can borrow some from the jail space.

While ZFS is highly useful for jails, it's not necessary. I ran jails on UFS for many years. Use what works for you and fits your environment.

Once you have your host partitioned and the operating system installed, look at the network.

Jail Networking

There are two seemingly conflicting aspects of jail networking: first, each jail expects full control of any IP addresses assigned to it; and second, jails can share IP addresses with other jails and even the host. You can start a jail using any IP address on the host, but that jail can't coordinate any network-facing services with other services running on that IP. If your jail shares the host's IP address, and the host runs SSH on port 22, the jail can't use port 22. If you try to start sshd(8) in the jail, the program will complain that it can't use port 22 and crash. Sharing IP addresses between jails, or even between jails and the host, requires the sysadmin to coordinate which ports belong to which hosts and to configure everything accordingly.

The simplest way to configure jails is to assign each its own IP address and give the host its own IP address. Each jail can then completely control its own IP address. Once you get the hang of this, you can start sharing addresses between jails. That means the host can't have daemons listening on IP addresses assigned to jails. Having a host's daemons listening on the jail's IP won't prevent the jail from starting, but it will prevent the jail from starting its own services on that port. Users like Bert will complain if they can't SSH to their private jails!

The cleanest way to configure a jail host is to decide that the host only provides jails. Any services run on the host must be in a jail. If you need simple services, like a nameserver or a mail exchanger, configure them in a jail. Not only is this easier than properly reconfiguring all these servers to attach only to the selected IP address; it also provides an additional layer of security for your other jails. An intrusion on the host automatically grants the intruder access to all of your jails, while an intrusion on a single jail confines the intruder to that jail.

Use sockstat(1) to identify programs listening on your network, as discussed in Chapter 9. Add the `-46` flags to show only IPv4 and IPv6 traffic,

and `-l` to show only listening sockets.

```
# sockstat -46l
USER     COMMAND  PID   FD PROTO  LOCAL ADDRESS     FOREIGN ADDRESS
root     ntpd     19776 20 udp6   *:123             *:*
root     ntpd     19776 21 udp4   *:123             *:*
--snip--
root     sshd     2846  3  tcp6   *:22              *:*
root     sshd     2846  4  tcp4   *:22              *:*
```

This fairly default FreeBSD install has two programs listening to the network: ntpd and sshd. Both are listening to all IP addresses. We must configure all of these daemons to listen only to the main server address.

Here are some common daemons that cause problems on host servers. In all of these, I'll assume that the jail host has an IP of 198.51.100.50.

syslogd

The system logger syslogd(8) opens a UDP socket so that it can send messages to other hosts. If you don't log remotely, or if you use a different logging solution, use the `-ss` flag in *rc.conf* to turn off the network component.

```
syslogd_flags="-ss"
```

If you need to send syslogd messages, use the `-b` flag to force syslogd to attach to only a single IP address.

```
syslogd_flags="-b 198.51.100.50"
```

Either solution lets your jails' administrators individually decide whether or not they're going to log across the network. See Chapter 21 for a full discussion of syslogd.

inetd

If you need inetd (see Chapter 20), you should almost certainly run it from within a jail rather than the host. If you can't weasel out of running inetd on the host, though, use the `-a` flag to restrict it to a single IP address, as in the following *rc.conf* snippet.

```
inetd_flags="-a 198.51.100.50 -wW -C 60"
```

If I had only specified the `-a` flag and the IP address, it would've overwritten inetd's default flags from */etc/defaults/rc.conf*. Every release of FreeBSD from the last few decades has used the default flags of `-wW -C 60`; I added my `-a` and the IP address to those flags.

sshd

The option `ListenAddress` in */etc/ssh/sshd_config* tells sshd(8) which addresses to bind to. Restrict it to only your host IP.

```
ListenAddress 198.51.100.50
```

If the only service your jail host offers is sshd(8), you've done well.

NFS

Network filesystem programs, such as rpcbind(8) and nfsd(8), bind to all IP addresses on a host, no matter what you do. Don't run these programs inside a jail, and don't run NFS from within a jail. If your clients need NFS mounts, have the host run these programs and provide the NFS mounts.

Network Time Protocol

The most problematic service on a jail host is timekeeping. All jails get their system clock from the host. FreeBSD's included time daemon, ntpd(8), listens to all IP addresses on the host—including the jailed ones. As the lone rare exception, though, I'm going to tell you to go ahead and run ntpd on the host.

A jail lacks the proper access to change the kernel's time. While you could run ntpd in a jail, it couldn't actually *do* anything. Go ahead and run ntpd on your jail host, and don't worry about it[1] listening to all IP addresses. Anyone who tries to run a UDP-based service other than ntpd on port 123 is probably trying to evade a packet filter. Make them work harder.

If you want to avoid even the chance of a collision, install the `openntpd` package. Unlike the base system ntpd(8), OpenNTPD can be configured to listen to a single IP address.

IP Addresses

Each jail can have one or more IP addresses. These addresses must be

attached to the host before you start the jail. A jail will run without any networking, but it won't be accessible beyond the host. Add any necessary IP addresses as aliases in */etc/rc.conf*.

Jails at Boot

To have FreeBSD start your jails at boot, set `jail_enable` in *rc.conf*.

```
jail_enable=YES
```

FreeBSD defaults to starting all jails listed in */etc/jail.conf*. If you want the system to start only a subset of those jails at boot, use the `jail_list` *rc.conf* option. Here, I have two jails, called *mariadb* and *httpd*. I want them started in this order so that my database jail is running before the web server that calls it.

```
jail_list="mariadb httpd"
```

During system shutdown, FreeBSD stops jails in the same order it starts them. Your application might not like that. In my example, I want the web server to turn off before the database backend. I'd rather have a website be flat-out unavailable than have users see the dreaded "database server is kaput" error.

```
jail_reverse_stop=YES
```

If the jail startup order is unimportant, you can start and stop all jails simultaneously.

```
jail_parallel_start="YES"
```

Now you can configure a jail.

Jail Setup

Now that I have a host, I can install some jails. I'll start with a jail called *mariadb*, for running . . . wait for it . . . MariaDB.

Each jail needs a dedicated root directory. All of my example jails live under */jail*. I normally put each jail in a directory named after the jail name—

in this case, */jail/mariadb*.

Each jail needs a primary IP. It can also have other IPs, as we'll see later, but let's start with one. The jail mariadb gets 203.0.113.51.

Each jail needs an internet hostname, just as if it were a real host. This jail will become *mariadb.mwl.io*.

Now we can put a userland in the jail.

Jail Userland

While you can install any userland components in a jail, all a jail requires is the base system. Grab the *base.txz* distribution set for your desired FreeBSD release and extract it in your jail's root directory.

```
# tar -xpf base.txz -C /jail/mariadb
```

That's a complete install of the base operating system. If you want additional distribution sets, such as the debugging symbols, extract them the same way.

If you've built your own FreeBSD base system, you can install it in the jail.

```
# cd /usr/src
# make installworld DESTDIR=/jail/mariadb
```

Jails also need supporting directories and assorted detritus created by the install process, but not by `make installworld`. The `make distribution` command creates those files. If you already have these directories and files, though, don't rerun `make distribution`: it'll overwrite any local changes. And don't forget the DESTDIR setting, unless you like resetting the host's configuration!

```
# make distribution DESTDIR=/jail/mariadb
```

You can also build a custom userland with only enough binaries for running a single program, much as you would for a traditionally chrooted program. For most of us, that's too much work, but if you want to break out ldd(1) and go wild, don't let me stop you.

Once you have a jail userland, tell FreeBSD about your jail in */etc/jail.conf*.

/etc/jail.conf

Traditionally, FreeBSD configured jails in */etc/rc.conf*. This was clunky and unwieldy. While FreeBSD still supports *rc.conf* configuration of jails, I recommend using the more flexible */etc/jail.conf* instead. This file isn't in UCL, although it looks like something UCL could support. Define each jail by a name. Give the jail parameters in braces after the jail name. Each parameter definition ends in a semicolon.

Many jail parameters have an equal sign, where we assign a parameter a value. Here, I set the parameter `path` to the value `/jail/mariadb`:

```
path="/jail/mariadb";
```

Other parameters enable or disable a feature with their mere presence. Here, I tell this jail to turn on the `mount.devfs` feature:

```
mount.devfs;
```

Jails support a whole bunch of "mount" parameters, with subparameters for different filesystems. This particular parameter specifically addresses mounting devfs.

Toggles can be turned off for a jail by adding *no* in front of the specific parameter. If I don't want to enable devfs, I wouldn't turn off the whole mount parameter; I'd put the *no* in front of the *devfs*.

```
mount.nodevfs;
```

Here's how I define a jail named *mariadb*:

```
mariadb {
 host.hostname="mariadb.mwl.io";
 ip4.addr="203.0.113.51";
 path="/jail/mariadb";
 mount.devfs;
 exec.clean;
 exec.start="sh /etc/rc";
 exec.stop="sh /etc/rc.shutdown";
}
```

The parameter `host.hostname` gives the jail's hostname. While the jail name is *mariadb*, this host identifies itself by the internet hostname *mariadb.mwl.io*.

The IP address is in `ip4.addr`. I've assigned the address 203.0.113.51 to this

jail. This IP must be on the host first.

The jail's root directory goes in the `path` variable. Here, it's set to */jail/mariadb*.

Almost every jail needs access to specific device nodes in */dev*, which requires mounting devfs (see Chapter 13) in the jail. Enable devfs with the `mount.devfs` setting. A jail defaults to getting only a few very specific device nodes. An untrusted user can sometimes use device nodes to escape a jail, so don't add additional devices without careful research. You can allow other device nodes with a custom devfs ruleset. Assign a custom devfs ruleset to the jail with the devfs_ruleset *jail.conf* parameter. I strongly recommend using the default jail devfs rules as a base and unhiding the additional devices this jail needs, rather than trying to build a custom devfs ruleset from scratch.

A jailed process can inherit parts of its environment from the parent process. The `exec.clean` option tells jail(8) to strip away all of the environment except for $TERM. The environment variables $HOME, $USER, and $SHELL get set to the target environment, normally that of the jail's root account. You'll almost always want `exec.clean`.

The `exec.start` and `exec.stop` options tell FreeBSD how to start and stop the jail.

In-Jail Startup

Jails can emulate a full-running FreeBSD userland, run a single process, or anything in between. You must either use the `exec.start` *jail.conf* parameter to tell FreeBSD what process to run in the jail or the `persist` parameter to declare you want the jail to exist even without any processes in it. Here, I start a full FreeBSD userland, using the normal FreeBSD startup script:

```
exec.start="/bin/sh /etc/rc"
```

If you need only a single command to run inside the jail, you can write your own startup script and use `exec.start` to run it when the jail boots. Your brand-new jail won't have an *rc.conf* yet, so it won't start any additional processes.

Instead of `exec.start`, you could set the `persist` option. This tells FreeBSD that a jail can exist without any processes running inside it. Including both

`persist` and `exec.start` means that FreeBSD will start a process for the jail, but when the process stops running, the jail won't shut itself down.

You can tell the jail to run an additional command after it starts with the `exec.poststart` option. Any command or script listed with `exec.poststart` gets run in the host once the normal */etc/rc* startup process (including any enabled packages) finishes. This lets you write scripts to glue jails together.

Similarly, you can use the `exec.prestop` option to run a command on the host before stopping the jail. When the sysadmin turns the jail off, the host first runs this command, and then the jail runs the normal shutdown command.

The `exec.stop` command tells FreeBSD what command to run inside the jail to shut the jail down. If you're simulating a full jail, you'll probably run `/bin/sh /etc/rc.shutdown` as in our example in the previous section.

Jail Defaults

You'll find that many of your jails share common settings. You can define those settings in the front of the configuration. All jails will use those settings unless you override them. This doesn't seem to make much sense when using a single jail.

```
exec.start="/bin/sh /etc/rc";
exec.stop="/bin/sh /etc/rc.shutdown";
exec.clean;
mount.devfs;

mariadb {
        host.hostname="mariadb.mwl.io";
        ip4.addr="203.0.113.221";
        path="/jail/mariadb";
}
```

Given this configuration, though, adding another jail becomes five lines including the braces.

```
httpd {
        host.hostname="httpd.mwl.io";
        ip4.addr="203.0.113.222";
        path="/jail/httpd";
}
```

Over dozens of jails, it saves a lot of trouble.

You can override the defaults within a jail's definition. If I don't want to

mount devfs(5) in a jail, I would set `mount.nodevfs` for that specific jail.

jail.conf Variables

You can use variable substitutions in jails. While you can define some of these variables, you can also pull some from the jail's settings. Variables are expanded in double quotes and in unquoted strings, but not in single-quoted strings.

Here, I define a variable for the directory that contains all of my jails and use that inside my jail definition:

```
$j="/jail";
mariadb {
        path="$j/mariadb";
--snip--
}
```

If I must move my jails to a new filesystem or pool, I can update *jail.conf* by changing the one variable rather than editing every definition.

Parameters as Variables

Once you define a jail parameter, you can use it as a variable. Every jail has at least one parameter, `name`. You can use these parameters to further expand default settings.

```
$j="/jail";
path="$j/$name";
host.hostname="$name.mwl.io";

mariadb {
        ip4.addr="203.0.113.221";
}
```

By setting the global default `path` to `$j/$name`, I've removed the need to define path for each individual jail.

You can use multiterm parameters with a period in them by enclosing the parameter in braces. While this doesn't make sense for parameters like `mount.devfs`, it's useful for per-jail parameters, like `host.hostname`.

```
path = "/jail/${host.hostname}";
```

I prefer to put my jails in directories named after the shorter name, rather

than the hostname, but feel free to indulge your own biases.

Combining parameters and variables with a coherent directory layout lets you squeeze each jail definition down into a single configuration statement.

Testing and Configuring a Jail

Once you have files for a jail, lock yourself in. Run the jail(8) command to run a single command inside the jail. You'll need four arguments: the path, the jail name, the primary IP, and the command to run.

```
# jail <path to jail> <jail name> <jail IP> <command>
```

Here, I use the jail in */jail/mariadb*, named *mariadb*, with the IP address 203.0.113.51, to run the command */bin/sh*:

```
# jail /jail/mariadb/ mariadb 203.0.113.51 /bin/sh
#
```

Run ls(1). You're in the root directory of your jail filesystem. This jail isn't quite in single-user mode, but no programs other than */bin/sh* are running here. You can do some basic setup, but not even devfs(5) is mounted.

```
# ps
ps: /dev/null: No such file or directory
```

Yes, the normal jail startup process would mount */dev*—but the jail has no user accounts, no root password, no daemons running, and absolutely nothing optional. Configure the jail before starting it.

Stuff to Steal from the Host

Some host setup information is also useful within the jail. You can copy this information from the host to the jail, but you must do this from the host, not the jail.

Each jail performs its own DNS resolution. You can probably copy the host's */etc/resolv.conf* into the jail.

Your jail probably shares the same time zone as the host. Copy the host's */etc/localtime* into the jail or run tzsetup(8) inside the jail to select a new time zone.

Create /etc/fstab

Many programs and scripts, including */etc/rc*, expect to find */etc/fstab* and have a tantrum if it's not there. Requiring */etc/fstab* is perfectly sensible in a real server, but a jailed machine has no need for a filesystem table. Create an empty filesystem table.

```
# touch /etc/fstab
```

I don't mind unhappy programs. I just don't want to listen to them whinge.

Create /etc/rc.conf

Either you'll do all jail management from the host, or you'll manage jails via SSH. You'll need an */etc/rc.conf* entry for sshd.

```
sshd_enable="YES"
```

Add any other settings you want while creating this file. If you know some of the settings packages will need, it won't hurt to set them before they're needed.

User Accounts and Root Password

You can add user accounts and change passwords only from within the jail. Set a root password with passwd(1) and run adduser(8) to add at least one user, for SSH. While SSH is not the only way to access the host, it's far easier in most cases.

Jail Startup and Shutdown

The host considers each jail an independent service, much like sshd(8), a web server, or any other daemon. Yes, each jail might run a whole bunch of services that need managing independently, but from the host's perspective, each jail is a single entity containing a group of processes. That's part of the separation between the host and the jail.

Use service(8) to start, stop, and restart jails. You'll need to provide one additional argument, the name of the jail. FreeBSD automatically starts them at boot, but you can stop, start, and restart them individually once the

system is running. Let's shut down my database jail and fire it up again.

```
# service jail stop mariadb
# service jail start mariadb
```

I could use the restart command, but that wouldn't look nearly so impressive here on the page.

If you omit the jail name, the service(8) command affects all jails that FreeBSD starts at boot.

```
# service jail restart
Stopping jails: httpd mariadb.
Starting jails: mariadb httpd
```

This lets you coherently reinitialize your production jail infrastructure.

FreeBSD defaults to starting all jails listed in */etc/jail.conf*. As discussed in "Jails at Boot" on page 568, you can change that in */etc/rc.conf*. The service(8) command can control jails that aren't autostarted, but you must specify them by name.

Jail Dependencies

If you have a whole bunch of jails, listing the start order in */etc/rc.conf* can get tedious. You'll most often need to set a start order to maintain service dependencies. Rather than defining the order in *rc.conf*, though, you can tell a jail that it requires another jail with the `depend` option.

```
httpd {
        ip4.addr="203.0.113.232";
        depend=mariadb;
}
```

The jail httpd won't start until the jail mariadb is running. A `depend` statement overrides a *rc.conf* `jail_list` entry.

Managing Jails

Virtualization doesn't make system administration tasks evaporate; it only adds options for performing typical sysadmin tasks. Here's some of those options.

Viewing Jails and Jail IDs

Use jls(8) to see all jails currently running on the system.

```
# jls
   JID  IP Address      Hostname        Path
    29  203.0.113.221   mariadb.mwl.io  /jail/mariadb
    30  203.0.113.222   httpd.mwl.io    /jail/httpd
    31  203.0.113.223   test.mwl.io     /jail/test
```

Each jail has a unique jail ID, or JID. The JID is much like a process ID; while each jail has one, the exact JID issued to a jail changes each time the jail is started. We'll use the jail ID or name to execute various jail-management tasks.

We also get each jail's IP address, hostname, and the path to the jail's files. You don't get the jail name, but those of us who use a hostname based on the jail name have no trouble figuring it out.

Jailed Processes

Jailed processes all get a process ID, like any other Unix process. Process IDs are not unique to jails; they're shared between the host, the jail, and all the other jails. You won't find repeated process IDs.

Jailed processes show up in ps(1) with the `-J` flag.

```
# ps -ax
  PID TT  STAT        TIME COMMAND
    0  -  DLs      2:56.24 [kernel]
    1  -  ILs      0:00.07 /sbin/init –
--snip--
35002  -  SsJ      0:00.01 /usr/sbin/syslogd -s
35129  -  IsJ      0:00.00 /usr/sbin/sshd
--snip--
```

Process IDs 35002 and 35129 are jailed.

View a particular jail's processes with ps(1) using the `-J` flag and the jail name.

```
# ps -ax -J test
  PID TT  STAT    TIME COMMAND
35561  -  IsJ  0:00.01 /usr/sbin/syslogd -s
35652  -  IsJ  0:00.00 /usr/sbin/sshd
35661  -  SsJ  0:00.03 sendmail: accepting connections (sendmail)
--snip--
```

Using `-J 0` excludes all jailed processes from ps(1) output, letting you more easily debug the host.

Commands like pgrep(1), pkill(1), and killall(1) all accept a `-j` argument to let you specify a jail. If you prefer using pgrep(1) to view process information, use `pgrep -lfj` and the jail name or JID.

```
# pgrep -lf -j mariadb
  PID TT  STAT    TIME COMMAND
35002  -  SsJ  0:00.01 /usr/sbin/syslogd -s
35129  -  IsJ  0:00.00 /usr/sbin/sshd
35158  -  SsJ  0:00.02 sendmail: accepting connections (sendmail)
--snip--
```

Why is Sendmail running inside this jail? Let's kill it.

```
# pkill -9 -j mariadb sendmail
```

Running pgrep again shows that Sendmail is dead.

This works well if you want to get information about which processes are running in a jail, but sometimes you have a process ID and must identify which jail it belongs to. That's where you need the `-O` option to ps(1). This option supports a bunch of keywords that adjust the output of ps(1) in ways not supported by the regular command line flags—specifically, `-O jail` adds a column for the name of the jail the process is running in.

```
# ps -ax -O jail | grep 39415
39415 mariadb  -  IsJ    0:00.00 /usr/local/libexec/mysqld
```

This process is running inside the jail mariadb.

Running Commands in Jails

The jexec(8) command lets the jail host administrator execute commands within a jail without going to the trouble of logging into the jail. This helps preserve the jail owner's sense of privacy.[2] When jail owner Bert calls to beg for help, I don't need his root password or even an account on his system. Using jexec requires knowing the jail's name or JID. Here, I use the host's root account to run `ps -ax` inside my jail mariadb.

```
# jexec mariadb ps -ax
  PID TT  STAT    TIME COMMAND
35002  -  SsJ  0:00.00 /usr/sbin/syslogd -s
```

```
35129  -  IsJ  0:00.00 /usr/sbin/sshd
--snip--
```

This command runs as root inside the jail. I might want to run the command as another jailed user, though. Give that username with the `-U` flag.

```
# jexec -U xistence mariadb ps
jexec: xistence: no such user
```

Well, that's not good. I'm expecting Bert to run my database. Let's make him a user account.

```
# jexec mariadb adduser
Username: xistence
Full name: Bert JW Regeer
Uid (Leave empty for default):
Login group [bert]:
Login group is bert. Invite bert into other groups? []: wheel
--snip--
```

This jail now has an account for Bert, using his preferred username and everything. I've added it to the wheel group within the jail. Remember, root access within a jail doesn't equal root access on the host. That's the whole point of jails.

I can now run commands as that user in that jail.

```
# jexec -U xistence mariadb sh
$
```

I'm locked up in jail! Specifically, in Bert's jail cell.

This jailed process will behave a little oddly, though. A process retains its environment. In this case, while I'm running as the user xistence, I retain all the environment settings I had in my nonjailed process. This includes stuff like $SSH_AUTH_SOCK, my IRC server setting, and more. I don't want this stuff in my jailed environment. If I'm logged in as Bert, I want to *be* Bert.

To strip your environment before entering a jail, use jexec's `-l` flag. This simulates a clean login.

```
# jexec -lU xistence mariadb sh
```

Should you always strip your environment before running a command in

a jail? No, not always. It depends entirely on what you're doing.

Many commands include support for running them on the host but targeting a jail. Always check the man page for such an option. One good example is sysrc(8), which lets you specify a jail with `-j`. Here, I enable MariaDB on the jail mariadb. MariaDB has chosen to continue to use MySQL naming conventions, so it's enabled with the *rc.conf* option `mysql_enable`.

```
# sysrc -j mariadb mysql_enable=YES
mysql_enable:  -> YES
```

This jail is now ready to run MariaDB.

Except for the bit where MariaDB isn't installed, of course. Let's take care of that next.

Installing Jail Packages

FreeBSD's package tools let you manage software either from within the jail or from the host. If the host administrator has allocated you a jail to configure, you probably want to manage packages from the jail. Jail packages work exactly as packages on any other FreeBSD host, as discussed in Chapter 15. If you're responsible for the whole system, including the host and all the jails on that host, you probably want to manage each jail's packages from the host rather than logging into each jail. Let's spend some time on that.

The pkg(8) command's `-j` flag lets you specify a jail to manage. You'll need one argument, the jail's name or JID. The `-j` flag must be given before the pkg(8) subcommand. Here, I install the MariaDB server on its dedicated jail:

```
# pkg -j mariadb install mariadb101-server
Updating FreeBSD repository catalogue...
FreeBSD repository is up to date.
All repositories are up to date.
The following 9 package(s) will be affected (of 0 checked):
--snip--
```

Note that pkg(8) offers no notice that it's installing packages within a jail. It assumes that if you're using `-j`, you know you're working in a jail.

When you manage a jail's packages from the host, the package tools don't

get installed on the jail. The jail has its own package database, stored within the jail, but the jail has no way to use that database directly.

Don't switch between managing packages on the host and from within the jail. Choose one method and stick with it.

Updating Jails

So you have umpteen bajillion jails on your host, each dedicated to performing its own task in perfect isolation. That's grand, until you have to apply security patches to all of the hosts. If you've built your FreeBSD from source, you'll need to install a new world in each jail. If you're running releases, though, freebsd-update(8) (see Chapter 18) can handle jails.

You can't use freebsd-update(8) *inside* a jail. The same things that isolate a jail from compromising the host system disallow some of the functionality freebsd-update(8) needs. Instead, you update the jail from the host.

Any time you need to update your system, update your host before updating your jails. The host must be running a version of FreeBSD equal to or newer than any jail.

Start by copying */etc/freebsd-update.conf* to an alternate file, such as */etc/jail-freebsd-update.conf*. Remove all components that aren't installed on the jail. Jails don't have kernels, and most of them don't have source code, so you'll probably wind up with an entry like this:

```
Components world
```

When you run freebsd-update(8), it checks the version of FreeBSD you're running on. It does this by querying the kernel. If you have FreeBSD 12.0 jails on a FreeBSD 13.0 system, the update program gets confused, chokes, and dies with mysterious errors. You need freebsd-update to use the release installed in the jail, not the version the host is running. Use the `--currently-running` option to tell freebsd-update(8) what version the jail is running. You must use the jail's release, including the patch level. While you could easily enough extract that information from the jail, I encourage you to let freebsd-update ask the jail what version it's currently running. Do this by using jexec(8) to query the version of FreeBSD running in the jail.

You'll also use the `-b` flag to tell freebsd-update(8) the directory the jail lives in.

Here, I update the jail called *test*. The jail's files are in */jail/test*. I use a jexec(8) command in backticks to check the current FreeBSD version.

```
# freebsd-update -f /etc/jail-freebsd-update.conf -b /jail/test/ --currently-running
`jexec -l test freebsd-version` fetch install
```

Once freebsd-update finishes running, restart your jail. It's upgraded.

Many people have written scripts to run through */etc/jail.conf* and upgrade all of their jails. If you have more than a couple jails, you find or write such a script.

More Jail Options

You can customize jails in all sorts of ways. The jail(8) man page includes the current list of jail options, but here's a few features I commonly use.

Rather than letting jail(8) assign jail IDs, you can assign each jail a permanent ID with the `jid` option.

```
jid=101;
```

The `securelevel` option lets you raise the securelevel (see Chapter 9) within a jail. The jail's securelevel can never be lower than that of the host.

You can view a jail's startup messages by manually running a */etc/rc* with a jail command. That's inconvenient for routine troubleshooting, though. Direct the jail's console messages to a file with the `exec.consolelog` option.

```
exec.consolelog="$j/logs/$name.log";
```

In addition to mounting */dev*, a jail can have its own fdescfs(5) and procfs(5) with the `mount.fdescfs` and `mount.procfs` options.

Jailing Ancient FreeBSD

In my experience, the phrase *enterprise network* is synonymous with "we have lots of ancient stuff that nobody dares touch." Jails can help you cope with some of these systems. In 2014, I worked at a company that ran a critical custom-built PHP and MySQL application on FreeBSD 4.10. I don't know when this server was installed, but FreeBSD 4.11 came out in January 2005,

so: before then. This application used ancient versions of Perl, PHP, OpenSSL, and more.

Worse, this application lived on a repurposed desktop machine. With a standard-issue, high-quality desktop hard drive. I hid a spare desktop machine of the same vintage under my desk, just so I had a hope of getting FreeBSD 4.10 onto it. The most proper solution was to rewrite or replace this application. Several sysadmins had faced that task—and failed. I decided to virtualize it. FreeBSD 4.10 doesn't run well on VMWare—yes, you can find de(4) and fxp(4) drivers, but they're for versions of those cards over a decade old. Here's how I got this ancient FreeBSD system into a jail.

Drop to single-user mode. Unmount */proc*—yes, FreeBSD 4 still used */proc*. Those were the days. Tar up the entire filesystem, including temporary directories, like */usr/obj*, */usr/ports*, */var/tmp*, and suchforth. By modern standards, they won't use much space at all, and you have no way to know what files you might need later. You can probably find an old PHP 5.0.whatever tarball out on the internet, but that would involve work.

Copy the tar file to your jail host and extract it in your jail directory.

```
# tar -C /jail/oldserver -xvpf oldserver.tgz
```

Be sure to use the `-p` flag to preserve permissions.

Now look at */etc/rc.conf*. The jail host will handle all networking functions, so turn off any statements that set IP addresses or set routes. Get rid of daemons that provide services the jail host offers, such as time, packet filters, and SSH. Your jailed host needs only the functions that directly support the application. In this case, I needed Apache and MySQL.

Consider the jail's */etc/fstab*. Do you need a NFS filesystem or some other special mount? Remove everything you don't need. If this application needs */proc*, provide it with the jail option `mount.procfs`.

Remove the old */dev*. You can't use FreeBSD 4 device nodes on a modern FreeBSD.

Configure the host to protect the jail. While people can write perfectly fine applications in Apache and PHP, not even the most ardent Apache and MySQL fan would encourage you to expose 15-year-old versions of these servers to the internet. Use the host's packet filter to protect the jail. Don't even consider using the migrated host's OpenSSH server.

You won't be able to use some FreeBSD 4 commands inside the jail, as the interfaces have diverged too far. A FreeBSD 4 ps(1) can't successfully query a modern FreeBSD kernel. You can copy statically linked versions of most of those programs from the host's */rescue* and copy them into the jail, however.

Is it this simple? No, not really. The older the source system is, the more problems you'll have. Most of the problems I had in this particular migration meant changing a configuration file to account for the new underlying filesystem. You'll need to perform your usual sysadmin debugging. But it's one way to get a modern network interface on a system lacking a device driver for it, and it's the only way to get ZFS on a FreeBSD 4 system.

Last Jail Notes

People have evolved many ways of using jails. Fully covering all of these features would pretty much require a book of its own, but here are some pointers.

You can use ZFS features to delegate a dataset entirely to the jail administrator so that jail owners can take their own snapshots and create their own child datasets. With the VIMAGE kernel option, you can give a jail its own routing table. If you're brave, nullfs(5) lets you recycle an operating system install and minimize disk utilization. You can establish per-jail resource limits with the RCTL kernel option.

If you have many jails, you might prefer using a jail management program, such as iocage or ezjail. Both are available in the Ports Collection.

Successfully using jails requires automating your maintenance. Each jail requires separate security patches, both for the userland and for installed packages. The more you can automate this process, the more likely it is that you'll actually perform such maintenance. I recommend using Ansible's jail modules or at least writing your own shell script to apply patches.

The jail(8) command will let you modify, create, and destroy jails without a command line. If you're doing extensive jail work, definitely read the man page.

Now let's look at some of FreeBSD's less well-known corners.

23
THE FRINGE OF FREEBSD

If you hang around the FreeBSD community for any length of time, you'll hear mention of all sorts of things that can be done if you know how. People build embedded FreeBSD devices and ship them to customers all over the world, who don't even know that they have a Unix-like server inside the little box running their air conditioner or radio relay station. People run FreeBSD on machines without hard drives, supporting hundreds or thousands of diskless workstations from a single server. You'll find bootable CDs and USB devices that contain complete FreeBSD systems, including all the installed software you could ever want. These things aren't difficult to do, once you know the tricks.

In this chapter, we're headed into the fringes of FreeBSD—the really cool things that are done by FreeBSD users but aren't necessarily supported by the mainstream FreeBSD Project. While you can find support and assistance through the usual channels, you must be prepared to debug and troubleshoot everything in this chapter even more than usual.

Terminals

A *terminal* is the device that people can log in on. The keyboard, video, and mouse make up a terminal that's also called a *console*. When you SSH into your host, it provides a *virtual terminal*. Terminal configuration is overwhelmingly automatic, but you might need to tweak it.

The file */etc/ttys* controls how and where users may log into your FreeBSD system. Do console logins work? How about virtual terminals?

What about logging in over serial lines? FreeBSD systems offer four standard terminals: the console, virtual terminals, dial-up terminals, and pseudoterminals.

The *console* is the only device available in single-user mode. On most FreeBSD systems, this is either a video console that includes the monitor and keyboard or a serial console accessed from another system. Once the system hits multiuser mode, the console is usually attached to a virtual terminal instead. The console device is */dev/console*.

A *virtual terminal* is attached to the physical monitor and keyboard. You can have multiple terminals on your one physical terminal. Switch between them with ALT and the function keys. The next time you're at the keyboard, hit ALT-F2. You'll see a fresh login screen, with `ttyv1` after the hostname. This is the second virtual terminal. Hitting ALT-F1 takes you back to the main virtual terminal. By default, FreeBSD has eight virtual terminals and reserves a ninth for X Windows. You can use the eight virtual text terminals even when you're in X, and some X desktops provide multiple X virtual terminals. The virtual terminals are the */dev/ttyv* devices.

A *dial-up terminal* is connected via serial line. You can attach modems directly to your serial ports and let users dial into your server. This isn't so common these days, but the same functionality supports logging in over a serial console. Dial-up terminals are the */dev/ttyu* devices.

Finally, a *pseudoterminal* is implemented entirely in software. When you SSH into your server, you don't need any actual hardware, but the software still needs a device node for your session. Pseudoterminals are the device nodes in */dev/pts/*. You don't configure pseudoterminals; they're automatically negotiated when you log in.

Configure access to the console, virtual terminals, and dial-up terminals in */etc/ttys*. You can enable serial access, require or disable passwords, and more.

/etc/ttys Format

A typical entry in */etc/ttys* looks like this:

```
ttyv0   "/usr/libexec/getty Pc"        xterm  on  secure
```

The first field is the terminal's device node. In this case, `ttyv0` is the first

virtual terminal on the system.

The second field is the program that's spawned to process login requests on this terminal. FreeBSD uses getty(8), but if you have a preferred terminal management program, you can use it instead. You'll find several in packages. This field takes one argument, the terminal configuration. The file */etc/gettytab* contains all the terminal configurations.

The third entry is the terminal type. The file */etc/termcap* describes all the innumerable terminal types FreeBSD supports. For really small systems, FreeBSD provides */etc/termcap.small* with only the most vital entries. Almost everything modern works with either `xterm` or `vt100`.

The fourth entry determines whether the terminal is available for logins or not. This could be `on` for accepting logins or `off` for not allowing them. The `onifconsole` setting permits logins on a serial port if the kernel configured the port as a console.

Last, we have the options. This example has the option `secure` set, which tells getty(8) that root may log into this console.

Offering terminals is a low-level system task handled directly by init(8). Changes to */etc/ttys* don't take effect until you tell init(8) to reread its configuration file. Init is always PID 1.

```
# kill -1 1
```

Insecure Console

When you boot FreeBSD in single-user mode, you get a root command prompt. This is fine for your laptop and works nicely for servers in your corporate data center, but what about machines in untrusted facilities? If you have a server in a colocation center, for example, you probably don't want just anyone to be able to get root-level access to a machine. You can tell FreeBSD that the physical console is insecure and make it require the root password to enter single-user mode. The system will then boot from power-on to multiuser mode without requiring a password, but it'll require the password when you explicitly boot in single-user mode.

Requiring a password in single-user mode doesn't completely protect your data, but it does raise the bar considerably. A lone tech working late, when nobody's looking, could boot your system into single-user mode and

add an account for himself in only 15 minutes or so. Dismantling your machine, removing the hard drives, mounting them into another machine, making changes, and bringing your server back online requires much more time, is far more intrusive, and is much more likely to be noticed by colocation management.

Find the console entry in */etc/ttys*:

```
console none        unknown off secure
```

You'll see that the console terminal isn't as full-featured as other terminals; it doesn't run getty(8) and uses the generic `unknown` terminal type. The console is intended for use only in single-user mode and when attached to a physical terminal, however, so that's fine.

To make the console require a root login when booted into single-user mode, change `secure` to `insecure`.

```
console none        unknown off insecure
```

Password-protecting the console dissuades casual mischief. It won't even slow a knowledgeable intruder with physical access to the machine.

Managing Cloudy FreeBSD

Clusters of hundreds or thousands of servers are growing increasingly common. Automation systems like Ansible and Puppet somehow let us maintain these systems in a semblance of order. Unix wasn't designed to be operated that way, however. Primordial UNIX was written to be administered by a highly skilled operator who had no problem handling the vagaries of countless different command output formats and even more configuration file styles.

FreeBSD is attacking the problems of cloud-scale management with libXo and universal configuration language (UCL).

LibXo

While automated monitoring is a necessity and can alert you to issues, when it comes to in-depth troubleshooting, nothing replaces logging into a host,

running a command, and interpreting the result. I've lost track of how many scripts I've written to parse the output of some obscure combination of ps(1) flags so that I could feed a number to the monitoring software. I've also lost track of how many hours I've spent debugging those scripts or explaining why the script I wrote to process one netstat(1) flag is irrelevant to the flags we're interested in right now.[1] Multiply this by those hundreds or thousands of servers, and getting information out of software quickly becomes a serious problem.

FreeBSD has cut down this problem space with libXo.

LibXo is a library that helps commands provide output not only in text form but also in XML, JSON, and even HTML. Instead of using grep(1) and awk(1) and whatever appalling combination of shell or Perl or Python you've brewed up to find desired information, you can have a parser extract data from a tagged format. You can dump command output straight to a web page.

Not all programs support libXo, but support is continually added to more programs. The man page declares whether a program supports libXo, but if you're too lazy to read it, you can try the command with the `--libxo` flag. All commands that support libXo use that command line option. You must also specify the output format, either text, XML, JSON, or HTML. Here, I run `arp -an` and identify JSON as the output format.

```
$ arp -an --libxo json
{"__version": "1", "arp": {"arp-cache": [{"hostname":"?","ip-address":"
203.0.113.221","mac-address":"08:00:27:31:91:0d","interface":"em0",
"permanent":true,"type":"ethernet"}
--snip--
```

How do you use this? Many of us won't. But if you're running dozens or hundreds of servers, you probably have the expertise in-house to painlessly parse this. Hundreds of tools can select tagged data, and your application developer probably has their preferred software already installed on your hosts. And while the output of arp(8) is fairly consistent, libXo also handles any arbitrary combination of flags to netstat(1), vmstat(8), and more. Learn to grab tagged data from the output once, and you're done writing those horrible scripts *forever*.

Universal Configuration Language

Unix systems have a pretty standard configuration file format. Hash marks are comments. There are variables. Maybe the presence of the variable in a config file is enough to activate a feature, or perhaps you have to set the variable to a value. They're all a little bit different, though. Some programs can pull in configuration snippets from a primary file and the files in a directory, like cron(8) does with */etc/crontab* and */etc/cron.d/*. Others can't. Some use braces to set aside chunks of configuration, where others use . . . whatever the programmer thought was a good idea 30 years ago. The result is that nobody looks at *syslog.conf* and thinks it looks like *pkg.conf*, even though they share common underlying concepts.

The *universal configuration language (UCL)* aims to change that. If all of these programs have a similar syntax, why not use a single parsing library for each? And if you have a parsing library, why not let it parse multiple formats? UCL lets you provide configuration files in classic Unix style, JSON, or YAML, ideal for automated management. It can extract configuration settings in shell code, UCL, JSON, or YAML.

At the time I write this, FreeBSD uses UCL for pkg(8). Support for other utilities, such as bhyve(8), is slowly happening. If you're managing large numbers of servers, check to see the status of UCL in your release.

Diskless FreeBSD

While FreeBSD isn't difficult to manage, dozens or hundreds of nearly identical systems can become quite a burden. One way to reduce your maintenance overhead is to use *diskless* systems. Diskless systems aren't forbidden to have hard drives; rather, they load their kernel and operating system from an NFS server elsewhere on the network.

Why use a diskless system for your server farm? Multiple systems can boot off of a single NFS server, centralizing all patch and package management. This is excellent for collections of terminals, computation clusters, and other environments where you have large numbers of identical systems. Rolling out an operating system update becomes a simple matter of replacing files on the NFS server. Similarly, when you discover that an update has problems, reverting it is as simple as restoring files on the NFS server. In either case, the only thing you have to do at the client side is reboot. As the clients have read-only access to the server, untrusted users

can't make any changes to the operating system. If you have only a couple of systems running, diskless is probably too much work for you, but any more than that and diskless is a clear winner.

Before you can run diskless systems, you must have an NFS server, a DHCP server, a TFTP server, and hardware that supports diskless booting. Let's go through each and see how to set it up.

TEST, TEST, TEST!

Your first diskless setup will be much like your first firewall setup: error-prone, troublesome, and infuriating. I strongly suggest that you test each step of the preparation so that you can find and fix problems more easily. Test instructions are provided for each required service.

Diskless Clients

Machines that run diskless must have enough smarts to find their boot loader and operating system over the network. There are two standard ways of doing this: BOOTP and PXE. *BOOTP*, the internet Bootstrap Protocol, is an older standard that fell out of favor long ago. *PXE*, Intel's Preboot Execution Environment, has been supported on almost every new machine for years now, so we'll concentrate on that.

Boot your diskless client machine and go into the BIOS setup. Somewhere in the BIOS, you'll find an option to set the boot device order. If the machine supports PXE, one of those options will be the network. Enable that option and have the machine try it first.

Your diskless client is ready. Now let's get the server ready.

DHCP Server Setup

While most people think of DHCP as a way to assign IP addresses to clients, it can provide much more than that. You can configure your DHCP server to provide the locations of a TFTP server, an NFS server, and other network resources. Diskless systems make extensive use of DHCP, and you'll

find that we use DHCP options you've never tried before.

OpenBSD's DHCP server won't support FreeBSD diskless clients; you must use ISC's DHCP server or some other more full-featured version. Configuring the ISC DHCP server to handle diskless systems is pretty straightforward once you have the MAC address of your diskless workstation.

MAC Address

To assign configuration information to a DHCP client, you need the MAC address of that client's network card. Some BIOS implementations provide the MAC addresses of integrated network cards, and some server-grade hardware has labels with the MAC address printed on them. Those options, however, are too easy, so we'll try the hard way.

When a machine tries to boot off the network, it makes a DHCP request for its configuration information. While you don't have a diskless configuration yet, any DHCP server logs the MAC address of clients. You can get the client information from the leases file, */var/db/dhcpd.leases*.

```
  --snip--
❶  lease 198.51.100.10 {
     starts 6 2017/09/16 06:57:23;
     ends 6 2017/09/16 07:07:23;
  --snip--
❷  hardware ethernet 08:00:27:d8:c1:1c;
     uid "\001\010\000'\330\301\034";
     set vendor-class-identifier = "PXEClient:Arch:00000:UNDI:002001";
  }
  --snip--
```

This client has a MAC address of 08:00:27:d8:c1:1c ❷ and has been offered IP address 198.51.100.10 ❶. Given this information, we can create a DHCP configuration to assign this host a static IP address and provide its boot information.

DHCP Configuration: Specific Diskless Hosts

We configured basic DHCP services in Chapter 20. Here's a sample dhcpd(8) configuration for a diskless client. This doesn't go inside a subnet statement but is a top-level statement on its own, even if it's on a subnet shared with nondiskless DHCP clients.

```
❶ group diskless {
        ❷ next-server 198.51.100.1;
        ❸ filename "pxeboot";
        ❹ option root-path "198.51.100.1:/diskless/1/";
        ❺ host compute1.mwl.io {
                ❻ hardware ethernet 08:00:27:d8:c1:1c  ;
                ❼ fixed-address 198.51.100.101 ;
          }
    }
```

We define a group called `diskless` ❶. This definition will allow us to assign certain parameters to the group and then just add hosts to the group. Every host in the group gets those same parameters.

The `next-server` setting ❷ tells the DHCP clients the IP address of a TFTP server, and the `filename` option ❸ tells clients the name of the boot loader file to request from that TFTP server. Remember from Chapter 4 that the boot loader is the software that finds and loads the kernel. Finally, `option root-path` ❹ tells the boot loader where to find the root directory for this machine. All of these options and settings are given to all clients in the diskless group.

We then assign our diskless client to the diskless group using the `host` statement and the hostname of this system ❺. Our first client is called `compute1`. This client is identified by its MAC address ❻ and is assigned a static IP ❼. It also receives the standard configuration for this group.

Create additional host entries just like this for every diskless host on your network.

Restart dhcpd(8) to make this configuration take effect. Now reboot your diskless client. The DHCP log should show that you've offered this client its static address. However, the DHCP client can't boot any further without a boot loader, which means you need a TFTP server.

DHCP Configuration: Diskless Farms

Perhaps you have a large number of identical diskless hosts, such as thin clients in a terminal room. It's perfectly sensible not to want to make a static DHCP entry for each thin client. Let these hosts get their boot information from the DHCP server, but without specifying a host address. They'll just

take an address out of the DHCP pool. Many clustering solutions include client services that register new hosts with whatever "cluster manager" they're using, so hardcoded addresses aren't so important.

You can also specifically identify hosts that are requesting DHCP information from PXE and assign those hosts to a specific group of addresses. A host booting with PXE identifies itself to the DHCP server as a client of type `PXEclient`. You can write specific rules to match clients of that type and configure them appropriately. Look in the DHCP manual for information on how to match on `vendor-class-identifier` and `dhcp-client-identifier`.

tftpd and the Boot Loader

We covered configuring a TFTP server in Chapter 20. The TFTP server must provide the *pxeboot* file for your diskless clients. FreeBSD provides *pxeboot* in the */boot* directory.

```
# cp /boot/pxeboot /tftpboot
# chmod +r /tftpboot/pxeboot
```

Try to download *pxeboot* via TFTP from your workstation. If that works, reboot your diskless client and watch it try to boot. The console should show a message like this:

```
Building the boot loader arguments
Relocating the loader and the BTX
Starting the BTX loader
```

You've seen this message before, when a regular FreeBSD boots off its hard drive. Your diskless client will identify the PXE version, print the memory, and declare that it's running the bootstrap loader. At that point, it'll circle endlessly trying to load the kernel. It can't load the kernel because we haven't yet set up the NFS server.

Diskless Security

Diskless systems run over NFS and have all of NFS's security issues. Even if you deploy Kerberos to encrypt NFS traffic, the initial network boot and mounting of the root filesystem is always unencrypted. Don't run diskless

nodes on the open internet.

You can somewhat protect your NFS server by assigning a different user for the NFS root account. Running `find /diskless/1 -user 0 -exec chown nfsroot {} \;` changes the owner of all files owned by root to be owned by the user nfsroot. You can then edit the *exports* file to map root to the nfsroot user. You'd need to revert that to run freebsd-update(8), however, and then restore it after applying patches. But when you're first learning, don't get fancy. Get a basic userland working first.

The NFS Server and the Diskless Client Userland

Many tutorials on diskless operation suggest using the server's userland and root partition for diskless clients. That might be easy to do, but it's not even vaguely secure. Your diskless server probably has programs on it that you don't want the clients to have access to, and it certainly has sensitive security information that you don't want to hand out to a whole bunch of workstations. Providing a separate userland is a much wiser option.

While you can provide a separate userland in many ways, I find that the simplest is to slightly modify the jail(8) construction process from Chapter 22. First, make a dataset, UFS filesystem, or directory for our diskless clients to use as their root directory, and then install a userland and kernel in that directory. Extract the *base.txz* and *kernel.txz* distribution files for the version of FreeBSD in that directory.

```
# tar -xpf base.txz -C /diskless/1/
# tar -xpf kernel.txz -C /diskless/1/
```

If you've built a FreeBSD you want to run, that works too. Here, we install a locally built userland in */diskless/1*:

```
# cd /usr/src
# make installworld DESTDIR=/diskless/1
# make installkernel DESTDIR=/diskless/1
# make distribution DESTDIR=/diskless/1
```

Now tell your NFS server about this directory. I intend to install several diskless systems on this network, so I offer this directory via NFS to my entire subnet. The clients don't need write access to the NFS root, so I export it read-only. The following */etc/exports* line does this:

```
/diskless/1 -ro -maproot=0 -alldirs -network 198.51.100.0 -mask 255.255.255.0
```

Restart mountd(8) to make this share available, and try to mount it from a workstation. Confirm that the directory contains a basic userland visible from the client and that clients can't write to the filesystem.

Your diskless host needs a root password. Set it using chroot(8) and passwd(1).

```
# chroot /diskless/1/ passwd
```

You'll need to tell the host that its root filesystem is read-only. Create */diskless/1/etc/rc.conf* and set `root_rw_mount` to **`NO`**. While you're in that directory, also create a *resolv.conf* for your client.

Now reboot your diskless client and see what happens. It should find the kernel and boot into an unconfigured multiuser mode. Depending on the server, client, and network speed, this might take a while to complete.

At this point, you could configure your userland to specifically match your single diskless client. You could make changes in */etc*, such as creating */etc/fstab* that reflects your needs, and copy password files into place. That suffices for one diskless client, but FreeBSD has infrastructure designed specifically to support dozens or hundreds of hosts off the same filesystem. Let's look into how this is done.

Diskless Farm Configuration

One of the benefits of diskless systems is that multiple machines can share the same filesystem. However, even on machines that are mostly identical, you'll probably find that you must make certain configuration files slightly different. FreeBSD includes a mechanism for offering personalized configuration files on top of a uniform userland by *remounting* directories on tmpfs(5) temporary filesystems and copying custom files to these partitions.

FreeBSD's default diskless setup lets you configure diskless workstations across multiple networks and subnets—an invaluable feature on large networks. If you have only a few diskless systems, however, you might find it slightly cumbersome at first. Over time, however, you'll find that you make more and more use of it. Diskless systems are a convenient solution to many problems.

A booting FreeBSD system uses the `vfs.nfs.diskless_valid` to see whether it's running diskless. If the sysctl equals `0`, it's running off a hard drive; otherwise, it's running diskless. On diskless systems, FreeBSD runs the */etc/rc.initdiskless* script to parse and deploy the hierarchical diskless configuration.

Configuration Hierarchy

Configure your diskless farm in the diskless host's */conf*. The */conf* directory can have a whole bunch of directories in it. The two critical ones are */conf/base* and */conf/default*, but you might also have separate directories for subnets and/or individual IP addresses. Diskless systems use the contents of these directories to build tmpfs filesystems on top of the mounted root partition so individual hosts can have unique settings and read-write filesystems. You can make any directory a tmpfs filesystem and populate it from this hierarchy, but every host needs a read-write */etc* directory, so we'll use that as our example.

The */conf/base* directory contains base system files that need to be mounted read-write on the diskless client. Create */conf/base/etc* and populate it with a set of */etc* files, and the diskless host can use them as the base of its tmpfs */etc*. (It can also recycle the diskless root's */etc*, as we'll see later.)

The */conf/default* directory contains defaults for your environment. Perhaps every host in your environment needs an */etc/fstab* that directs it to mount a shared data store. You'd create */conf/defaults/etc/fstab*, and the diskless system would copy that to every host on top of the base system from */conf/base/etc*. I'd also distribute your environment's generic *rc.conf* in the default directory.

You can also have per-subnet directories. Name that directory after the subnet's broadcast address, the top address in the network. My diskless farm runs on the subnet 198.51.100.0/24, with a broadcast address of 198.51.100.255. If I created */conf/198.51.100.255/etc/rc.conf*, every host in that subnet would get that *rc.conf*. If I had a special */etc/fstab* for diskless hosts on that subnet, I could put it in */conf/198.51.100.255/etc/fstab* and it would overwrite the default. I'd also add files in */etc/rc.conf.d/* for special services that run only on that subnet.

Finally, I could have per-host directories. If I created

/conf/198.51.100.101/etc/rc.conf.d/apache, the host 198.51.100.101—and only that host—would get that file. If that particular host needed a truly unique */etc/fstab*, I could put it in */conf/198.51.100.101/etc/fstab*, and it would overwrite both the default and the subnet */etc/fstab*.[2]

This hierarchical configuration gets deployed through a process called *diskless remounting*.

Diskless Remounting /etc

The diskless system checks the file */conf/base/etc/diskless_remount* for a list of directories it should mount as memory filesystems. Without this file, no memory filesystems get created, and your diskless host shares a single read-only userland with all of the other diskless hosts. The `diskless_remount` file contains a list of filesystems to be remounted.

```
/etc
```

This tells FreeBSD to build an MFS */etc* and copy the diskless root's existing */etc* onto it, giving us a base to work from.

You don't necessarily want all of the files in the diskless root's */etc* on your diskless host's */etc*. It's a memory filesystem, so why waste memory holding stuff you don't need? You also don't want to imply to junior sysadmins that the hosts support functions that they don't. Diskless systems shouldn't keep logs locally, so they don't need *newsyslog* or */etc/newsyslog.conf*. You don't back up diskless clients, so */etc/dumpdates* is also unnecessary. Browsing */etc* will reveal quite a few files irrelevant to diskless hosts. If you remove too much, however, your system won't boot, and the list of necessary files isn't intuitive. For example, if you remove */etc/mtree*, the machine will hang in single-user mode because it can't repopulate the MFS */var* partition.

Put the full paths to your unwanted files and directories in the file */conf/base/etc.remove*. For example, the following entries remove the */etc/gss* and */etc/bluetooth* directories as well as the syslog and backup files discussed earlier. You don't need to copy over */etc/resolv.conf*. FreeBSD's */etc/rc.d/resolv* startup script creates one from the original DHCP response that booted the host.

```
/etc/gss
/etc/bluetooth
/etc/dumpdates
```

```
/etc/dumpdates
/etc/resolv.conf
/etc/newsyslog.conf
/etc/syslog.conf
```

Not so hard, is it?

Now let's put some things back into our configuration.

Finalizing Setup

Now that you have an installed system, let's do some fine-tuning. Diskless clients need third-party packages and assorted configuration files. The easiest and safest way to finish setting up your client is through using the chroot(8) program, which locks you into a subdirectory of the filesystem. By using chroot(8) on the NFS server, you can get read-write access to the filesystem almost exactly as it will exist on the diskless client.

```
# chroot /diskless/1
```

Yes, */etc* still has hierarchical overrides, but other parts of the system exist exactly as the diskless client sees them. Any changes you make while chrooted will be coherent to the client.

Installing Packages

Use pkg(8) to install software on a diskless client. Use the `-c` flag to specify the diskless root directory and have pkg(8) chroot into it.

```
# pkg -c /diskless/1/ install pkg
```

You now have the package tools, database, and repository information on your diskless client.

```
# pkg -c /diskless/1/ install sudo
```

Install any software you need this way.

SSH Keys

Perhaps the most annoying thing about diskless clients is the host's SSH

keys. In normal operation, every host needs unique SSH keys. If you're running on a private network, you might decide to have all the diskless clients share the same SSH key. You might decide to have each host autogenerate new SSH keys at boot time. As */etc* exists on tmpfs, those keys will vanish at shutdown, but users will quickly grow accustomed to the "host key has changed" messages. That's not something you want users to grow accustomed to, though.

Establishing persistent, unique host keys for each diskless client, however, isn't hard. Create a */conf* directory for each host.

```
# mkdir -p /diskless/1/conf/198.51.100.101/etc/ssh
# cd /diskless/1/conf/198.51.100.101/etc/ssh
```

In this directory, create the SSH keys for each algorithm your version of SSH uses. While ssh-keygen(1) includes the `-A` flag to autogenerate missing keys, it places those keys in */etc/ssh*. That won't work for your diskless userland or even in a chroot. You'll need to create those keys the old-fashioned way.

```
# ssh-keygen -N "" -qt algorithm -f ssh_host_algorithm_key
```

You'll need to substitute the name of the cryptographic algorithm twice, in lowercase. For example, here's how you'd create a DSA SSH key:

```
# ssh-keygen -N "" -qt dsa -f ssh_host_dsa_key
```

Today, OpenSSH creates keys for RSA, ECDSA, and ED25519. Create each of those. Key creation is easily scriptable. See */etc/rc.d/sshd* for examples.

Diskless clients let you easily run thousands of nearly identical machines. Now let's look at protecting just one.

Storage Encryption

FreeBSD supports two different disk encryption methods, GBDE and GELI. Both tools work very differently, support different cryptographic algorithms, and are designed for different threat models. People talk about encrypting disks all the time, but you rarely hear discussions of what disk encryption is supposed to protect the disk from.

GBDE, or *Geom-Based Disk Encryption*, has specific features for high-security environments where protecting the user is just as important as concealing the data. In addition to a cryptographic key provided by the user, GBDE uses keys stored in particular sectors on the hard drive. If either key is unavailable, the partition can't be decrypted. Why is this important? If a secure data center (say, in an embassy) comes under attack, the operator might have a moment or two to destroy the keys on the hard drive and render the data unrecoverable. If the bad guys have a gun to my head and tell me to "enter the passphrase or else," I want the disk system to say, `The passphrase is correct, but the keys have been destroyed`. I don't want a generic error saying, `Cannot decrypt disk`. In the first situation, I still have value as a blubbering hostage; in the latter, either I'm dead or the attackers get unpleasantly creative.[3]

GELI is much more flexible, but it won't protect me from bodily harm the way GBDE might. If someone might steal my laptop for the confidential documents on it, or if an untrusted system user might snoop my swap space to steal secrets, GELI suffices. GELI doesn't try to protect my person, just my data. As I won't take any job that poses a higher than average risk of exposure to firearms (keeping in mind that I live in Detroit), that's perfectly fine with me. GELI also uses FreeBSD's cryptographic device driver, which means that if your server has a hardware cryptographic accelerator, GELI takes advantage of it transparently.

I should mention that people lose more data to encryption misconfiguration or lost keys than to laptop theft. When I hear someone say, "I've encrypted my whole hard drive!" I have a nearly psychic vision of the future where that same person is saying, "I've lost access to everything on my hard drive!" More often than not, I'm correct. Consider carefully whether you really, truly *need* disk encryption. If you do need it, also back your files up. Those government spooks aren't going to crack the encryption on your laptop. They're going to wait for you to decrypt it yourself—and then they'll break in.

If you want to encrypt your laptop, use the FreeBSD installer to do so. You should still read this section so you understand how the disk encryption works, but if the installer wants to do the work for you, let it. We'll walk through using GELI to encrypt a disk partition on */dev/da0*, storing the cryptographic keys on the USB storage device mounted on */media*. You

might find it more sensible to use a filesystem in a file (see Chapter 13) as an encrypted partition. Very few people actually need to encrypt their entire hard drive, and in certain circumstances, doing so might raise suspicions. I have enough trouble explaining to airport security why my computer "looks so weird." In their minds, a boot prompt that says, `Insert cryptographic key and enter cryptographic passphrase` is only one step away from `This man is a dangerous lunatic who requires a very thorough body cavity search`. If you really do need to encrypt certain documents, chances are they total only a few megabytes. That's a perfect application for a filesystem in a file or a flash drive.

Note that you must load the *geom_eli.ko* kernel module before working with GELI.

Generating and Using a Cryptographic Key

GELI lets you use a key file and/or a passphrase as cryptographic keys for an encrypted device. We'll use both. To generate your cryptographic key file, use dd(1) to grab a suitable amount of data from */dev/random* and write it to a file. Remember, */media* is where our USB device is mounted. If you really want to protect your data, create your key directly on the USB device and don't leave it on your filesystem where a hypothetical intruder could recover it. (Even deleting the file still leaves remnants that a skilled attacker could conceivably extract.)

```
# dd if=/dev/random of=/media/da0p1.key bs=64 count=1
1+0 records in
1+0 records out
64 bytes transferred in 0.000149 secs (429497 bytes/sec)
```

The 64 bytes of data constitute a 512-bit key. You can increase the size of the key if you like, at the cost of extra processor overhead when accessing the encrypted filesystem. Don't forget that your passphrase also increases key complexity.

To assign a passphrase to the key, use `geli init`. The `-s` flag tells geli(8) the desired sector size on the encrypted filesystem; 4,096 bytes, or 4KB, is usually a decent sector size for this application. The `-K` indicates the key file. You must also specify the device to be encrypted.

```
# geli init -s 4096 -K /media/da0p1.key /dev/da0p1
Enter new passphrase:
Reenter new passphrase:
```

A passphrase is much like a password except that it can contain spaces and be of any length. If you really want to protect your data, I recommend using a passphrase that is several words long, contains nonalphanumeric characters, and is not a phrase in your native language.

Now that you have a key, attach it to the device to be encrypted.

```
# geli attach -k /media/da0p1.key /dev/da0p1
Enter passphrase:
```

GELI now knows that */dev/da0p1* is an encrypted disk and that the file */media/da0p1.key* contains the key file. Once you enter the passphrase, you can access the decrypted contents of the encrypted disk at the new device node, */dev/da0p1.eli*. Of course, you need a filesystem to put any data on that disk.

Filesystems on Encrypted Devices

Before you build a filesystem on your encrypted device, purge the disk of any lingering data. Programs like newfs(8) and zpool(8) don't actually overwrite most of the bits in a new partition; they simply add superblocks that indicate the location of inodes. If you've used this disk before, an intruder would be able to see chunks of old files on the disk. Worse, he'd see chunks of encrypted data placed there by GELI. Before you put a filesystem on the disk, it's best to cover the disk with a deceptive film of randomness to make it much more difficult for an intruder to identify which blocks contain data and which do not. Use dd(1) again:

```
# dd if=/dev/random of=/dev/da0p1.eli bs=1m
```

FreeBSD has an infinite supply of chaos—or, in technical terms, */dev/random* is nonblocking. The amount of time needed to cover the whole disk with high-quality randomness depends on your storage system. It might take a day.

Now that your disk is full of garbage, put a filesystem on it and attach it to your system. I'll often use UFS on such encrypted devices.

```
# newfs /dev/da0p1.eli
# mount /dev/da0p1.eli /mnt
```

Your encrypted disk device is now available on */mnt*. Store your confidential files there.

Encrypted disks have many more possibilities. Either read geli(8) or check out my book *FreeBSD Mastery: Storage Essentials* (Tilted Windmill Press, 2014).

This takes you through some of FreeBSD's murkier corners. Now let's see what to do when things go *really* wrong . . . .

24
PROBLEM REPORTS AND PANICS

FreeBSD is produced by human beings. Human beings make mistakes. Some of these mistakes are pretty trivial, while others can crash the whole system. FreeBSD explicitly has no warranty, but the community takes problems fairly seriously. Developers can't fix these problems without a proper bug report, however—and that's where you come in. Learning how to file a usable bug report will help you interact with not only the FreeBSD Project but also every other entity that produces software.

Bug reports need program output that demonstrates the bug, but what if the problem crashes the whole system? A system-halting *panic* is perhaps the most vexing of problems, but with proper preparations, you can deal with a panic as routinely as any less intrusive bug. The FreeBSD folks definitely want debugging output from your panic reports, and you can easily provide it in your problem report.

But first, useful bug reports.

Bug Reports

A *bug report* is a detailed description of a problem that causes the system to behave in an unexpected manner. That's kind of vague, yes. What's unexpected? What's a problem? There's a whole spectrum of valid problem reports, ranging from "I'd expect a man page for this thing" to "when I mount an SMB filesystem, the operating system crashes." Trivial problems like a missing man page reference might not seem worth your time to report, but every reference in every man page is there because someone thought it

was worth including. So what's a bug report, and what isn't?

A bug report isn't where you say that you have a problem. A bug report is where you prove that *FreeBSD* has a problem. Yes, I said *prove*. It's not that FreeBSD is innocent until proven guilty, but for you to submit a bug report, you must substantiate your claim. Bugs filed without evidence will be closed with terse replies, like "not a bug" or "useless report." The proper venue to cry for help is a search engine, followed by a mailing list or the FreeBSD forums.

Any variation on "I don't know what I'm doing" doesn't belong in a bug report. This includes "FreeBSD doesn't work the way I think it should" or "something bad happens when I do something dumb." If you start your new hobby of free climbing with a master-rated sheer cliff face, fall, and break your fool neck, the hospital kind of has to take you in. If you yank out a new hard drive halfway through running newfs(8) and complain about filesystem corruption, the FreeBSD folks will dismiss your bug.

Bugs can be about system inconsistencies. Every network interface, API, and system call has a man page. If you try to call up the man page for a system call and don't get a match, that's a bug. If you're reading source code and find a place where the documentation doesn't match the code, that's a bug. If you can make a program (or the whole system) reproducibly crash, that's a bug.

You can also file bugs to submit improvements to the FreeBSD Project. The key word here is *improvement*, not *wish*. An improvement needs actual code attached to the bug, along with how you've tested your code and any related information, such as specifications and standards. If you file enough of these, you might be invited to become a committer.

Bugs are collaborative. By filing a bug, you're indicating a willingness to work with the FreeBSD developers to resolve your issue. This might mean applying a patch, trying a different approach, or running debugging commands and sending the output to the developers. Filing a bug and expecting an answer like, "Fixed! Go do this," is unrealistic. Including everything in your initial bug report helps resolve the issue much more quickly. Err on the side of providing too much data.

Repeat after me: "Free software. Donated support time." A server's RAID card making your hard drives spin to a conga beat feels critical to you, but the people on the other end of the bug are giving up their personal time to

help you. Remember that.

It's best if you're running a recent version of FreeBSD when filing a bug. If you file a bug on FreeBSD 12.0-RELEASE when 12.4-p15 is the current version, someone will ask you to update and try again. Nobody will look at a problem report for a FreeBSD release past End of Life.

Before Filing a Bug

Ideally, you won't ever have to file a bug. Not only is a proper report for a serious bug a lot of work for you; it's a lot of work for the FreeBSD developers. The FreeBSD Project has an internal mailing list dedicated to assessing the bug database and guiding reports to their most likely owners. While sending an email to a FreeBSD mailing list announces your woes to thousands of people, opening a bug announces your woes to thousands of highly skilled people *and* demands that they handle virtual paperwork for you. Before filing a bug, be absolutely certain that both you and the FreeBSD Project need it.

Does the problem happen on all of your hosts or only on one? Problems restricted to a single host might result from failing hardware. Consistent, reproducible behavior is much more likely to be a bug.

Treat your issue as a general problem and search the usual FreeBSD resources. Review the FAQ and the Handbook. Check the FreeBSD bug database at *https://bugs.FreeBSD.org/* for an existing bug. Search the mailing list archives, the forums, and the wider internet for people who've already had this problem. Ask on the forums or the FreeBSD-questions mailing list whether anyone else has seen this behavior. Is this expected, or should you open a bug? The questions people ask will be invaluable in troubleshooting your problem and in creating your bug report.

Before starting a bug report, gather every scrap of information that might possibly be helpful. This includes:

- Verbose boot output
- System version
- Custom kernel configuration, if any
- Program debugging output
- What do you expect to happen?

- What actually happens?

Can you reproduce this problem? A developer investigating a bug needs a reproducible test case. If your server starts singing show tunes at 3 AM, that is a problem. If it happened only once and you can't reproduce it, you're best served by keeping your mouth shut so people don't think you're loony. If it happens whenever you run a particular combination of commands on certain hardware, though, the matter can be verified and investigated so that either the problem is resolved or someone offers your server a recording contract.

FreeBSD tracks bugs with Bugzilla at *https://bugs.FreeBSD.org/*. Before submitting a bug, search the existing bug database to see whether there's something similar or related. Does your problem resemble any existing bugs? If your server sings Disney tunes, but another bug shows someone's identical hardware does Broadway hits, you should probably mention that bug in your report. Are there any illuminating comments on those bugs? The comments might tell you how to cope with or work around your issue without filing another identical bug. If you want updates, add yourself to the bug's *cc* field. You'll get an email every time the bug is updated.

While you're searching Bugzilla, create an account there. Even if you don't need to file this bug, one day you will . . . and you'll probably be pretty annoyed about it. Having your Bugzilla account ready will make that submission just a little bit easier.

If you get this far and still have a problem, you might actually need to file a bug report. Let's see what not to put in it.

Bad Bug Reports

The easiest way to understand a good bug report is to read some bad ones and identify what makes them bad. Digging through the closed bugs uncovers bunches of bad reports, but here's an archetype:

> When I boot the FreeBSD 12.1 ISO image, I can't get past the "Welcome to FreeBSD" options screen. The boot menu is stuck, and each time the screen refreshes it stays at 10. It doesn't matter what I press, the system never boots. If I press a whole bunch of buttons, I eventually get a kernel panic. The same ISO image launches in VirtualBox and I can install it to

the disk.

The bug report includes the model number of a dead standard SuperMicro motherboard, keyboard, and mouse. None of the hardware is exotic. The reporter suggests reproducing the problem by booting the ISO with similar hardware.

First off, the reporter obviously has a problem installing FreeBSD. It might even be that FreeBSD has a problem. I've no doubt that this system fails at boot exactly as advertised. But there's no evidence and no diagnostic information. The reproduction process isn't very useful; if every 12.1 installation image behaved this way on common hardware, the release engineers would have never signed off on the release.

Including the hardware make and model isn't as useful as you might hope. Vendors occasionally change chipsets without changing the model number. The verbose boot information identifies the hardware in the machine in a way that the model number never can. This reporter can't get a 12.1 verbose boot, however.

If I experienced this behavior, I'd first try a second CD. Perhaps the first burned disk was bad. If the behavior persisted, I'd download a slightly older version of FreeBSD to see whether the problem exists there. If 12.0 fails, how about 11.0? I'd include the verbose boot information from the earlier version in my bug report. If the older version failed, I'd ask the FreeBSD-questions mailing list for further advice before filing a bug.

As you might guess, nobody follows up on bugs like this.

Many developers like fixing bugs. They enjoy digging through code and identifying subtle problems. What they don't enjoy is sorting through people's erratic bug reports; they expect to be paid for dealing with difficult people. Your goal is to file a bug so complete and compelling that a developer that's looking for a bug to work on will think you're easy to work with—and then, you need to actually *be* easy to work with.

The FreeBSD FAQ includes a joke by Dag-Erling Smørgrav: "How many -current users does it take to change a light bulb?" The answer is 1,169 and includes "three to submit (bugs) about it, one of which is misfiled under doc and consists only of 'it's dark.'" If your bug amounts to "it's dark," it's a bad problem report.[1]

SPECULATION VS. EVIDENCE

Whenever you submit a bug report to any person or organization, separate your evidence from what you *think* is going on. Evidence is actionable; your speculation is not. Including speculation doesn't hurt, but it needs to be clearly separated from the evidence. How many times have you received a support call from a user who claims he's having a particular problem, but, once you dig into the issue, it turns out that everything he told you is bogus and something totally unrelated is going on? Yeah. Don't be that user. Keep your speculation separate.

The Fix

The most important part of any bug report is the fix. How do you remedy the problem? Perhaps all you have is a workaround. "The program crashes if I do this, but I can run it and pipe the output through such-and-such and do well enough." That comment helps the next person to hit your bug.

When you hit a bug, take a look at the source code. Fixing typos in man pages or on the website isn't hard. If you're a programmer, a couple minutes of perusing the source might uncover the problem. If it doesn't, well, figuring out why the system behaves in this way will make you a better programmer and debugger.

Maybe you can't fix this bug. Letting people know the bug exists is still helpful. But by including a fix, your bug transcends a report and becomes a contribution to the community.

Filing Bugs

All bugs get filed and handled at *https://bugs.FreeBSD.org/*. FreeBSD has three categories of bug: ports, base system, and documentation. Use a *ports* bug for anything with add-on software. Use the *base system* for anything that gets installed with a basic FreeBSD install. Use the *documentation* category for problems with man pages, the FAQ, the Handbook, and the website.

Each brings up a slightly different web form. The fields needed for documentation and ports bugs are mostly subsets of the base system bugs, though, so we'll walk through filing a base system bug.

The web form includes several drop-down fields that let you steer your bug toward the right people. Your bug might get reassigned right after you file it, but that's okay; initially, you're looking for someone who understands what the heck you're talking about. You don't want a doc committer triaging system call issues or a source committer figuring out a port-packaging problem.

The `Component` field lets you select a part of the system the bug affects. The list of components varies over time, but selecting a component brings up a description. While there's always a catch-all field, such as `Bin` for base system bugs, making a good choice will accelerate handling of your bug.

In the `Version` field, select the FreeBSD version this bug applies to.

The `Severity` field is both somewhat misleading and requires a little detachment from your own emotions. The choices are "Affects Only Me," "Affects Some People," and "Affects Many People." A bug terrible enough that you consider gnawing off your own foot to escape might affect only you. It's critical to you, but not to the FreeBSD project. Resist the urge to declare that it affects everyone. Similarly, a typo on the website might be visible to everyone, but if nobody's noticed until now, it's probably not worth "Affects Many People." Reserve the more important severity levels for bugs that negatively impact all users of a particular device driver or anyone who uses a certain filesystem. If you get a reputation for filing trivial reports as critical, you'll quickly find yourself being ignored. The FreeBSD Project works on the honor system, and reputation counts for more than you might think.

In the `Hardware` field, select the platform you found the bug on. Even if that seems irrelevant, it might be critical.

The `OS` field is vestigial from Bugzilla. Ignore it.

Below these drop-downs, Bugzilla offers text fields. These need a little more thought.

The `Summary` takes a brief description of the problem. A good summary should provide unique information to make your bug stand out from other bugs. "Panic when unmounting an SMB filesystem" is decent. "Can't install," "system broken," and "problem" are terrible. A developer perusing the bug database will see your summary first. A bad summary will encourage

him to gloss right over it.

The `Description` area is where you get to describe the problem. Don't rant and rave about how awful everything is. Say what happened and what you expected to happen. Include debugging output, if it's short enough to fit reasonably; otherwise, add the debugging output as an attachment. Include advice on how to replicate the problem. If you have a fix, give it. Add in anything you've discovered about the problem. Sometimes, the most unusual detail provides the vital clue.

Add `Attachments` beneath the description. This is where you can upload your custom kernel configuration, verbose boot messages, kernel panic messages, and lengthy diatribes.

Use the **Preview** button to verify you included everything you thought you did. Once the bug looks correct, hit **Submit**.

After Submitting

You'll shortly receive an email stating that you're now the proud owner of bug number such-and-such. Any response you make to that email gets attached to that bug, so long as you don't change the subject.

A high percentage of bugs that include the proper information get closed quickly. Complex or elusive bugs might take longer, but if you provide enough detail, you'll see updates.

If it seems that your bug report has been forgotten, drop a note to the appropriate mailing list with your bug number, a brief explanation of the issue, and a sentence or two on why it's important. FreeBSD is a volunteer effort, and it's possible that something happened to the person who would normally handle that bug report. While many FreeBSD developers are professional programmers, for many of them, this is still a hobby that must take a backseat to sick kids or the big work deadline. If nothing else, hire a developer on a contract basis to address your particular issue.

If you file a notably vexing bug, a FreeBSD developer will probably ask you for more information. Provide it as quickly and thoroughly as possible. If you don't understand what they're asking for, spend some time researching and then ask. Most developers are happy to provide pointers to a willing and basically competent partner, especially if you can help them improve their code.

I've lost count of how many FreeBSD bugs I've filed. I tend to file either trivial or serious bugs, such as documentation errors and kernel panics, but very little in between. Most were solved and/or committed and then closed. The odd ones were mostly trivial goofs on documentation that lives under */usr/src/contrib*, an area where the FreeBSD Project specifically disavows responsibility for minor fixes. If a doofus like me can get over 90 percent of his bugs successfully closed, anyone can. Be warned, however: if you submit enough correct patches, you'll find that the committers you work with will start to talk about you behind your back. Eventually, they'll grow tired of acting as the secretary for your high-quality work and offer you commit access. If you refuse, they'll offer more insistently. Don't worry; becoming a committer isn't that painful. The rumors that the FreeBSD Project initiation ritual involves a bunch of Danes with axes behind a bike shed are completely untrue. Mostly.

Keep filing good bug reports anyway; that's the only way FreeBSD improves!

The worst sort of bug to deal with is a full-on system crash. Let's talk about how to get information from one.

System Panics

A *panic* is when the operating system completely stops working. All systems, from the network stack to the disk drive, stop working. A system chooses to panic, or completely stop working, when the kernel faces an unresolvable conflict. If the system achieves a condition that it doesn't know how to handle, or if it fails its own internal consistency checks, it throws up its hands and says, "I don't know what to do!" A panic is the kernel's version of malicious obedience.[2] Production versions of FreeBSD are increasingly difficult to panic, but it can still happen. The easiest way to panic a system is to do something daft, like yank out a non–hot swappable hard drive while it's in use. Panics aren't uncommon when running -current; they're not frequent, mind you, but they're not exotic rarities.

FreeBSD is very complex, and neither its royal blood lineage nor the open source development process can protect it from all bugs. Fortunately, that heritage and the development process do give you the tools you need to provide the information for other people to debug your problem. You might

begin with a cryptic error code, but you'll quickly learn that your string of garbage characters means something to someone.

A panicking kernel can copy critical information into a *crash dump*. The crash dump contains enough information about the panic that hopefully a developer can identify and fix the underlying problem. Configure every system to capture crash dumps before you allow them to enter production. FreeBSD can capture crash dumps with an install-time setting, but if you reconfigure your server or have unique disk partitioning, you'll want to confirm that crash dumps still work. This precaution will be wasted on most of your servers but pays off when something explodes.

Recognizing Panics

When a system panics, it stops running all programs, writing to the disk, and listening to the network. On any version of FreeBSD except -current, a panicking system automatically reboots. Not all unexplained reboots are panics—bad hardware, such as a failing power supply or cruddy memory, can cause a reboot without any sort of log or console message. If you're running -current, though, a panic will cause a console message much like this:

```
panic: Assertion cp->co_locker == curthread failed at /usr/src/sys/modules/smbfs/../..
/netsmb/smb_conn.c:363
cpuid = 5
KDB: stack backtrace:
db_trace_self_wrapper() at db_trace_self_wrapper+0x2b/frame 0xfffffe085d0db630
vpanic() at vpanic+0x182/frame 0xfffffe085d0db6b0
kassert_panic() at kassert_panic+0x126/frame 0xfffffe085d0db720
smb_co_unlock() at smb_co_unlock+0x9c/frame 0xfffffe085d0db740
smb_co_put() at smb_co_put+0x68/frame 0xfffffe085d0db770
nsmb_dev_ioctl() at nsmb_dev_ioctl+0x484/frame 0xfffffe085d0db800
devfs_ioctl_f() at devfs_ioctl_f+0x15d/frame 0xfffffe085d0db860
kern_ioctl() at kern_ioctl+0x230/frame 0xfffffe085d0db8c0
sys_ioctl() at sys_ioctl+0x17e/frame 0xfffffe085d0db9a0
amd64_syscall() at amd64_syscall+0x2de/frame 0xfffffe085d0dbab0
Xfast_syscall() at Xfast_syscall+0xfb/frame 0xfffffe085d0dbab0
--- syscall (54, FreeBSD ELF64, sys_ioctl), rip = 0x800b8016a, rsp = 0x7fffffffe4f8,
rbp = 0x7fffffffe530 ---
KDB: enter: panic
db>
```

The only part of this message that seems even vaguely sensible to me is the first line. I unmounted an SMB filesystem and got this panic message. The first line mentions `smbfs`, `netsmb`, and `smb_conn`, which seem pretty SMB-ish

to me.

The `db>` at the bottom is a debugger command prompt. Hit ENTER a couple of times and you'll see the debugger respond; you can enter commands. Debugger instructions aren't Unix commands, but they help you extract more information out of the system.

Responding to a Panic

If you get a panic, the first thing to do is get a copy of the panic message. Since FreeBSD is no longer running, the standard methods for copying data from your machine won't work—you can't SSH in, and script(1) is no longer viable. The console might even be completely locked up and unresponsive instead of being in the debugger. In any event, you must have that error message.

Back in the bad old days of the 1990s, FreeBSD didn't automatically reboot after a panic; originally, it just sat there displaying the panic message. The first time I saw a panic, I scrambled for paper and pen. Eventually, I found an old envelope and a broken stub of pencil that made marks if you held it at just the right angle and crawled between the server rack and the rough brick wall. I balanced the six-inch black-and-white monitor in one hand, and with my other hand, I held the envelope against the wall. Apparently, I grow a third hand under duress because I recorded the panic message on the envelope somehow. Finally, scraped and cramped, I slithered back out of the rack and victoriously typed the whole mess into an email. Surely the FreeBSD Project's Panic Emergency Response Team would be able to look at this garbage and tell me exactly what had happened.

I quickly learned that FreeBSD has no elite PERT standing by to take my problem report. Instead, I got a lonesome email: "Can you send a backtrace?" When I asked how, I was directed to a man page. (Drag yourself all the way back to Chapter 1.) Fortunately, the panic was easily reproducible—the only thing that had to happen to recreate the issue was for a customer to log in to the system. I spent the rest of the day struggling to master serial consoles and core dumps.

The problem with the panic message on my envelope was that it gave only a tiny scrap of the story. It was so vague, in fact, that it was like describing a stolen car as "red, with a scratch on the fender." If you don't

give the car's make, model, VIN, and license plate number, you can't expect the police to make much headway. Similarly, without much more information from your crashing kernel, the FreeBSD developers can't catch the criminal code.

The good news is, panic handling has vastly improved since those days. FreeBSD can automatically record crash dumps and capture everything about a panic. There's even a toggle in the installer to enable it. I highly recommend testing the panic capture before putting a machine into production, however. This way, if you get a panic, you're ready and you'll be able to easily file a complete problem report.

Preparations

Configuring crash dumps requires telling FreeBSD which swap device to save the dump on, through the `dumpdev` */etc/rc.conf* variable. If you set `dumpdev` to `AUTO`, the kernel will automatically save the dump to the first swap device. You can specify a different swap device if needed, but the whole dump must fit in a single swap device. If your regular swap space doesn't have enough space to hold the dump, add a disk to get sufficient swap space and set `dumpdev` to that partition.

The Crash Dump in Action

When a system configured to capture panics crashes, it saves a copy of the kernel memory. The copy is called a *dump*. The system can't save the dump straight to a file. The crashed kernel doesn't know anything about filesystems, for one thing, and the filesystem might be corrupt or a write could corrupt it. A crashed kernel understands partitions, however, so it can write the dump to a partition. Most FreeBSD hosts have readily available scratch space—the swap partition. FreeBSD defaults to dumping on the first swap partition on the system, placing the dump as close to the end of the partition as possible. After the dump, the computer reboots.

After a panic, a host's filesystems will almost certainly be dirty. Perhaps they're ZFS or they use soft updates journaling, but the system still must recover from the journal or roll back to the last successful ZFS transaction group. Cleaning a filesystem with fsck(8) can use a lot of memory, so FreeBSD must enable swap before running fsck(8). Hopefully, you have

enough memory for fsck(8) to not require swapping, and if swapping is necessary, hopefully you have enough swap space to avoid overwriting the dump file lurking at the end of the swap partition. Worst case, you could boot into single-user mode, enable swapping to partitions that don't have the dump, clean up a filesystem to save the dump to, and then manually run savecore(8).

Once FreeBSD has a useful filesystem where it can save a core dump, it checks the swap partition for a dump. If it finds a core dump, FreeBSD runs savecore(8) to copy the dump out of swap and into a proper filesystem file, runs crashinfo(8) to gather information from the dump, clears the dump from swap space, and continues rebooting. You now have a kernel core file usable for debugging.

The savecore(8) automatically places kernel dumps in */var/crash*. Each crash is in a file called *vmcore* with a trailing period and number. The first panic is *vmcore.0*, the second *vmcore.1*, and so on. FreeBSD defaults to keeping the most recent 10 crash dumps. The file *vmcore.last* always points to the most recent crash dump.

If your */var* partition is not large enough to contain the dump, choose a different directory with the `dumpdir` variable in *rc.conf* :

```
dumpdir="/usr/crash"
```

While savecore(8) also supports a few other options, such as compression, they aren't usually necessary on modern systems.

FreeBSD defaults to running crashinfo(8) on each recovered crash dump. The crashinfo(8) program runs a series of debugger scripts to gather information from the panic, storing it in a convenient text file, *core.txt.0*. This information includes a panic backtrace, process list, and a whole bunch of virtual memory statistics.

SERIAL CONSOLES AND PANICS

While a serial console isn't strictly necessary for panic debugging, it can be invaluable when dealing with a stuck machine. While a Java applet that grants remote access is better than nothing, the ability to capture everything with

script(1) makes serial consoles worthwhile. If you really want to be prepared for a panic, make sure all of your machines have serial consoles or at least dual consoles. If possible, log the output of your serial consoles; this way, you'll get the panic message even if the system isn't configured for crash dumps. If your laptop doesn't have a serial port, take a photograph of the panic message and attach it to the bug report.

Testing Crash Dumps

You've set the `savecore` *rc.conf* option, so everything should work. Any time you hear the phrase "should work," immediately ask, "How can I verify that it does work?" Force FreeBSD to panic by setting the sysctl `debug.kdb.panic` to any integer above 0. While this is the ugliest way to reboot a machine ever, it does make the host run through the panic and core-preservation process. Shut down any active processes that might corrupt data if interrupted, such as databases, and deliberately trigger a panic.

```
# sysctl debug.kdb.panic=1
```

You'll see the panic message flash on the console, followed by the host's progress in dumping core to swap space. If you watch the reboot messages, you should see a quick mention of saving core files. When you can log into the machine again, take a look in */var/crash*. You'll find three files: *info.0*, *vmcore.0*, and *core.txt.0*.

The *info.0* text file describes the dump recovery process. It includes the hostname, the architecture, the panic message, and more. The most important detail is the last line, however.

```
Dump Status: good
```

This dump is usable. You can proceed with debugging.

The file *vmcore.0* contains the memory dump in binary form. It should be anywhere from a couple hundred megabytes to gigabytes, depending on what your host was doing when it panicked.

The file *core.txt.0* contains the panic information from vmcore.0. When you file a problem report, include the *core.txt* file for your panic.

Congratulations—you have a core dump! When your host actually panics, you can get information from the dump. Sometimes, though, the panic gets a little more complicated.

Crash Dump Types

FreeBSD supports three different sorts of crash dumps: minidumps, full dumps, and textdumps. All get written to swap space at a panic and copied to files at boot.

A *minidump*, the current default dump format, contains the memory used by the kernel. While the kernel itself isn't that big, you'll also get the UFS buffer cache. The swap space needed should be only a fraction of your system memory, but it's possible that it could be almost as large as your system memory, depending on what your system was doing at the time. The dump excludes both memory not used by the kernel and the ZFS ARC.

A *full dump* contains every scrap of memory the system has. If it's in RAM, it gets dumped. The whole kernel memory? Yep. Your web server's buffer? Passwords? It's all in there. A full dump takes up as much swap space as your host has memory. Enable full dumps by setting the sysctl `debug.minidump` to 0. Enable full dumps only if a FreeBSD developer asks you to do so to help debug a particularly intractable panic.

A *textdump* is an advanced type of dump that contains only the information captured by the ddb(8) debugger and associated scripts. It's available only on hosts with the DDB option in their kernel—as in, not the GENERIC kernel of any release. It's in the -current GENERIC kernel, though, so you brave souls running -current can take advantage of textdumps.

Textdumps

A textdump takes advantage of the ddb(8) debugger to run scripts on a panicked kernel. The default scripts in */etc/ddb.conf* pull the most commonly needed information out of the kernel and then dump that information to disk. While crashinfo(8) runs on a captured memory image, though, ddb(8) runs on the panicked kernel. An experienced developer can take advantage of

this. You might not be an experienced developer, but if you've made it this far into this book, you can follow directions and edit */etc/ddb.conf*, and that's close enough.

Enable textdumps with the `ddb_enable` *rc.conf* option.

```
# sysrc ddb_enable=YES
```

At system boot, the kernel debugger ddb(8) reads the debugging scripts from */etc/ddb.conf* and loads them into the kernel. The debugger runs those scripts at a panic. The scripts switch the kernel to textdump mode, call several commands to gather useful information, write that data to swap space, and reboot the host. Textdumps aren't as useful as minidumps, but they fit in much less space.

A complete textdump shows up in */var/crash* as a tar file, *textdump.tar.0*. The textdump number gets incremented at each panic, and the file *textdump.tar.last* always points to the most recent textdump. The textdump(4) man page describes each of the files found inside the tarball, but as a user, what you really need to know is that you attach the whole thing to your bug report.

Dumps and Security

The *vmcore* file contains everything in your kernel memory at the time of the panic, which might include sensitive security information. Someone could conceivably use this information to break into your system. A FreeBSD developer might request a copy of the *vmcore* file and the bad kernel for many legitimate reasons; it makes debugging easier and can save countless rounds of email. Still, carefully consider the potential consequences of someone having this information. If you don't recognize the person who asks or if you don't trust him, don't send the file and *don't* feel bad about it. Take the time to research any developer who wants your vmcore. Even if they seem reputable and respected, it's perfectly acceptable if you decide to work via emails back and forth rather than offer up the vmcore. Anyone qualified to work on your crash understands why you hesitate to send a vmcore, and anyone who tries to shame you into sending it probably shouldn't have it.

Posting a link to your vmcore on the public internet offers the guts of your server to the whole world. Don't do that.

If the panic is reproducible, however, you can cold-boot the system to single-user mode and trigger the panic immediately. If the system never starts any programs containing confidential information and nobody types any passwords into the system, the dump can't contain that information. Reproducing a panic in single-user mode generates a security information–free, sanitized dump. Boot into single-user mode and then run:

```
# mount -ar
# /etc/rc.d/dumpon start
# command_that_panics_the_system
```

The first command mounts the filesystems as read-only so that you won't have to fsck(8) yet again after a panic. The second command tells FreeBSD where to put a dump. Finally, run the command that triggers the panic. Triggering the panic might require more than one command, but this should get a clean dump for you in most cases.

If your panic requires that you load confidential information into memory, that information will be present in the dump. If you can file a useful bug report, you're among the FreeBSD elite. Congratulations!

AFTERWORD

If you've made it this far, you now know how to manage and use FreeBSD as a platform for just about any server task. You might have to learn new protocols and how to configure new programs, but the underlying operating system is pretty much a solved problem. Congratulations! FreeBSD is a wonderful, flexible platform, capable of assuming just about any role in your network. To wrap things up, I'd like to discuss some other aspects of FreeBSD.

We've talked about FreeBSD's features throughout this book: the programs, the kernel, the features, and so on. One thing we haven't covered is the community that creates all this.

The FreeBSD Community

The FreeBSD community includes computer scientists, experienced programmers, users, system administrators, documentation writers, and just about anyone interested in the system. They come from countries all around the world and have education levels ranging from high school to post-doctoral. I personally have had dealings with FreeBSD users from every continent and most of the larger islands on Earth.[1] Nationality simply isn't important—nor is race, color, gender, sexual orientation, or creed.

Some are computer scientists. Some work at cloud providers or manufacturing firms. Some are physicians, and some work as clerks in dubious little stores that disappear just before the government notices them. At one point, I worked closely with a brilliant developer who turned out to be too young to legally drive. Oddly enough, time zones are important, but

only because they impact the developers' ability to communicate with each other. Since most of the community's interaction is online, the only things that represent you are your words and your work. These are the people who improve FreeBSD and drive it forward, making it more than a collection of ones and zeros and more than just a way to serve websites.

One of the interesting things about the FreeBSD community is that it has developed methods for coping with changes in its leadership. Many open source projects have a single leader or a small static leadership team. When those people decide to move on, the project is probably over. Someone else might branch or fork that project, but the original community usually fragments. The people who created FreeBSD have mostly moved on to other things, but the community has grown other leaders. After five generations of leadership, FreeBSD as a project has demonstrated a resilience to leadership changes that is almost unique in the open source world. Today's FreeBSD leaders take a very active interest in their own replacements, mentoring and coaching those junior community members who seem most likely to become the leaders of the 2020s and and even the 2030s.

This ability to change leadership has kept FreeBSD as a community vital and helped it adapt to the world. In 1994, nobody thought an open source project would need a Code of Conduct or a process for discussing and managing architectural changes. FreeBSD now has both. Where once the Core Team handled all the central decisions, Core has delegated many of its responsibilities to more specific groups. All organizations change as they grow and mature. Change destroys many organizations, but FreeBSD has demonstrated that it can survive and prosper while redefining itself.

What's more, they always welcome those who came before, both in the Subversion repository and at the bar. The original author of the UFS filesystem still hangs around. One of FreeBSD's founders recently rejoined the project. Some people have been committers since the founding. And somehow, they've let me hang around since the mid '90s—probably because I'm too large to easily shift against my will, but I'll take it.

Why Do We Do It?

Each person works on FreeBSD for their own reasons. A tiny portion of

people are paid to improve the code by corporations dependent on FreeBSD, such as Dell EMC and Netflix. The FreeBSD Foundation hires developers to complete specific tasks on a contract basis. Most developers work on FreeBSD as a hobby, either so they can program things more correctly than they're allowed to at their day job or so they can do work that interests them. How many of you have completed work projects less successfully than you'd like because of outside influences? And how many of you have jobs that pay the bills but don't leave you feeling fulfilled? Developing FreeBSD allows people to satisfy both those itches.

Many contributors are not software developers but work on some other part of FreeBSD instead. Some write documentation, some design the websites, some just lurk in shadowy alleys of the forums and answer user questions. People who can't do anything else test release candidates and snapshots, hunting bugs that crop up only in their environments. Many people spend hours and hours working on FreeBSD-related matters. Why? I can assure you that the royalties on this book won't come close to compensating me for the time I could spend with my family. I'm a full-time writer now, but if I were to chase money, I'd write about Windows, Linux, and the latest cockamamie management snake oil. Instead, I'm writing a book about FreeBSD.

Worse, I'm writing this Afterword up in my office on a Saturday afternoon while my family is downstairs having a grand old time persuading the family of Canadian Red Squirrels that's conquered the barbeque grill to move back into the tree. There's yelling and squeaking and the occasional shout of "Oh, God, please no!"—so a good time is being had by all. You'd be within your rights to ask: "What is wrong with you? Why do you do this?"

We do it for the satisfaction of creating something useful to the rest of humankind and to return some of what we've been given.

You're free to simply take what FreeBSD offers and do whatever you wish with it. I did exactly that for a while. After a couple years, once I became a modestly competent sysadmin, I found that I wanted to return something to the community. This is how the community grows, and a growing community means that FreeBSD will prosper.

If you want some of that satisfaction yourself, there's a place for you too.

What Can You Do?

If you're interested in supporting FreeBSD, for whatever reason, there's space for you. Ever since I started with FreeBSD back in 1996, every so often, someone posts, "I'd like to help, but I can't code." (I'm pretty sure I sent that email to the *questions@* mailing list back in 1998 or so.) The standard response to these posts is silence. If you've already decided that you can't help, you're right—you can't. Once you decide that you can help, though, you can.

Nobody denies that some high-visibility programmers are the celebrities of FreeBSD. Many of those have impressive skills, and most of us could never dream of being the next Robert Watson or John Baldwin. Even if you can't program your way out of a damp paper bag, however, you can still help.

You're just asking the wrong question.

Don't ask what FreeBSD needs. You can't provide that, unless you have a large bank balance begging for a charitable cause to belong to. (If you do have spare piles of cash in desperate need of someone to nurture them, the FreeBSD Foundation would be happy to adopt them and cuddle them and make them feel loved.) Don't say, "Wouldn't it be cool if FreeBSD did such-and-such?" if you can't create that yourself. Lots of people can do that.

Instead, ask yourself what skills you have. Any large organization needs many different people, and whatever skills you have today are useful to FreeBSD. Can you write documents? Dive into the official documentation, and maybe port a popular tutorial from the forums to the official doc repository. Do enough of that, and the doc team will drag you into FreeBSD and brand a commit bit on your forehead.

Are you a web designer? Independent web designers provide valuable third-party resources, such as *https://freshports.org/* and *http://daemonforums.org/*. There's lots of room in this space, and you can fill it.

Is there third-party software you need that hasn't already been ported to FreeBSD? Bludgeon it into working and then turn it into an official port. FreeBSD is always looking for more maintained software. Once you've done a couple of those, you can adopt maintainership on ports you need that no longer have maintainers. Keep that up, and the Ports Team will come for you to make you a committer.[2]

I write copiously and passably well. I wrote some updates for the FAQ and then the first edition of the book. The FAQ updates made me a committer, although I let that lapse many years ago when I turned my attention to writing more books. The mere sight of code I've written drives small children to desperate sobs and compels sweet old ladies to make the sign to ward off the evil eye, but the FreeBSD folks welcome me as one of their own and treat me as a partner simply because I do the work.

What is it that *you* do? What is it that you enjoy doing, even if you don't get the chance to do it often? Leverage that skill. It will be appreciated.

If Nothing Else . . .

If you truly have no useful skills, and you have no other ideas, reread this book. Read the documentation on the FreeBSD website. Subscribe to *FreeBSD-questions@FreeBSD.org*, or join the forums, and help other users. Many people started contributing to FreeBSD in exactly this way.

I encourage you to direct people to existing information resources whenever possible. When someone asks a question answered in the FAQ, steer them there. If the question has been asked before, suggest that they search the mailing list archives. Teaching people to help themselves is the most effective use of your time—not just in FreeBSD, but in the world as well. As the old saying goes, teach a man to fish and you can sell him fishhooks.

Do enough of that and you'll want to update the FreeBSD FAQ just so you don't have to answer that question one . . . more . . . time. Submit enough FAQ updates, and once again, the doc team will offer you a commit bit.

Best of all, after answering questions for a while, you'll develop a deeper understanding of FreeBSD's needs. One of those needs will almost certainly match your skills.

Getting Things Done

Here's the big secret of success in FreeBSD: everything that it contains is there because somebody saw a need they could fill and did something about

it. NetBSD and FreeBSD started when a bunch of 386BSD patchkit users got sick of waiting for the next official release. I didn't ask for permission to write this book before starting. The fine folks over on *bugbusters@FreeBSD.org* don't wade through the bug database for fun; they do it because they think it's important enough to spend their time on. (And if you *are* a programmer, wading through the bug database and finding problems you can solve is one of the best contributions you can make.)

Once you have an idea, search the mailing lists for discussions about it. Many projects are suggested and debated but never implemented. If someone's brought up your idea, read the archived discussion. If the idea met with general approval in the last few years, but nobody's working on it, get to work! The FreeBSD folks will be perfectly content if the first time they hear from you is in a bug saying, "Hi, here are my patches to implement this feature, as discussed in such-and-such mailing list thread."

Whatever you do, don't go on the mailing list or forums to ask, "Why doesn't someone else do the work for X?" Most of these suggestions fall into three categories: obvious ("Hey, wouldn't it be cool if FreeBSD ran on Teslas?"), foolish ("Why isn't there a kernel option BRINGMEACOLDBEER?"), or both ("Why not support my Sinclair ZX80?"). In any of these cases, the person asking is both unqualified to perform the work themselves and claims to be helpless to support others who *could* do the work. All these suggestions do is waste bandwidth and annoy people. Bandwidth is cheap; people are not.

In short: shut up and work. Do what you can, and do it well, and people will appreciate it. Programmers can help by jumping into the bug database and picking a promising bug to attack. Nonprogrammers can help by finding a hole they can fill and doing the work to fill it in. You might become a leader in FreeBSD, or you might be known as "that awesome woman who hangs out on *-questions@* and helps people with EFI boot loaders." All are absolutely vital. Your help makes FreeBSD prosper. Stick around long enough, and what starts by helping people with EFI boot loaders might grow into representing the whole of FreeBSD.

I look forward to seeing you on the mailing lists.

BIBLIOGRAPHY

This bibliography contains two types of books: those I use as my own reference and those I've written that you can consult for more information.

References

These books helped me learn stuff I used in writing this book. You might find them educational.

Hansteen, Peter N. M. *The Book of PF: A No-Nonsense Guide to the OpenBSD Firewall*. 3rd ed. San Francisco: No Starch Press, 2015.

Kong, Joseph. *Designing BSD Rootkits: An Introduction to Kernel Hacking*. San Francisco: No Starch Press, 2007.

———. *FreeBSD Device Drivers: A Guide for the Intrepid*. San Francisco: No Starch Press, 2012.

Kozierok, Charles M. *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference*. San Francisco: No Starch Press, 2005.

McKusick, Marshall Kirk, George V. Neville-Neil, and Robert N. M. Watson. *The Design and Implementation of the FreeBSD Operating System*. 2nd ed. Upper Saddle River, NJ: Addison-Wesley, 2014.

Stevens, W. Richard, and Stephen A. Rago. *Advanced Programming in the UNIX Environment*. 3rd ed. Upper Saddle River, NJ: Addison-Wesley, 2013.

Books I've Written

Over the last 10 years, FreeBSD has grown. A lot. In writing *Absolute FreeBSD*, I attempted to strike a balance between explaining everything and writing a book small enough to hold in your hand. ZFS, for example, merits

a book the size of this one. I've tried to give you what you *must* know instead of everything there is *to* know.

You'll want more on some topics. I've written books that can give you that more. Mixing them in with the regular bibliography felt way too self-aggrandizing, so here's a separate list.

Jude, Allan, and Michael W. Lucas. *FreeBSD Mastery: Advanced ZFS*. Grosse Pointe Woods, MI: Tilted Windmill Press, 2016.

———. *FreeBSD Mastery: ZFS*. Grosse Pointe Woods, MI: Tilted Windmill Press, 2015.

Lucas, Michael W. *FreeBSD Mastery: Specialty Filesystems*. Grosse Pointe Woods, MI: Tilted Windmill Press, 2016.

———. *FreeBSD Mastery: Storage Essentials*. Grosse Pointe Woods, MI: Tilted Windmill Press, 2014.

———. *Network Flow Analysis*. San Francisco: No Starch Press, 2010.

———. *Networking for Systems Administrators*. Grosse Pointe Woods, MI: Tilted Windmill Press, 2015.

———. *PAM Mastery*. Grosse Pointe Woods, MI: Tilted Windmill Press, 2016.

———. *PGP & GPG*. San Francisco: No Starch Press, 2006.

———. *SSH Mastery*. 2nd ed. Grosse Pointe Woods, MI: Tilted Windmill Press, 2018.

———. *Sudo Mastery*. Grosse Pointe Woods, MI: Tilted Windmill Press, 2013.

INDEX

Symbols and Numbers

$ (cash symbol)
    and log date format, 555
    for username, 190
:: (double-colon substitution), in IPv6 addresses, 134
= or == (equal signs), 349
\ (line-continuation character), 306–307, 459
~ (tilde), for user's home directory, 190
4K drives, 202
32-bit compatibility libraries, 33
32-bit Intel-style processor, 17
32-bit number, 129
64-bit Intel-style processor, 17
386BSD, xxxvi

A

ABI (application binary interface), 413–414
ACK packet, 464
ACPI (Advanced Configuration and Power Interface), 59
active memory, 535
active slices, 223
add-on software, 336
addr keyword, for CIFS configuration, 312
adduser(8) program, 171
administrative group, creating, 182–184

aesni(4) kernel module, 482, 526
agent, in SNMP, 557
aggregation protocols, 163
AIX, xli
`alert` log message, 547
aliased mailing lists, 504
aliases
    for files, disabling, 237
    for IPv6 addresses, 147–148
    for network card, 68
    and outgoing connections, 148
    for pkg(8) subcommands, 349
`ALL EXCEPT` keyword
    for login restriction, 187
    for TCP wrapper, 458–459
`ALL` keyword
    for login restriction, 187
    for TCP wrapper, 458–459
`allow` option, for TCP wrapper rule, 459
`AllowGroups` option, for SSH, 496
`AllowTcpForwarding`, 495
`AllowUsers` option, for SSH, 496
Amanda, 87
AMD, xlii
amd64, 17
“ancient crap,” 20
Ansible, 64
a.out binaries, 408
Apache web server
    and httpd program, 456
    packages, 341
Apple, macOS, xl
applications, boot environment and, 279

applications (of OSI), 126
`apropos mount_`, 282
apropos(1), 5, 9–10
archives
    compression for, 91
    creating in tar, 88–90
    list mode for, 90
ARP (Address Resolution Protocol), 141–142
ARP table, 141–142
ashift property, 268
asynchronous mounts, 235–236
AT&T, xxxiv
    UNIX work, xxxv–xxxvi
atime (access-time stamp), disabling, 236
`attach` rules, for devd(8), 299
attachments, building kernel and, 106–107
attackers. *See* intruders
authoritative nameservers, 150
`autoboot_delay` option, 58
`AUTO_INSTALL`, for mergemaster, 447
automounting, 318
autonegotiation, by switch, 141
autoremoval of software, 350–351
`AUTO_UPGRADE`, for mergemaster, 447
avail memory, 61

B

background fsck, 66, 247–248
backslash (\), for line continuation, 306–307, 459
backups
    of jail, 564

system, 84
tapes for, 84–87, 106
verifying, 89–90
of working kernel, 107
Bacula, 87
bandwidth
and performance, 526
PF and, 467
`Banner` option, 495
base 2, 129
base 10, 129
base-dbg, 33
baseboard management controller (BMC), 76–77
`BATCH` environment variable, 374
`BATCH_DELETE_OLD_FILES` option, 445
baud rate, 74–75
`beadm activate` command, 278
`beadm create` command, 278
`beadm destroy` command, 279
beadm(8) program, 277
`beastie_disable` option, 58
`begemotSnmpdCommunityString` MIB, 561
Berkeley Software Distribution (BSD), xxxiv–xxxv, xxxvi
release of code, xxxvi
BGL (Big Giant Lock), 397–398
bhyve(8), 24, 564
developers, xlix
Big Giant Lock (BGL), 397–398
binary branding, 414
binary files
compression and, 273
for logs, 556

binary math, 129
binary updates, 428–434
    scheduling, 434
binary values, 99
BIOS (basic input/output system), 20, 50–51
bits, 129
`blacklistctl dump` command, 474
blacklistd(8), 319, 470–475
    configuring, 471–473
    configuring clients, 473–474
    de-blacklisting, 474–475
    managing, 474
blacklisting, 454
blocking on disk, 532
blocks
    in FFS, 232
    fsck(8) program to verify, 246
`$BLOCKSIZE`, 250
Bluetooth, 319
bonding, 163
boot blocks, 51. *See also* loader
/boot/defaults/loader.conf file, 16, 57
boot environment, 259, 276–279
    activating, 278
    and applications, 279
    at boot, 279
    creating and accessing, 277–278
    removing, 279
    viewing, 277
/boot/kernel directory, 97, 107
/boot/kernel.good directory, 107
boot loader, 30, 77
    for boot disk, 226

and kernel, 96
and tftpd(8), 590–591
/boot/loader file, 51
/boot/loader.conf file, 16, 57
boot menu, options, 58–59
boot messages file, 62
/boot/modules directory, 97
Boot Multi User loader option, 51
boot process, 49–82
/dev at, 295–297
configuring VLANs, 165
on legacy hardware, 222
loader, 51–52, 57
configuration, 57–58
loader prompt, 55–57
loading modules, 105
multiuser startup, 63–71
/etc/rc.conf.d/ file, 64–71
sysrc(8), 63–64
ntpd(8) in, 506
options, 58–59
power-on, 50–51
serial consoles, 74–82
configuration, 77–79
IPMI setup, 76–77
physical setup, 75–76
using, 79–82
single-user mode, 52–54
startup messages, 59–62
tmpfs(5) at, 289
boot-time tunables, 57, 101
testing, 103
BOOTP (Bootstrap Protocol), 588

bootstrap code, 223
`boot_verbose="NO"` option, 58
botnets, 169
bottlenecks, 545
    analysis with vmstat(8), 528–532
brandel(1), 417
branding software binaries, 417–418
`bridge` module, 562
broadcast address, 133
broadcast domain, 140
broadcast protocol, Ethernet as, 140
BSD 4.4-Lite, xxxvi
BSD (Berkeley Software Distribution), xxxiv–xxxv, xxxvi
    license, xxxv
BSD labels, 224, 227–229
BSD partitions, assigning specific letters, 228–229
bsdinstall(8), 31, 34
bsdlabel(8), 215
BSDstats Project, xxxix
bsdtar, 88
bsnmpd(8), 557
    configuring, 560–561
    loading modules, 562
buffer overflow, 168
bug reports, 599, 600–605
    actions after submitting, 605
    bad, 602–603
    before filing, 601–602
    filing, 603–605
    speculation vs. evidence, 603
Bugzilla, 601–602
buses, building kernel and, 106–107

bytes, 130
bzip compression, 91

C

CA (Certificate Authorities), 478–479
cables, for Ethernet, 140
caching nameserver, 153–154
camcontrol(8), 202–203
Capsicum security system, 319
`carmount` dataset property, 262
`ca_root_nns` package, 478
cash symbol ($), for username, 190
`cdrtools` package, 283, 286
CDs
    burning, 27
    filesystem for, 283
"certificate signer is unknown" warning, 481
certificates, 478–481
    creating request, 479–480
    self-signed, 480–481
`CFLAGS` (compiler flags) options, 324–325
CHANGES file, in Ports Collection, 365
checksum, 372
    SHA512/256, 486
chflags(1), 194
chmod(1) program, 183–184
chown(1) program, 183–184
chpass(1), 175–176
chrooting, 594
    tftpd(8) support for, 520
CIFS (Common Internet File System), 301, 310–315

configuring, 311
file ownership, 315
kernel modules for support, 311
mounting share, 313–314
name resolution, 313
prerequisites, 310
ciphertext, 475
Cisco switches, 163
class environment, 190–191
clean login, for jailed environment, 577
cleartext, 475
client list, for wrappers, 457–458
clients
access to NFS export, 307
configuring for blacklistd, 473–474
diskless, 588
MAC addresses for, 514
showing available mount for NFS, 309
for SSH, 497–498
clock synchronization at boot, 506
cloud-scale management, 586–587
clri(8) program, 247
clustering, disabling, 236–237
code freeze, 423
cold backup, 90
`comconsole`, 78
command line, xlvi
customization options, 374–375
command prompt, 47, 452
commands
question mark for listing, 55
running in jail, 576–578
service support for, 73

committers, xxxvii–xxxviii
    becoming, 605
Commodore 64, emulator, 413
Common Access Method (CAM), 202–203
common name, for server, 480
communities, for SNMP security, 560
comparison operators, in */etc/syslog.conf* file, 549
`COMPAT_FREEBSD32` option, for kernel, 115
components, connections, 61
`Components src world kernel` option, for freebsd update, 429
compress(1) command, 91
compressed installation media, 26–27
compression
    for archives, 91
    for log files, 556
    in ZFS, 273
Computer Systems Research Group (CSRG), xxxiv
/conf/base directory, for diskless farms, 593
/conf/base/etc/diskless_remount, 593
/conf/default directory, for diskless farms, 593
confidentiality, of cryptosystems, 475
`configtest` command, 73
configuration files, in packages, 337
connected protocol, 138
connectionless protocol, 137
connections, outgoing, and aliases, 148
console, 584
    insecure, 585
consumer, for geoms, 206
context switch, 396
continuation line, \ (backslash) for, 306–307, 459
CONTRIBUTING.md file, in Ports Collection, 365

contributors, xxxviii
cookies, 379
Coordinated Universal Time, 504
`COPTFLAGS`, 325
copy-on-write (COW), 270–271
copycenter, xxxv
copying files, over SSH, 498–499
COPYRIGHT file, in Ports Collection, 365
core settings, 30–32
    configuring, 46
    trimming kernel for, 113–115
corrupted files, 354, 372
cpio, 87
CPU
    bottleneck analysis with vmstat(8), 531
    and performance, 526
    threads waiting for, 529
    trimming kernel for type, 113
    usage, 543
CPU core, 399
`cpu` entry, in kernel configuration file, 108
CPU package, 399
crash dump, 608–609
    configuring, 608
    and security, 611–612
    swap partition for, 25
    testing, 609–610
    types, 610
crashinfo(8), 609
`crit` log message, 547
cron(8), 520–523
crontabs, 520
cryptography, 9–11

generating key, 597
CSRG (Computer Systems Research Group), xxxiv
CTM, 435
CUPS (Common Unix Printing System), 516
current resource limits, 190
customizable builds, xliii
customization files, 16
customization options
global, 375–376
Ports Collection, for command line, 374–375
setting default versions, 376–377
CVS (CVSup), 435
`CXXFLAGS`, 325

D

daemon, name in wrapper, 456
DAEMON provider, 404
data integrity. *See also* integrity check
and lost data, 245
database files
backup process and, 90
changes, 88
updating for mergemaster, 447
database software, shutdown and, 74
datagram protocol. *See* UDP (User Datagram Protocol)
datalink layer (of OSI), 125, 127–128, 138
datasets, 258–263
creating, 261
destroying and renaming, 261–262
moving files to new, 262–263
properties, 260–261
inheritance, 261

unmounted parent, 262
date
ISO 8601 time format, 555
for password changes, 176
DB9-to-RJ45 converters, 75–76
dd(1), 288, 597, 598
ddb(8) utility, 319, 610–611
deadlock, 399
deadly embrace, 399
`debug` log message, 548
debugging Linux mode, 418–420
debugging symbols, 33
`debug.kdb.panic` sysctl, 609
`DEBUG_LEVEL`, in *pkg.conf* file, 340
decimal math, 129
for computing netmasks, 132
default accept vs. default deny, 454–455
default config, not copying, 17
default directory, 16
default files, 16–17
default GPT partitioning, 36
default groups, 171, 184–185
DEFAULTS file, 109
`DEFAULT_VERSIONS` variable, 376–377
Defense Advanced Research Projects Agency (DARPA), xxxiv
`DEGRADED` pool state, 275
deleting
partitions, 216–217
shared libraries, 445
slices, 226
user accounts, 178
`deny` option, for TCP wrapper rule, 459

`DenyGroups` option, for SSH, 496
`DenyUsers` option, for SSH, 496
dependencies, 346
    for jails, 575
    packaged, 378
    in poudriere, 392
    removing, 350
desktop FreeBSD, xlvi
DESTDIR setting, 569
`detach` rules, for devd(8), 300
/dev, at boot, 295–297
/dev/console, 584
/dev/esa0 device node, 85
/dev/nsa0 device node, 85
/dev/pts, 584
/dev/sa0 device node, 85
/dev/ttyu, 584
/dev/ufs file, 239
devd(8) daemon, 320
    dynamic device management with, 299–300
devfs(5) program, 281, 295–300
    in jail, 570
devfs.conf(5), 297
`devfs_hide_all` rule, 297
`devfs.rules` file, 297
“device busy” error, 217
device daemon, 320
device drivers, 5
    Common Access Method (CAM), 202–203
    hints for, 102–103
    in kernel, 115
    loading, 105
    man pages for, 61

for proprietary hardware, 19
device name, for root partition, 53
device nodes, 62
filesystem for managing, 295–300
MBR, 224–225
permissions of, 296
for tape drives, 85
`DEVICE_POLLING`, 162
`devices` entry, in kernel configuration file, 108
`df` command, 232, 250
DHCP (Dynamic Host Configuration Protocol), 41, 144, 149, 512–516
for diskless farms, 590
global settings for clients, 514–515
how it works, 513–514
for IPv6, 42
rogue servers, 513
server setup, 588–591
subnet settings, 515
dhcpd, 513
configuring, 514–515
diagnostic messages, in boot process, 60
dial-up terminal, 584
diff mode, 89
digital certificates, 478. *See also* certificates
digital signatures, 476
directories
adding to shared library, 407–408
backups, 90
exporting multiple, in NFS, 306
hierarchies, 325
for unprivileged users, 452
dirty disks, forcing read-write mounts, 248
dirty filesystems, 244–245

disable soft updates flag, 65
disaffected users, 169
disconnecting serial consoles, 80
disk controllers, SATA, 24
disk ID labels, 212–213
disk images, mounting, 292–293
disk partitioning schemes
    assigning, 217–218
    removing, 217
disk space, for jails, 382
disklabel, 224
    and MBR alignment, 225
diskless farms, DHCP for, 590
diskless FreeBSD, 587–594
    clients, 588
    DHCP server setup, 588–591
    farm configuration, 592–594
    finalizing setup, 594–595
    security for, 591
    userland, NFS server and, 591–592
diskless remounting, 593–594
disks, 20–25
    bottleneck analysis with vmstat(8), 530
    installing files on new, 253–254
    labeling, 211–214
    lies, 201–202
    partitioning, 20, 23, 34–41, 252–253
    as performance bottleneck, 532
    schemes for, 217–218
    in single-user mode, 52–53
    viewing, 55–56
    ZFS and block size, 267–268
distfile, 372

distfiles directory, in Ports Collection, 366
distinfo file, 370
DNS (Domain Name Service), 150–154
    configuring, 42–43
    /etc/hosts, local names with, 151–152
    host/IP information sources, 151
    nameservice configuration, 152–153
documentation, 1–14, 33
domain, accessing in CIFS, 314–315
`domain` keyword, 152
download timing, 344
Dragonfly BSD, xxxix–xl
Dragonfly Mail Agent (DMA), 499, 500–503
    forwarding mail beteen users, 503–504
drives. *See also* disks
    reattaching and detaching, 276
    replacing, 276
DTrace, xli
du(1) program, 251
dual console, 78
dual-stack setup, 130
DuckDuckGo, 11
dump partition, 37
dump(8) command, 87
    backup level for, 210
    and snapshots, 244
dumps. *See* crash dump
DVD images, 27
Dvorak keyboard layout, 69
dynamic device management, with devd(8), 299–300

E

ECC RAM, 21
ECDSA key, 493
ED25519 key, 493
`$EDITOR` environment variable, 175
EFI (Extensible Firmware Interface), 20
ejecting removable media, 285
ELF binaries, 408
email, 499–504
    attachments, 13
    etiquette for requesting help, 12–13
    mailwrapper(8), 499–500
email signatures, 13
`emerg` log message, 547
emergency disk space, 252
empty filesystem file, creating, 293–294
emulation, ABI reimplementation vs., 414
encapsulation, 127–128
Encrypt Disks option, 39
Encrypt Swap option, 39
encryption, 595–598
    evaluating need for, 596
    of filesystems, 22–23
    of partitions, 65
    public-key, 475–482
enterprise network, 580
environment variables, 356
    cron and, 521
EoL (End of Life), of release, 26
epochal seconds, and real dates, 487–488
equal signs (`=` or `==`), 349
`erase` command, 87
`err` log message, 547
error messages, xlvii

Escape to loader prompt option, 52
/etc/adduser.conf file, configuring, 172–173
/etc/amd.map file, 318
/etc/auto_master file, 318
/etc/blacklistd.conf file, 319, 471
/etc/bluetooth file, 319
/etc/casper directory, 319
/etc/cron.d, 319
/etc/crontab file, 319, 520
    format, 521–523
*/etc/csh.**, 319
/etc/ddb.conf file, 319
/etc/defaults/devfs.rules file, 320
/etc/defaults/periodic.conf, 327–328
/etc/defaults/rc.conf file, 62
/etc/devd.conf file, 320
/etc/devfs.conf file, 296–297, 320
/etc/devfs.rules file, 320
/etc/dhclient.conf file, 320
/etc directory, 317–333
    across Unix versions, 317–333
/etc/disktab file, 320
/etc/dma/dma.conf file, 501, 502
/etc/exports file, 304, 307
/etc/freebsd-update.conf file, 429–430, 579
/etc/fstab file, 209–210, 416
    configuring, 253
    and file-backed filesystems, 294
    for jail, 573
    and memory disks, 292
    mounting partitions listed, 234
    and removable media, 285–286

*/etc/ftp.** file, 321
/etc/group file, 180–181, 441–442
/etc/hostid file, 321
/etc/hosts file, 151
    local names with, 151–152
/etc/hosts.allow file, 456–462
    example, 462
/etc/hosts.equiv file, 321
/etc/hosts.lpd file, 322
*/etc inetd.con*f file, 509–510
/etc/jail.conf file, 568, 569–573, 574
/etc/localtime file, 322
/etc/locate.rc file, 323
*/etc/login.**, 323
/etc/login.access file, 185
/etc/login.conf file, 188, 189
    environment settings, 190–191
/etc/mail/aliases file, 503–504
/etc/mail/mailer.conf file, 500
/etc/make.conf file, 324–325, 375, 439, 448–449
    for poudriere, 387–388
    and single ports, 376
    `WRKDIRPREFIX` option, 380
/etc/master.passwd file, 173–174
    editing, 176–178
/etc/motd file, 325
/etc/mtree directory, 325
/etc/netstart shell script, 54, 326
/etc/network.subr shell script, 326
/etc/newsyslog.conf file, 553
    sample entry, 557
/etc/newsyslog.conf.d/ directory, 553

/etc/nscd.conf file, 326
/etc/nsmb.conf file, 311
    keywords, 311–313
    options, 314–315
/etc/nsswitch.conf file, 151, 507
/etc/ntpd.conf file, 505–506
*/etc/opie**, 326–327
*/etc/pam.d/**, 327
/etc/passwd file, 173–174
/etc/pccard_ether script, 327
/etc/periodic.conf file, 327–328, 355
/etc/pf.conf file, 328, 465–467
/etc/pf.os file, 328
/etc/phones file, 328
/etc/pkg file, 356
/etc/pkg/FreeBSD.conf file, 356–357
/etc/printcap file, 329, 517–518
/etc/profile file, 329
/etc/protocols file, 126, 329
/etc/pwd.db file, 173–174, 329
*/etc/rc**, 329–330
/etc/rc script, 62–63, 71–74, 380, 539
    debugging custom, 405
    ordering, 402–403
    providers, 404–405
    REQUIRE statement in, 405
    and securelevel, 197
    service(8) command, 71–73
/etc/rc.conf file, 62, 145–146, 568
    changing from command line, 63–64
    `cloned_interfaces`, 164
    to enable sshd at boot, 492
    frozen with `schg`, 198

`ifconfig` statements in, 148
for jail, 574
/etc/rc.conf.d/ file, 64–71
console options, 69–70
to enable blacklistd, 471
filesystem options, 65–66
`kern_securelevel_enable`, 195
network daemons, 66–67
network options, 67–68
network routing options, 68–69
startup options, 64–65
/etc/rc.d/sendmail script, 503
/etc/rc.subr file, 404
/etc/regdomain.xml file, 330
/etc/remote file, 79–80, 330
/etc/resolv.conf file, 152, 153
/etc/rpc, 330
/etc/sc.d/localpkg, 405
/etc/security/ directory, 330
/etc/services file, 138–139
/etc/shells file, 179
/etc/skel file, 331
/etc/snmpd.config file, 560
/etc/spwd.db file, 173–174
/etc/src.conf file, 331, 439, 448–449
SVN-UPDATE, 437
/etc/ssh, 331
/etc/ssh/sshd.config file, 473, 494
/etc/ssl/ directory, 331
/etc/ssl/openssl.cnf file, 477
/etc/sysctl.conf file, 160
/etc/syslog.conf file, 548
comparison operators, 549

space or tabs, 551
/etc/syslog.d/ directory, 548
/etc/termcap, 332
/etc/termcap.small, 332
/etc/ttys file, 332, 584
console entry, 585
format, 584–585
/etc/unbound/, 332
/etc/wall_cmos_clock file, 332
/etc/zfs/ directory, 333
Ethernet, 125, 140–142
speed, 141
evaluations, in queries, 348–349
`exec.clean` option, for jail(8), 570
`exec.stop` command, 571
exports, mounting, 309
EXT filesystem, 283
Extensible Firmware Interface (EFI), 20
`extracommands` command, 73
extract mode, for tar, 90
extracted files, permissions for, 91
ezjail, 581

F

failover, 163
fallback brand, sysctls to set, 418
FAQ (Frequently Asked Questions), 7–8, 9
Fast EtherChannel (FEC), 163
Fast File System (FFS), xxxiv, 232–233
for kernel, 114
FAT (MS-DOS), 283

FAT32, formatting media, 286
`FAULTED` pool state, 275
faults, bottleneck analysis with vmstat(8), 531
`FCODES` variable, 323
fdesc(5), 301
fdisk(8), 215
`FETCH_RETRY` option, 344
`FETCH_TIMEOUT` option, 344
FFS. *See* Fast File System (FFS)
file-backed filesystems, and */etc/fstab*, 294
file descriptor filesystem, 301
file flags, 192–194
    limitations, 197
    setting and viewing, 194
files, xlviii
    aliases for, disabling, 237
    autoupdate unchanged, 447
    backups to, 90
    checking for obsolete, 444–445
    copying over SSH, 498–499
    corrupted, 354, 372
    customization, 16
    default, 16–17
    filesystems in, 293–294
    installing on new disks, 253–254
    moving to new dataset, 262–263
    ownership, 183–184
        in CIFS, 315
    tftpd and, 519
filesystem table, 209–210
filesystems, xliii, 20–25. *See also* foreign filesystems
    coherence, 88
    dirty, 244–245

on encrypted devices, 597–598
encryption, 22–23
file-backed, and */etc/fstab*, 294
in files, 293–294
jailed access to part, 564
lesser-known, 300–301
memory, 288–292
mount(8) to view mounted, 210–211
mounting and unmounting, 233–237
problems, 66
selecting, 34
size for, 243
user mounting of, 284
viewing current settings, 241–242
FILESYSTEMS provider, 404
`FILESYSTEMS` variable, 323
find(1) program, 244
finding
man pages, 5
packages, 340–342
snapshots, 244
firewall, 67, 465
blacklistd(8) and, 470
NFS and, 308
flags, for log rotation, 556
flash drives, foreign filesystems for, 284
flash (*.img*) format, 26
floppy disk drive, 102–103
fonts, on console, 69
Force 4K Sectors option, 39
foreign filesystems, 281–315
and permissions, 283–284
for removable media, 284–288

supported, 282–283
fortune(6), 511
forums, 2, 8
old information, 336
posting to, 14
searching, 11
fragments, 249
in FFS, 232
PF and, 466
frame, 127–128
free memory, 535–536
FreeBSD. *See also* upgrading FreeBSD
basics, xxxiv–xxxvii
birth of, xxxvi–xxxvii
development, xxxvii–xxxix
getting, 25–26
problem-solving resources, 9–11
resources for troubleshooting, 601–602
security announcements, 170–171
shrinking, 448–449
and SNMP, 557–562
strengths, xlii–xliii
support model, 426
testing, 426–427
versions, 26, 422–427
FreeBSD-current, 422–423
FreeBSD-stable, 423–425
snapshots, 425
which to use, 427
who should use, xliii–xliv
FreeBSD attitude, 2
freebsd-boot partition, 36
FreeBSD community, 613–614
reasons for volunteers, 615

ways of supporting, 615–616
FreeBSD Foundation, 9
FreeBSD fringe, 583–598
cloudy FreeBSD, 586–587
diskless farm configuration, 592–594
diskless FreeBSD, 587–594
storage encryption, 595–598
terminals, 584–586
FreeBSD Journal, 9
FreeBSD mirrors, 26, 27
FreeBSD Porter's Handbook, 393
FreeBSD Project
leadership, 614
submitting improvements to, 600
FreeBSD-specific time, 555
`freebsd-update cron` command, 434
`freebsd-update install` command, 433–434
`freebsd-update rollback` command, 434
`freebsd-update upgrade` command, 431
freebsd-update(8), 428, 579
running, 430–434
FreeBSD.conf file, 390
`FREEBSD_HOST` variable, 383
`FREEBSD_ID` option, for mergemaster, 447
FreeBSD.org website, 7–8
FreeNAS, xl
Frequently Asked Questions (FAQ), 7–8, 9
FreshPorts, 9, 342
fsck(8) program, 52, 246–248, 608
background, 247–248
`-y` flag, 247
fsdn(8) program, 247

fstyp(8) program, 284
ftpd(8) daemon, 321
    and user login, 179
full dump, 610
fully qualified domain name, 67

G

GBDE (GEOM-Based Disk Encryption), 22, 65, 595–596
GELI, 22, 65, 596, 597
`geli init`, 597
GENERIC file, 109, 438
GENERIC install, 105
`GENERIC` kernel, 420
    building, 439–440
GENERIC.hints file, 109
GEOM, 204–208
    autoconfiguration, 205
    control programs, 207–208
    device nodes and stacks, 208
    journaling, 238, 240–241
    labels, 214
    vs. volume managers, 206
    withering, 214
GEOM classes, 205
geom_eli.ko kernel module, 596
geom_journal kernel module, 240
getty(8) program, 539
GhostBSD, xl
GIDs file, in Ports Collection, 365
git(1), 16
`gjournal label` command, 240
gjournal provider, creating, 240

gjournal(8), 238, 240–241
`glabel create` command, 214
glob(3), 556
gmirror(8) class, 205
Google, 11
`gpart add` command, 229
`gpart bootcode` command, 222
`gpart create` command, 217, 227
`gpart delete` command, 226
`gpart destroy` command, 217, 252–253
`gpart modify` command, 221
`gpart resize`, 221
`gpart show` command, 215, 218, 220, 221, 226, 227–228, 229, 285
gpart(8) command, 214–217, 220
    for managing MBR slices, 225–226
GPT (GUID Partition Tables), 20, 209, 595–598
    creating partitions, 219–220
    default partitioning, 36
    device nodes, 218–219
    expanding disks, 223
    GUID labels, 213
    labels, 213–214
    partitions
        changing labels and types, 221
        resizing, 221
        types, 219
    scheme creation, 252
    and UEFI, 222–223
gptboot(8), 218, 222
gptzfsboot(8), 218
Greenwich Mean Time, 504
group ID (GID), 181, 183
groups of users, 180–185

administrative group, creating, 182–184
to avoid root, 182
creating, 181
default, 184–185
for logs, 553–554
membership changes, 181
system accounts, 182
growfs(8) command, 243
growisofs(1) command, 287–288
gstat(8), 532
GUID (globally unique identifier), for GPT partition, 211
GUID Partition Tables (GPT), 20, 23
partitioning scheme, 218–223
gvinum(8), 206
gzip compression, 91

H

hacking, 170, 490
halt(8) command, 74
Handbook, 7–8, 9
hard disks, 208–209
multiple, 24
hardening options for system, 44–45
hardware
cryptographic support, 482
customized with FreeBSD, 583
device names for, 62
as files, xlviii
for FreeBSD, 17–20
hot-swappable, 299–300
optimizing network, 159
proprietary, 19

hardware clock, 60
hardware MIBs, 100
hardware threading, 400
help, 600
    asking for, 11–14
        composing message, 12–13
        responding to email, 14
    mailing lists and forums for, 2
    man pages, 3–6
        finding, 5
        navigating, 5
        sections, 4
    minimizing requests, 2
    providing, 616–617
    resources, 1–14
hexadecimal numbers, 129–130
home directory, 46
    for user, 172
$HOME environment variable, 570
$HOME/.nsmbrc file, 311
/home partition, 23
host addresses, for login restriction, 187
"host key has changed" message, 595
host, logging to, 550–551
`host.allow` option, 192
`host.deny` option, 192
`Hostess` module, 562
hostname, 67
    false, for Dragonfly, 501
    installer request for, 31–32
    for login restriction, 186
hot-swappable hardware, 299–300
HTML, 12–13

`HTTP_PROXY` environment variable, 356
hubs, for Ethernet, 140
human errors, recovery from, 55
`hushlogin` environment variable, 191
HyperThreading, 400
hypervisors, FreeBSD on, 20

I

i386 platform, 17
ICMP (Internet Control Message Protocol), 126, 136–137
    PF and, 468
    redirects, 67–68
`ident` entry, in kernel configuration file, 108
ifconfig(8) command, 68, 144–145
    to create VLAN interfaces, 164–165
    to enable polling, 162
    `name` keyword, 148–149
`ignorelogin` environment variable, 191
illumos, xli
Image Writer for Windows, 27
inactive memory, 535
INCLUDE_CONFIG_FILE option, 117–118
INDEX file, in Ports Collection, 365
INET networking option, for kernel, 114
`inet6` keyword, 145–146
inetd(8) daemon, 66, 508–512
    changing behavior, 512
    jail for, 567
    sample configuration, 511
    servers configuration, 510–511
    starting, 511–512
    wrappers and, 456

infinite loop, memory allocation with, 25
`info` log message, 548
inheritance
    of dataset properties, 261
    repositories, 357–358
init(8), 539
inodes (index nodes), 232
    fsck(8) program to verify, 246
input in Unix, xlvii–xlviii
input/output
    and performance, 526
    top(1) tool and, 538
`install clean` command, 380
installation images, 26–27
installing
    files on new disks, 253–254
    jail packages, 578
    kernel, 439
    Linux packages, 419
    packages on diskless client, 594–595
    pkg(8), 338–339
    poudriere, 383
    poudriere ports tree, 386
    software, 342–344
        from Ports Collection, 370–381
installing FreeBSD, 29–47
    core settings, 30–32
    disk partitioning, 34–41
    distribution selection, 32–33
        UFS installs, 34–38
        ZFS installs, 39–41
    finishing, 46–47
    network and service configuration, 41–46

planning, 15–28
configuration with UCL, 17
default files, 16–17
disks and filesystems, 20–25
getting FreeBSD, 25–26
hardware, 17–20
network installs, 27–28
integers, 99
integrity check
of cryptosystems, 475
for packages, 354–355
resiliency and, 237
in ZFS, 265
for zpool, 273–276
Intelligent Platform Management Interface (IPMI), 76–77
interface
multiple IP addresses on single, 147–148
renaming, 148–149
testing, 146
internet, accessibility of old data, 14
Internet Control Message Protocol (ICMP), 126
Internet Protocol (IP), 125
interruptible NFS mount, 309
intruders
mtree for preparing for, 485–489
network targets, 198–199
ioapic device, 61
iocage, 581
iostat(8), 528
IP addresses
attaching syslogd(8) to single, 552
for BMC, 76
for interface, 68

for jails, 566, 568
list for wrappers, 457–458
multiple on interface, 147–148
setting, 146–147
sshd listening to, 494
unusable, 133
IP Filter, 463
IP (Internet Protocol), 125
adding to interface, 145–146
IPFW, 463
IPMI (Intelligent Platform Management Interface), 76–77
SOL (Serial-over-LAN) connections, 80–81
IPMItool, 80–81
IPSEC networking option, for kernel, 114
IPv4, 41
IPv4 addresses, 131–133
IPv4-only stack, 130
IPv6 addresses, 133–136
aliases, 147–148
assigning, 136
IPv6 network, 42
exporting to, 307
IPv6-only stack, 130
ISC DHCP server, 513
ISO 8601 time format, for logs, 555
ISO 9660 filesystem, 283
burning to optical media, 287
creating, 286–287
IVCSW (involuntary context switches), 538

J

jail ID, 575

jails, 381, 563–581
for ancient FreeBSD, 580–581
basics, 564–565
at boot, 568
clean login for, 577
creating, 383–386
customizing, 579–580
defaults, 571–572
defining, 570
dependencies, 575
disk space, 382
host server setup, 565–568
networking, 565–568
in-jail startup, 571
installing packages, 578
notes on, 581
parameters as variables, 572–573
processes in, 575–576
rules, 298–299
running commands in, 576–578
setup, 568–575
userland, 569
startup and shutdown, 574
testing and configuring, 573–574
updating, 578–579
viewing, 386, 575
jexec(8) command, 576–578
jls(8), 575
job control, xxxiv
job scheduler, 520–524
cron(8), 520–523
periodic(8), 323, 327, 523–524
journaling
GEOM, 238, 240–241

and recovery, 246
and soft updates, 238, 242
Joy, Bill, xxxv
JSON, 17

K

`KeepModifiedMetadata`, for freebsd update, 429
kenv(8), 101
Kerberos authentication, 10, 301
`KERNCONF` variable, 110, 439
kernel, 95–121
assumptions, 396–397
basics, 96–97
booting alternate, 111–112
building, 105–112
buses and attachments, 106–107
preparation, 106
troubleshooting, 118–119
working kernel backup, 107
building, installing, and testing, 439–440
configuration file format, 107–109
configuration, `no` option and `include`, 119
custom configuration, 112–119
trimming, 112–118
enabling crash dumps, 44
environment, 101–103
inclusions, exclusions and expansion, 119–121
and jails, 564
locks, 399
network capacity in, 157–158
options, 58
kernel-dbg, 33

kernel debugger configuration utility, 319
kernel memory, minidump of, 25
kernel modules, 19, 103–105
    loading and unloading, 104
    loading in boot process, 105
    skipping, 121
    viewing loaded, 103–104
`kern.elf32.fallback_brand` sysctl, 418
`kern.elf32.nxstack` sysctl, 485
`kern.elf64.fallback_brand` sysctl, 418
`kern.elf64.nxstack` sysctl, 485
`kern.hostname` sysctl, 97
`kern.ipc.nmbclusters` systcl, 160
`kern.ipc.somaxconn` systcl, 161
`kern.maxusers` systcl, 160
key fingerprint, 493, 497
keyboard, console options for, 69–70
keymap, selecting, 31
`KeyPrint` option, for freebsd update, 429
keystrokes, script to copy, 92
keyword searches, on man pages, 5
Keywords directory, in Ports Collection, 365
kldload(8), 104
kldstat(8) command, 103–104
kldunload(8), 104
knobs, 62. *See also* tunables
`KNOWN` keyword, for TCP wrapper, 458
kqueue(2), 413
krb5, 10

L

labels. *See also* BSD labels
    changing for GPT partition, 221
    for disks, 211–214
    for partition, 37
    UFS, 239, 243
    viewing, 212
LACP (Link Aggregation Control Protocol), 163
lagg(4), 163
    configuring, 164
laptop theft, 596
ldconfig(8), 406–407
    and weird libraries, 408–409
`ldconfig_local_dirs` variable, 407
`ldconfig_paths` variable, 407
`LD_LIBRARY_PATH` environment variable, 71, 409–410
`LD_PRELOAD` environment variable, 409–410
legacy boot, 20
legacy hardware, boot process on, 222
legacy mode, 50
LEGAL file, in Ports Collection, 365
legal restrictions
    on Ports Collection, 369–370
    on software, 337
Let's Encrypt, 481
Level 2 Adaptive Replacement Cache (L2ARC), 267
libarchive(3) command, 88, 92
libiconv.ko module, 311
libmap file, 410, 456
libmchain.ko module, 311
libraries. *See* shared libraries
LibXo, 586–587
license, 337

line-continuation character (\), 306–307, 459
link aggregation, 163
linprocfs(5), 301, 416
Linux, xli
    commercial software, 420
Linux mode, 415, 418–420
    debugging, 418–420
    testing, 417
Linux packages, installing, 419
Linux process filesystem, 301
Linuxator
    installing and configuring, 415
    userland, 416
Linuxisms, 413
`ListenAddress`, 494
listeners on ports, 156–157
live system, activating, 93
`lm75` module, 562
load average, 534
load, once-in-a-lifetime vs. standard, 161
loader, 51–52
    booting from, 57
    configuration, 57–58
    variables, 56
loader configuration file, 16
loader prompt, 55–57
loader.conf file, 104
`loader_logo` option, 58
local blacklistd rules, 471
local build, installing jail from, 385–386
local configuration files, 16
`LOCAL`, for login restriction, 187

local-link addresses, 135–136
local mail delivery, disabling, 501
local partitions, mounting, 53
`LOCALBASE` variable, 381
`localpkg` script, 405
`local_unbound`, 44, 154
locate(1), 323
lock order reversal, 399
locking
    SMP and, 397–398
    user accounts, 178
log rotation, 553
log sockets, 552–553
logical block addressing (LBA), 202
logical port, 138
login(8), 539
login classes, 172, 188–192
    class definitions, 188
`LoginGraceTime`, 495
logins
    control, 191–192
    restricting, 185–188
    on serial console, 81
logs, 66
    backup process and, 90
    changes, 88
    of connection attempt, 460
    file management, 553–557
    overlap, 551
    for poudriere, 389
    rotation by size and time, 555
    from script(1) command, 92
    sending messages to programs, 550

specifying senders, 552
with syslogd, 546–553
message levels, 547
in verbose mode, 553
loopback device, 117
lost+found directory, 246
lp (default printer), 517
lpd(8) printing daemon, 70, 516, 517–518
LPD (Line Printer Spooler Daemon), 516
`ls` command, for viewing flags, 194
ls(1) program, 183–184
`lsdev`, 55–56
`lsof` package, 490
lz4 compression algorithm, 273

M

MAC addresses, 141–142
for clients, 514
for DHCP client, 589
MAC table, 141–142
macOS, xl
macros, configuring for PF, 466
`MAIL` environment variable, 190
mail server, 499
mail, status mail, 545–546
mailing lists, 2
aliased, 504
archives, 8
searching, 11
for FreeBSD-stable, 424
general questions, 13
old information in archives, 336

MAILNAME, for dma(8), 501
mailq(1) program, 500
`MailTo root` option, for freebsd update, 429
mailwrapper(8), 499–500
maintenance jobs, 327, 545–546
major release, 422
`make build` command, 373
`make buildkernel` command, failure, 118
`make buildworld` command, 438
`make check-old` command, 444
`make check-old-libs` command, 445
`make checksum` command, 372
`make clean` command, 380
`make config` command, 371–372
`make config-recursive` command, 377
`make configure` command, 373, 374
`make deinstall` command, 379
`make delete-old-libs` command, 445
`make depends` command, 373
`make extract` command, 373
`make fetch` command, 372
`make install` command, 373
`make installkernel` command, 111, 439
`make installworld` command, 443–446
`make missing` command, 378
`make package` command, 379
`make patch` command, 373
`make pretty-print-config` command, 374–375
`make readmes` command, 369
`make rmconfig-recursive` command, 377
`make showconfig` command, 375
make(1) program, 362

SMP and, 400
`make_buildkernel` command, 110
`make_distribution` command, 569
Makefile, 362, 365, 370
makefs(8) program, 286–287
`makeoptions` entry, in kernel configuration file, 108
malloc-backed memory disks, 290
man pages, 3–6, 600
contents, 6
for cryptography, 9–10
finding, 5
navigating, 5
sections, 4, 6
`manpath` environment variable, 191
MASQUERADE, for dma(8), 501
`MaxAuthTrie`, 495
maximum resource limits, 190
MBR (master boot record), 20, 208–209, 218, 222
device nodes, 224–225
and disklabel alignment, 225
partitioning, 23
partitioning scheme, 223–226
mbrowse, 559
mbufs, 157, 159–160
mdconfig(8), 291, 292
mdmfs(8), 290–291, 292–293
memory, 61
allocation in infinite loop, 25
bottleneck analysis with vmstat(8), 529
and network optimization, 159–161
and performance, 526
for */tmp*, 65
usage, 535–536, 542

memory disks, 117, 289, 290–292
    creating and mounting, 290–291
    and */etc/fstab*, 292
    shutdown, 291
memory filesystems, 288–292
`MergeChanges` option, for freebsd update, 429
merged from current (MFC), 424
mergemaster(8), 440–443, 446
    customizing, 446–447
message of the day (motd) file, 325
metadata, 232
MIBs (management information base), 98–99
    SNMP, 558–559
Microsoft Outlook, email from, 13
minidump, 610
    of kernel memory, 25
MINIMAL file, 110
MINIMAL kernel, 111
minor release, 422
Mirror Swap option, 39
mirror VDEVs (virtual devices), 266, 274
mirrors, 26, 27
`mixpasswordcase` option, 192
Mk subdirectory, in Ports Collection, 366
mkisofs(1), 287
mksnap_ffs(8) program, 244
modular kernel, 96
`MODULES_OVERRIDE` option, 121
monitor
    console options for, 69
    display on, 70
monitoring system security, 489–490

mount point, 209
    temporary, for new partition, 253
mount(8) program, 233–237
    for foreign filesystems, 282–284
    options, 210
mountd(8) daemon, 303
mounting
    disk images, 292–293
    exports, 309
    filesystems, 233–237
    local partitions, 53
    share in CIFS, 313–314
    thumb drive, 285
mounts
    showing available for NFS client, 309
    stackable, 254–255
mount_smbfs(8), 313–314
mouse, 69–70
MOVED file, in Ports Collection, 365
moving
    files to new dataset, 262–263
    package cache, 345
`msdosfs` mount type, 283
mt(1) command, 87
mtree(1), 485–489
    exclusion file, 488
    spec file output, 487–488
        checking for differences, 488–489
        saving, 488
multitasking, preemptive, 397
multiuser startup, 63–71

N

name service, 150. *See also* DNS (Domain Name Service)
    switching, 507–508
named(8) program, 553
names
    for boot environment, 277
    for interfaces, changing, 148–149
nameserver, caching, 153–154
nameserver list, 153
navigating man pages, 5
`nbns` keyword, for CIFS configuration, 312
NDP (Neighbor Discovery Protocol), 142
net-snmp, 559
NetBSD, xxxvi, xxxix
Netflix, xxxvii
`Netgraph` module, 562
`net.inet.ip.portrange.reservedhigh`, 139
`net.inet.ip.portrange.reservedlow`, 139
`net.inet.tcp.cc.available` sysctl, 528
netmasks, 131–133
    computing in decimal, 132
`netstat`, 489, 527
netstat(8) program, 154–155
    to calculate mbuf clusters, 160
    per-protocol performance statistics, 158
    viewing open network connections, 156–157
network, 123–142
    activity, 154–158
    bandwidth, and performance, 526
    bits and hexes, 128
    capacity in kernel, 157–158
    configuration, 142–165

prerequisites, 144–149
installing jail from, 384
interface selection, 41
layers, 124–126
optimizing performance, 158–162
maximum incoming connections, 161
memory usage, 159–161
polling, 161–162
performance monitoring, 527–528
in single-user mode, 54
time, 504–507
traffic control, 454
network adapter
aliases for, 68
teaming, 162–164
Network Address Translation (NAT), PF and, 467
Network Configuration screen, 41–42
network daemons, 66
Network File System (NFS). *See* NFS (Network File System)
network installs, 27–28
network layer (of OSI), 125, 127, 128
network number, 133
network-related options, for kernel, 114
network secure mode, 196
network stacks, 130
Network time protocol (NTP), 505, 567–568
NETWORKING provider, 404
newfs(8) command, 253, 294
newfs_msdos(8) program, 286
newsyslog(8), 553
NFS (Network File System), 301–310
enabling client, 308–310
exporting multiple directories, 306

exports configuration, 304–308
and firewalls, 308
interoperability, 302
and jails, 567
kernel options supporting, 115
mount options, 309–310
server
configuration, 302–303
and diskless client userland, 591–592
and upgrades, 448
and users, 305–306
versions, 302
zfs(8) for managing, 308
nfsd(8), 303
niceness, 543–545
Nintendo GameCube, emulator, 413
nmbclusters, 160
noasync mounts, 236
nobody account, 453
noexec mount option, 236
`nologin` environment variable, 191
`nomatch` rules, for devd(8), 300
nonautomatic packages, 346
changes, 352
nonexecutive stack, 484–485
nonrepudiation, of cryptosystems, 475
normal, defining, 527
`nosymfollow` option, 237
“not a working copy” error, 436
NOTES file, 110, 119, 438
`notice` log message, 548
`notify` rules, for devd(8), 300
nscd(8) service, 326

NTP (Network time protocol), 505, 567–568
ntpd(8) program, 44, 504, 553
    configuring, 505–506
null memory disk, 290
null modem cable, 75
NULLCLIENT option, for Dragonfly, 502

O

obsolete files, checking for, 444–445
`offline` command, 87
`OFFLINE` pool state, 275
`ONLINE` pool state, 275
opaque sysctls, 97
opaques, 99
open files, listing all, 490
Open System Interconnection (OSI) network protocol stack, 124
    applications, 126
    datalink layer, 125, 127–128, 138
    network layer, 125, 127, 128
    physical layer, 124, 128
    transport layer, 126, 127, 128
OpenBSD, xxxix
`openntpd` package, 568
OpenSolaris, xli
OpenSSL, 476, 477
    clients, 497
    passwords and keys, 499
`openssl s_client` command, 481
operating systems
    multiple, 24
    packages and upgrades, 449–450
    panic, 606–612

running software from wrong, 412–418
OPIE (One-time Passwords In Everything), 326–327
optical disk (*.iso*) format, 26
optical media
burning ISOs to, 287
burning UDF to, 287–288
creating, 286–287
/etc/fstab entry for, 285
foreign filesystems for, 284
`options` entry, in kernel configuration file, 108
`OPTIONS_SET` variable, 376
`OPTIONS_UNSET` variable, 376
Oracle Solaris, xl
organization employees, security risks from, 169
output. *See also* input/output
in Unix, xlvii–xlviii
ownership
of device node, changing, 296
of files, 183–184
in CIFS, 315
log files, 553–554
in TFTP, 519

P

package cache, 345
package database
changing, 351–352
querying, 346–347
packaged dependencies, 378
packages, 336–356
branches, 358–359
building, 379

fetching, 344
files in, 337, 353
finding, 340–342
information and automatic installs, 346
installing on diskless client, 594–595
integrity, 354–355
for jails, installing, 578
locking, 352–353
maintenance, 355
networking and environment, 355–356
Ports Collection and, 363
repositories, 356–358, 389
building, 371
customization, 357
private, 381–391
remote custom, 390–391
security, 490
and system upgrades, 449–450
uninstalling, 350–351
upgrading, 359–360
packaging system, xliii
packet filtering, 454, 462–470
default accept vs. default deny, 463–464
and stateful inspection, 464–465
packet sniffers, 492
packets, 127
normalization in PF, 466
pagedaemon, 529
pages of memory, 529
paging, 530, 540–541
PAM (Pluggable Authentication Modules), 327
panic, 599, 606–612
recognizing, 606–607
responding to, 607–612

serial consoles and, 609
parallel builds, limiting, 391
`PARANOID` keyword, for TCP wrapper, 458
parent datasets, unmounted, 262
partition table, 38
partitioning schemes, 23, 35, 208–209
MBR (master boot record), 223–226
partitions, 20, 208–209
adding new, 37
alignment, 220
BSD label, creating, 227–228
for disk, 23, 34–41, 252–253
encrypted, 65
mount point for, 209
removing, 216–217
removing space, 250–251
UFS for, 23–24
viewing, 215–216
passphrase, 597
for certificate, 480
for full-disk encryption, 39
`passwd`, 54, 174
`passwd_format` option, 191
password
changing, 174
in CIFS, 314
control, 191–192
default for new users, 173
for dma(8), 502
expiration, 176
for group, 181
for OpenSSL, 499
root, 41, 46, 179–180

for single-user mode, 585
for user, 172
`password` keyword, for CIFS configuration, 312
patches
levels, 422
updating to latest, 430–434
`PATH` environment variable, 190, 191
path, for log file, 553
patterns, for queries, 347
pax, 87
performance
computer resources and, 526–527
monitoring, 526–562
bottleneck analysis with vmstat(8), 528–532
disk I/O, 532
network, 527–528
with top(1), 533–538
per-protocol statistics, from netstat, 158
tuning, 541–545
periodic(8), 323, 327, 523–524
Perl modules, 341–342
permissions
of device node, 296
for extracted files, 91
and foreign filesystems, 283–284
for logs, 554
`PermitRootLogin`, 495
`PF` module, 562
PF (packet filter), 463
and blacklistd(8), 471
configuring, 465–467
managing, 468–470
small server example, 467–469

pfctl(8), 463, 468–470, 475
    for active anchor, 471
pf.ko module, 463
pfSense project, xl
PGID (process group ID), 539
pgrep(1), 576
physical address, 141
physical layer (of OSI), 124, 128
physical serial lines, 79
pidfile, 556–557
ping, 146
pipes, xlvii–xlviii
`pkg autoremove` command, 351
`pkg check` command, 355
pkg-check(8) tool, 354
`pkg clean` command, 345
pkg-create(8), 345
`pkg delete` command, 350
pkg-descr file, 370
`pkg help` command, 338
pkg-help file, 370
`pkg info` command, 346, 353
`pkg install` command, 342–343, 354, 390
pkg-plist file, 370
`pkg query` command, 347
`pkg remove` command, 379
`pkg unlock` command, 353
`pkg update` command, 358
`pkg upgrade` command, 359
`pkg which` command, 353
pkg(8), 336, 337–338
    command aliases, 349

common options, 339
configuring, 339–340
`fetch`, 344
installing, 338–339
and jails, 578
repository download, 390
pkg.conf file, 390
customizing download behavior, 344
`DEBUG_LEVEL` in, 340
`PKG_CACHDIR`, 345
`PKG_ENV` section, 356
pkgNG, 336
pkg_query(8), 387
`PKG_REPO_SIGNING_KEY` variable, 393
pkg_static(8), 449
pkg_tools, 336
`pkg_upgrade` command, 449
plaintext email, 12–13
PMBR (protective master boot record), 218, 222
polling
mode, 527
on network, 161–162
pool. *See* zpool (storage pool)
portability of FreeBSD, xlii
porting, 412
portmaster, 371
ports, 138–139
listeners on, 156–157
open, 155–156
review of open, 198
updating installed, 450
Ports Collection, xliii, 336, 358, 361–393
cleaning up ports, 380

contents, 365–367
customization options, 373–381
for command line, 374–375
install path, 380–381
installing software from, 370–381
and Linux mode, 418
port flavors, 378
problem ports, 389
tracking build status, 379–380
uninstalling and reinstalling ports, 379
ports index file, 367–370
ports tree, 363. *See also* Ports Collection
read-only, 380
`portsnap cron update` command, 364
portsnap(8), 364, 393
portupgrade, 371
POSIX standard, 412–413
posting to forums, 14
`poudriere bulk` command, 388
`poudriere options` command, 388
poudriere package-building system, 361, 371, 381–391
configuring ports, 386–388
installing and configuring, 383
installing ports tree, 386
large and small systems, 391–392
make.conf for, 387–388
package list for, 387
repository, 389
resources, 382
running, 388–389
updating, 392–393
`poudriere ports` command, 386
power-on, 50–51

PowerPC, 18
preemptive multitasking, 397
preening, 246
prefix length, 131–133. *See also* netmasks
`PREFIX` variable, 381
primary partitions, 223
Primordial Unix Compression, 91
print servers, 516–518
printer, configuration information, 329
printing, 516–518
    /etc/princap file, 517–518
`priority` environment variable, 191
private key, 476
private repository, 389–391
process ID (PID), 533
    for jails, 575
process state, 539
processes
    bottleneck analysis with vmstat(8), 529
    following, 539–540
    in jails, 575–576
    priority in top(1) tool, 544
    rescheduling to balance, 543
    vs. thread, 401
processors
    multiple, 396
    and SMP, 399–401
    virtual, 400
procfs(5) program, 281, 300
procfs (process filesystem), 416
production releases of FreeBSD, 26
programs, logging by name, 550
proprietary hardware, 19

protective master boot record (PMBR), 218
provider, for geoms, 206
proxy server, need for, 356
PRUNEPATHS variable, 323
ps(1) command, 534, 576
pseudodevices, in kernel, 116–117
pseudorandom numbers, 117
pseudoterminal, 584
public-key encryption, 475–482
    certificates, 478–481
public-key files, 493
PuTTY, 497
pw(8) command, 178
pwd_mkdb(8), 174, 176
PXE (Preboot Execution Environment), 588

Q

qemu-user-static package, 382
quarterly branches, in package system, 358
queries
    evaluations in, 348–349
    remote, 347
QWERTY keyboard, 69

R

RAID controllers, 18, 204
    ZFS and, 22
RAID-Z, 265, 266
    and pools, 267
    virtual devices, 274
RAID-Z2, 266

RAID-Z3, 266
RAM, 61. *See also* memory
random password generator, 172
random quote generator, 510–511
`range` keyword, in dhcpd, 515
rcorder(8), 402–403
read-only mounts, 235
read-only sysctls, 100
read-write mounts, forcing on dirty disks, 248
README file, in Ports Collection, 366, 369
real memory, 61
reboot(8) command, 74
rebooting, to test interface changes, 149
recursion, front-loading, 377–378
recursive nameserver, 150
redundancy, 274
    in ZFS, 265–267
Reed, Darren, 463
regular expressions, in ruleset, 298
Release Engineering team, 424
releases of FreeBSD, 422
`reload` command, 73
remote blacklistd rules, 471, 472
remote computers, intruders and, 168
remote logins, SSH for, 66–67
remote modems, phone numbers for, 328
removable hardware, kernel support for, 117
removable media
    ejecting, 285
    and */etc/fstab*, 285–286
    foreign filesystems for, 284–288
`REMOVED` pool state, 275

renice(8), 544–545
repairs, 92–93
repositories
    inheritance, 357–358
    for packages, 356–358, 389
        customization, 357
        remote custom, 390–391
    private, 381–391
REQUIRE statement, in rc script, 405
`requirehome` environment variable, 191
reserved ports, 139
resilvering, 274
resolver, 150
resources
    and performance, 526–527
    user limits, 189–190
restarting, services, 72
`retension` command, 87
`rewind` command, 87
rewinding tapes, 84–85
rmuser(8) program, 178
rndc(8), 184
rollback of FreeBSD update, 434
root
    email sent to, 503
    in jail, 564
    password change by, 174
    server login as, 495
    user changes by, 176
    user groups to avoid, 182
root dataset, 259
root directory, for tftpd, 518
root filesystem

partition letter for, 228
as read-write, 53
root partition, for filesystem, 37–38
root password, 41, 46, 179–180
for jail, 574
requiring, 484
root user, and NFS server, 305
rootkit-hunting software, 490
rotating logs, by size and time, 555
`routers` option, in dhcpd, 515
rpcbind(8) daemon, 303
RPCs (remote procedure calls), 303, 330
RSA key, 493
rsync(8) program, 352
rtld(1), 406, 409
running processes, 534
runtime tunable sysctl, 101
`RUN_UPDATES` option, for mergemaster, 447

S

Safe Mode, 59
Samba, 315
`sappnd` flag, 193
SATA disk controllers, 24
savecore(8), 608–609
/sbin/nologin, 183
scheduling
to balance processes, 543
binary updates, 434
tasks, 520–524
`schg` flag, 193

scp(1), 498
script(1) command, 92
script kiddies, 168–169
scripts, startup and shutdown, 402–405
scrubbing, 466
SCSI drives, for tape backups, 84
SCSI_DELAY option, for kernel, 115
SCTP transport protocol, in kernel, 114
searching ports index file, 368–369
SEARCHPATHS variable, 323
sector size, 202
Secure Shell, 331
securelevels, 192, 195–198
    limitations, 197
security, 167–199
    attackers, 168–170
    and crash dump, 611–612
    data protecton, 428
    default accept vs. default deny, 454–455
    for diskless systems, 591
    file flags, 192–194
    FreeBSD announcements, 170–171
    global settings, 482–485
        install-time options, 483–484
        secure console, 484
    hacking, 490
    for inetd, 509
    LD_ environment variables and, 409
    login classes, 188–192
    monitoring system, 489–490
    network targets, 198–199
    for packages, 490
    preparing for intrusions with mtree(1), 485–489

removable media risks, 284
resource limits, 189–190
securelevels, 192, 195–198
in SNMP, 559–560
TFTP and, 518
user security, 171–178
for users, 185–192
workstation vs. server, 199
`security.bsd` sysctl tree, 485
`security.jail` sysctl tree, 565
self-signed certificates, 480–481
Sendmail, 499
shutting down, 503
sendmail(8) daemon, 70
`sendmail_outbound_enable`, 70
Separate Intent Log (SLOG), 267
serial consoles, 74–82
configuration, 77–79
disconnecting, 80
IPMI setup, 76–77
and panics, 609
physical setup, 75–76
using, 79–82
serial port protocol, 74–75
server security, vs. workstation, 199
`ServerName update.freebsd.org` option, for freebsd update, 429
SERVERS provider, 404
service(8) command, 71–73, 402
services
configuration, 41–46
for jails, 566
listing and identifying enabled, 71–72
managing, 72–73

restarting, 72
sysrc(8) to enable, 63–64
`set` command, 56
`setenv` environment variable, 191
setuid programs, disabling, 236
`severity` option, for log message, 460
sftp(1), 498
shared libraries, 71, 405–410
adding directories to search list, 407–408
attaching to programs, 406–409
obsolete, 445–446
program requirements, 409
remapping, 410–412
versions and files, 406
`sharenfs` property, for NFS exports, 308
shares, mounting in CIFS, 313–314
`shell` environment variable, 191
`$SHELL` environment variable, 570
shell scripts, variables, 461
shells, 178–179
selecting, 52
for user, 172
shorn write, 270
`show` command, 56
showmount(8) command, 309
shutdown, 73–74
memory disks, 291
and stopping jails, 568
syncer and, 245
shutdown scripts, 402–405
shutdown(8) command, 74
SID (session ID), 539
`SIGHUP`, logfile rotation on, 557

signal, for log rotation, 557
single-key ciphers, 475
single-user mode, 51, 52–54
    network in, 54
    programs available, 53–54
    reproducing panic in, 611–612
    upgrades and, 448
skipping modules, 121
SLAAC, 42
sleeping processes, 534
slice device node, 224–225
slicer, 207–208
slices, 223
    activating, 226
    creating, 225–226
    removing, 226
smart host, for Dragonfly, 501
SMB (Server Message Block), 310
smbfs.ko module, 311
smbutil(8) program, 310–315
    `view` command, 313
SMP (symmetric multiprocessing), 115, 396–401
    problems, 399
    and processors, 399–401
SMT (Simultaneous Multi-Threading), 400
snapshots, 271–273
    accessing, 272
    creating, 271–272
    destroying, 273
    disk usage, 244–245
    finding, 244
    of FreeBSD-current and -stable, 425
    taking and destroying, 244

UFS, 243–245
vs. journaling, 238
ZFS, 276
SNMP (Simple Network Management Protocol), 557–562
basics, 557–560
MIBs (management information base), 558–559
security, 559–560
sockstat(1), 155, 198, 489, 566
soft updates, 237
for background fsck, 247–248
and journaling, 238, 242
software
add-on, 336
attaching shared libraries to, 406–409
building, 362
commercial for Linux, 419
installing, 342–344
from Ports Collection, 370–381
jail for, 565
running from wrong architecture or release, 420
running from wrong OS, 412–418
ABI reimplementation, 413–414
binary branding, 414
emulation, 413
recompilation, 412–413
software binaries, branding, 417–418
SOL (Serial-over-LAN), 74
IPMI connections, 80–81
Solaris, xl, 21
solid-state disks, TRIM protocol, 242–243
sort(1) program, 251
source code, 336
building FreeBSD from, 437–448

for FreeBSD upgrade, 435
for kernel, 106
for port, 371
and software, 362–363
updating, 436
upgrading from, 428
Sparc hardware, 18
sparse files, 293–294
`spawn` option, for TCP connection, 461–462
special mounts, 234
SRV record, 357
`ssh-keygen` command, 493, 595
SSH (Secure Shell), 44, 478, 491–499
clients, 497–498
configuring daemon, 494–497
copying files over, 498–499
keys and fingerprints, 493
diskless clients and, 595
for remote logins, 66–67
server, 492–493
enabling blacklistd in, 473
user access, managing, 496
sshd(8), 492–493
jail for, 567
SSL (Secure Sockets Layer), 478
library, 376
stack guard page, 485
stack, nonexecutive, 484–485
stackable GEOM classes, 205
stackable mounts, 254–255
standard error, xlvii
standard input, xlvii
standard output, xlvii

STAPE environment variable, 85
startup scripts, 402–405, 446
startup services, 44
startup/shutdown scripts, from vendors, 405
stateful inspection, 464–465
stateful protocol, 138
stateless protocol, 137
`status` command, 73, 86
status mail, 545–546
storage
    adding to UFS, 252–255
    device control programs, 205
    device nodes, 202–203
    disks, lies, 201–202
    encryption, 595–598
    GEOM, 204–208
    hard disks, partitions and schemes, 208–209
    identifying devices, 204
    on jail host, 565
streaming protocol, 138
strings, 99
striped VDEVs (virtual devices), 265
su(1) (switch user) command, 179
subnets, 133–136
Subversion (SVN), 435
    updates, 436
Sun Microsystems, xxxv, xl
`sunlnk` flag, 193
superblock, 232
SVN (Subversion), 435
svn(1), 16
svnlite(1) command, 435
    for source code, 436

swap-backed memory disks, 290
swap partition, for crash dump, 608
swap space, 24–25, 37, 39, 536, 540–541, 542
    partition letter for, 228
switches
    for Ethernet, 140
    failure, 141
    quality, 159
symbol versioning, 406
symlinks, disabling, 237
symmetric multiprocessing (SMP), 396–401
SYN packet, 464
syncer, 245
synchronous mounts, 235
sysctl(8) program, 97–101
    MIBs (management information base), 98–99
    values and definitions, 99
sysctl.out file, 97
sysctls
    changing, 100–101
    runtime tunable, 101
    to set fallback brand, 418
    viewing, 100
syslog protocol, levels, 547
syslog server, 141
syslogd(8), 66, 460, 546–553
    customization, 552–553
    facilities as source of log entry, 546–547
    and jails, 566–567
    local facilities, 549
    logging by program name, 550
    processing messages with, 548–553
sysrc(8), 63–64

systat(1), 528
system accounts, 182
system administrator, xlv–xlviii
system backups, 84
system calls, 419
system clock, setting, 43
system shutdown, 73–74
system status, top(1) tool for overview, 533

T

tables, configuring for PF, 466
tape drives, density, 86
tapes
    for backups, 84–87
    rewinding, 84–85
tar(1) command, 87–92, 254
    modes, 88–90
    non-default storage, 90
    verbose flag for, 90
tarball, 88
targets, for *Makefile*, 362
Tarsnap, 87
tasks, scheduling, 520–524
tasting, 205
TCP/IP network protocol, 123, 124
    basics, 136–139
TCP port 22, 494
TCP port, for Dragonfly, 501
TCP (Transmission Control Protocol), 126, 137–138
    displaying retransmits, 157
    NFS over, 303
TCP wrappers, 454, 455–462

configuring, 456–462
client list for, 457–458
keywords, 458
TCP_HHOOK networking option, for kernel, 114
TCP_OFFLOAD networking option, for kernel, 114
telnet(1), 481, 492
Templates directory, in Ports Collection, 366
temporary mount point, for new partition, 253
`term` environment variable, 191
terminal emulators, 79
terminal server, 75
terminals, 332, 584–586
testing
crash dump, 609–610
FreeBSD, 426–427
interface, 146
changes, by rebooting, 149
jails, 573–574
kernel, 439
Linux mode, 417
text editor window, 175
textdump, 610–611
TFTP (Trivial File Transfer Protocol), 518–520
tftpd(8), 518
and boot loader, 590–591
configuring, 519
threading library, 402
threads, 398, 401–402
bottleneck analysis with vmstat(8), 529
three-way handshake, 138, 464
thumb drive
with partition table, mounting, 285
writing images to, 288

tiered hardware, 17, 18
"tilde-dot" disconnect sequence, 80
tilde (~), for user's home directory, 190
time
    epochal seconds and real dates, 487–488
    for logs, 554–555
    redistributing, 506–507
time servers, 505
time slice, 397
time zone, 43–44
    local data, 322
    setting, 504
time zone files, 430–431
timecounter, 60
timed rc script, 404
`times.allow` option, 192
`times.deny` option, 192
`timezone` environment variable, 191
tip(1) program, 79
TLS (Transport Layer Security), 478
    connecting to protected ports, 481–482
    enabling, 502
    host key, 479
/tmp, memory for, 65
`TMPDIR` variable, 323
tmpfs(5) program, 289
Tools directory, in Ports Collection, 366
top(1) tool, 533
    and I/O, 538
    process list for, 537–538
Transmission Control Protocol (TCP), 126
transport layer (of OSI), 126, 127, 128
transport protocol, ports, 138–139

traps in SNMP, 558
TRIM protocol, 242–243, 291
trimming kernel, 112–118
troubleshooting, 599–612. *See also* bug reports
    dependency problems, 354–355
    kernel builds, 118–119
    resources for, 601–602
TrueOS, xl
truncate(1) program, 293–294
trunking, 163
truss(1), 418–419
tsch shell, 46
    nice vs., 544
tunables, 62, 102
tunefs(8), 241–242, 249
tutorials, 8
`twist` option, for TCP connection, 460–461
typescript file, 92
`TZ` environment variable, 505
tzsetup(8), 504

U

`uappnd` flag, 193
uart(4) device driver, 78
`uchg` flag, 193
UCL (universal configuration language), 17, 587
UDF (Universal Disk Format), 283
    burning to optical media, 287–288
    creating, 287
UDP (User Datagram Protocol), 37, 126, 137
    NFS over, 303

PF and, 468
UEFI (Unified Extensible Firmware Interface), 50
and GPT, 222–223
UFS (Unix File System), xliii, 20, 231–255
adding new storage, 252–255
block and fragment size, 239–240
components, 232–233
creating and tuining, 239–243
expanding, 243
installs, 34–38
for jails, 565
minimum free space, 242
mount options, 234–237
mounting, 282
partitioning with, 23–24
and poudriere, 383
recovery and repair, 245–249
resiliency, 237–238
in single-user mode, 52–53
snapshots, 243–245
disk usage, 244–245
finding, 244
vs. journaling, 238
taking and destroying, 244
space reservations, 249
and top(1), 533–536
tuning, 241–243
`UFS_DIRHASH` option, for kernel, 114
UIDs file, in Ports Collection, 366
`umask` environment variable, 191
umount(8), 282, 285
`UNAVAIL` pool state, 275
uncompressed installation media, 26–27
uninstalling

packages, 350–351
ports, 379
universal configuration language (UCL), 17
University of California, Berkeley, xxxiv
Unix, xxxiv, xlvi–xlviii
versions, xl–xlii
Unix administrator, xlv
Unix File System (UFS), xliii, 21. *See also* UFS (Unix File System)
Unix-like, xlii
Unix Sytems Laboratories (USL), xxxvi
`UNKNOWN` rule, for TCP wrapper, 457, 458
unmounted parent datasets, 262
unmounting
filesystems, 233–237
memory disks, 291
unprivileged users, 45, 452–453
untarring, 90
`UpdateIfUnmodified` option, for freebsd update, 429
UPDATING file
for building FreeBSD, 438
in Ports Collection, 366
updating FreeBSD, source code, 436
upgrading FreeBSD, 421–450
binary updates, 428–434
checking for obsolete files, 444–445
and data risk, 428
methods, 428
optimizing and customizing, 434
packages and, 449–450
release updates, 431–434
reverting updates, 434
from source code, 435
versions, 26, 422–427

uptime, 534
USB drives
    creating key on, 597
    for tape backups, 84
    unmounting, 285
User Datagram Protocol (UDP). *See* UDP (User Datagram Protocol)
$USER environment variable, 570
user groups, 173
user ID (UID), 171, 183
user sessions, logging by, 550
userland, 97, 415, 444
    building, 438–439
    diskless client, NFS server and, 591–592
    for Linuxator, 416
username
    for dma(8), 502
    false, for Dragonfly, 501
users, xxxix
    account expiration, 176
    adding, 45–46
    changing accounts, 175–176
    deleting accounts, 178
    editing, 173–178
    filesystem mounting by, 284
    groups, 180–185
    for jail, 574
    locking accounts, 178
    NFS and, 305–306
    nobody account, 453
    resource limits, 189–190
    for running tftpd(8), 519
    security, 171–178, 185–192
        creating user, 171
    unprivileged, 45, 452–453

Uses directory, in Ports Collection, 366
/usr/campat/linux , 416
/usr/lib/compat directory, 445
/usr/local/etc/pkg/repos directory, 357
/usr/local/k1//k0etc/dhcpd.conf file, 514–515
/usr/local/lib, vs. per-port library directories, 408
/usr/local/poudriere dataset, 383
/usr/ports directory, 364
/usr/ports/INDEX file, 367–370
/usr/ports/LEGAL file, 369–370
/usr/ports/packages directory, 379
/usr/ports/UPDATING file, 392
/usr/sbin/sendmail, 499, 501
/usr/share/snmp file, 559
/usr/src/UPDATING, 437
UTC (Universal Time Clock), 504
`uunlnk` flag, 194

V

/var/cache/pkg, 345
/var/crash, 609
/var/cron file, 520
/var/db/dhcpd.leases file, 514, 589
/var/db/freebsd-update, 430
/var/db/pkg/vuln.xml file, 490
/var/messages file, errors from background fsck, 248
/var/run/dmesg.boot file, 59, 62, 84
variables. *See also* tunables
VCSW (voluntary context switches), 538
VDEVs (virtual devices), 265–267
vendors, startup/shutdown scripts, 405

verbose boot mode, 58, 59
verbose flag, for tar, 90
verbose mode, logs in, 553
`verbose_loading` variable, 16
verifying backups, 89–90
version control system, for configuration file, 16
`VersionAddendum`, 494
`vesa_load_ioctl` function, 118
`vesa_unload_ioctl` function, 118
`vfs.nfs.diskless_valid`, 592
`vfs.usermount` sysctl, 284
Vigor, 175
Vinum, 206
vipw(8) program, 176–178
virtual devices, selecting, 40
virtual disk, expanding, 223
virtual memory, bottleneck analysis with vmstat(8), 530
virtual network stack, for jails, 564
virtual processors, 400
virtual terminal, 584
virtualization, 24, 563
virtualization server, ZFS for, 21
`vlan_` variable, 165
VLAN (virtual LAN), 164–165
vmstat(8), 541
    bottleneck analysis with, 528–532
    continuous, 531–532
vnet(9), 564
vnode-backed memory disks, 290
vnodes (virtual nodes), 233
volume managers, vs. GEOM, 206
VuXML (Vulnerability and eXposure Markup Language), 490

W

`warning` log message, 547
wear-leveling, 242–243
web interface, for configuring BMCs, 76
`welcome` environment variable, 191
whatis(1), 5, 10
wildcards, for log messages, 548–549
wired memory, 536
wireless cards, 330
`WITH` environment variable, 375
WITNESS, 60
`WITNESS` kernel option, 399
`wlan` module, 562
`workgroup` keyword, for CIFS configuration, 312
workstation, security, vs. server, 199
wrappers, 454. *See also* TCP wrappers

X

X Windows, 584
`X11Forwarding`, 495
Xenix, xli
XZ compression, 91

Y

YAML, 17, 342

Z

Zetabyte Filesystem (ZFS). *See* ZFS
`zfs create` command, 261

`zfs destroy` command, 261–262
.zfs directory, 272
`zfs get` command, 260–261
`zfs list` command, 258–259
ZFS pools, 263–265
`zfs rename` command, 262
`zfs set` command, 260
ZFS (Zetabyte Filesystem), xli, xliii, 21–22, 257–279
    Advanced Replacement Cache, 536–537
    datasets, 258–263
    and disk block size, 267–268
    installs, 39–41
    for jails, 565, 581
    and poudriere, 383
    and RAID controllers, 18–19
    in single-user mode, 53
    and top(1), 536
zfs(8)
    error messages, 270
    for managing NFS, 308
`zfs_destroy` command, 273
`zfs_list` command, 272
`zfs_scrub` command, 274
`zfs_snapshot` command, 271–272
`zfs_status` command, 273
`zpool create` command, 268
`zpool get` command, 264
`zpool list` command, 263
`zpool online` command, 276
`zpool relace` command, 276
`zpool status` command, 264, 268
zpool (storage pool), 259, 263
    creating and viewing, 268–269

destroying, 270
integrity and repair, 273–276
managing, 267–270
multi-VDEV, 269–270
properties, 264–265
and RAID-Z, 267
zpool(8), error messages, 270
zsetup(8), 322
`zstatus_status` command, 274

Absolute FreeBSD, 3rd Edition is set in New Baskerville, Futura, Dogma, and TheSansMono Condensed.

There's more to keeping FreeBSD free than meets the eye.

They work hard behind the scenes and you hardly ever see them. They're The FreeBSD Foundation and they quietly fund and manage projects, sponsor FreeBSD events, Developer Summits and provide travel grants to FreeBSD developers. The FreeBSD Foundation represents the Project in executing contracts, license agreements, copyrights, trademarks, and other legal arrangements that require a recognized legal entity. The Foundation's funding and management expertise is essential to keep FreeBSD free. And keeping it free is getting more costly every year.

That's why they need your help. The work of The FreeBSD Foundation is entirely supported by your generous donations.

To make a donation visit our web site at:

www.freebsdfoundation.org

Please, help us today. Help keep FreeBSD free.

The FreeBSD™ FOUNDATION

The FreeBSD Foundation P.O. Box 20247, Boulder, CO 80308, USA
Phone: +1-720-207-5142 Fax: +720-222-2350 Web:
www.freebsdfoundation.org

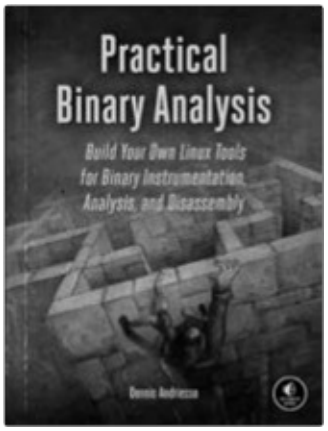
The Electronic Frontier Foundation (EFF) is the leading organization defending civil liberties in the digital world. We defend free speech on the Internet, fight illegal surveillance, promote the rights of innovators to develop new digital technologies, and work to ensure that the rights and freedoms we enjoy are enhanced — rather than eroded — as our use of technology grows.

RESOURCES

Visit *https://www.nostarch.com/absfreebsd3/* for resources, errata, and more information.

More no-nonsense books from **NO STARCH PRESS**

LINUX BASICS FOR HACKERS
Getting Started with Networking, Scripting, and Security in Kali
by OccupyTheWeb
FALL 2018, 248 PP., $34.95
ISBN 978-1-59327-855-7



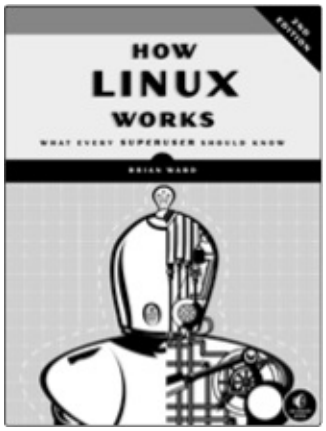
PRACTICAL BINARY ANALYSIS

Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly

by DENNIS ANDRIESSE

FALL 2018, 440 PP., $49.95

ISBN 978-1-59327-912-7



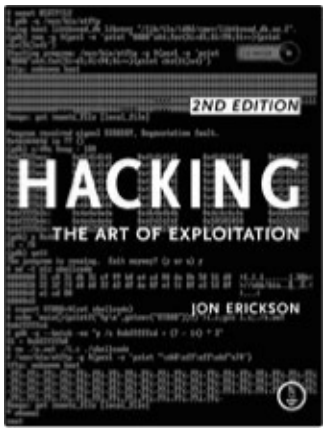
HOW LINUX WORKS, 2ND EDITION

What Every Superuser Should Know

by BRIAN WARD

NOVEMBER 2014, 392 PP., $39.95

ISBN 978-1-59327-567-9



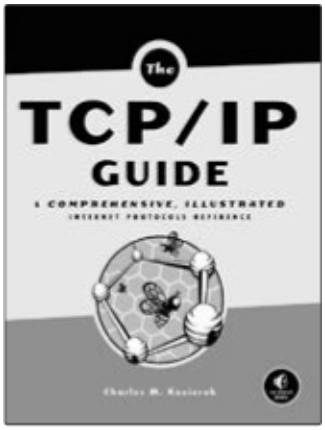
HACKING, 2ND EDITION

The Art of Exploitation

by JON ERICKSON

FEBRUARY 2008, 488 PP., $49.95

ISBN 978-1-59327-144-2



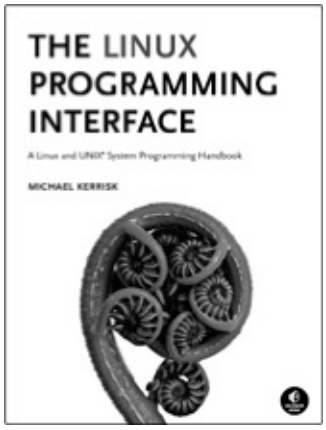
THE TCP/IP GUIDE

A Comprehensive, Illustrated Internet Protocols Reference

by CHARLES M. KOZIEROK

OCTOBER 2005, 1616 PP., $99.95

ISBN 978-1-59327-047-6



THE LINUX PROGRAMMING INTERFACE

A Linux and UNIX System Programming Handbook

by MICHAEL KERRISK

OCTOBER 2010, 1552 PP., $99.95

ISBN 978-1-59327-220-3

PHONE:
1.800.420.7240 OR

1.415.863.9900

EMAIL:
SALES@NOSTARCH.COM

WEB:
WWW.NOSTARCH.COM

THE DEFINITIVE GUIDE TO FREEBSD®

With a foreword by MARSHALL KIRK Mc KUSICK

FreeBSD—the powerful, flexible, and free Unix-like operating system—is the preferred server for many enterprises. But it can be even trickier to use than either Unix or Linux, and harder still to master.

In this completely revised and updated third edition of *Absolute FreeBSD*, FreeBSD committer Michael W. Lucas covers the newest features and teaches you how to manage FreeBSD systems. You'll dive deep into server management, learning both how things work and why they work the way they do. New to this edition is coverage of modern disks and redesigned jail and packaging systems, as well as FreeBSD transformative features designed for cloud-based management, like libxo and UCL.

You'll also learn how to:

- Choose the right filesystem for your environment
- Back up and restore critical data
- Tweak the kernel—and when not to

- Configure your network, including how to activate interfaces and select congestion control algorithms
- Manage UFS, ZFS, and other critical filesystems
- Work with advanced security features like blacklistd and packet filtering
- Implement container-style virtualization with jails
- Perform panic management and bug reporting

Whether you're a beginner simply in need of a complete introduction to FreeBSD or an experienced sysadmin or devops person looking to expand your skills, *Absolute FreeBSD* will show you how to take your FreeBSD system from "just working" to "working well." Don't leave your cubicle without it.

ABOUT THE AUTHOR

After using Unix since the late '80s and spending twenty-odd years as a network and system administrator specializing in building and maintaining high-availability systems, Michael W. Lucas now writes about them for a living. He's written more than 30 books, which have been translated into nine languages. His critically acclaimed titles include *Absolute OpenBSD*, *Cisco Routers for the Desperate*, and *PGP & GPG*, all from No Starch Press. Learn more at *https://mwl.io/*.

THE FINEST IN GEEK ENTERTAINMENT™
www.nostarch.com

FOOTNOTES

INTRODUCTION

1. At the time, several thousand dollars for a computer was dirt cheap. You young punks have no idea how good you have it.

2. If you're ever in a position where you need to prove that you are Alpha Geek amongst the pack, running Unix on a 1998 palmtop will almost certainly do it.

3. Original Trek. End of discussion. Fight me.

4. In the first edition of this book, I neglected to mention exactly how to do a similar Stupid Desktop Trick, which generated more questioning email than any other topic in the whole book. In the second edition, I swore I wouldn't make that same mistake again but neglected to mention which software package provides the run-around daemons. They say the third time's the charm.

5. This ugly thing takes the output of the last command, sorts it in reverse order by the contents of the sixth column, and presents it one screen at a time. If you have hundreds of lines of output, and you want to know which entries have the highest values in the sixth column, this is how you do it. Or, if you have lots of time, you can dump the output to a spreadsheet and fiddle with equally obscure commands for a much longer time.

1 GETTING MORE HELP

1. It's called *honesty*. IT professionals may find this term unfamiliar, but a dictionary can help.

2. Computer hardware has gotten faster and smaller, but not particularly better.

3. And it's one that's in the Handbook, the FAQ, the mailing list archives, *and* the forums.

4. Yes, there is a standard for email signatures and how you should behave on the internet. RFC 1855 should be enforced with a spiked club and a gel-fueled flamethrower.

2 BEFORE YOU INSTALL

1. The i386 platform persists despite efforts to rename it *amd32*. I mean, who bought that pricey Intel hardware anyway?

2. I no longer have customers, so, sadly, I was unable to test on their hardware.

3. ZFS expert Allan Jude often declares that disks plot against us, but a disk's plot pales next to a RAID controller's perfidy.

4. Mind you, casual thieves will consider a laptop running FreeBSD effectively encrypted anyway.

5. While I won't stand in the way of progress, I reserve the right to snicker when progress drives into the ditch.

3 INSTALLING

1. For those who skipped the Acknowledgments: Bert donated $800 to the FreeBSD Foundation in exchange for the privilege of being abused herein. I'm not gratuitously tormenting Bert; he paid real money for it.

2. Indiana, I'm looking at you.

3. This isn't true. Bert's not even that accomplished.

4 START ME UP! THE BOOT PROCESS

1. Such systems should be rare, but with your luck, you'll find one.

2. I'm confident in my sed(1) and awk(1) hackery, but not quite "run this on 400 virtual servers and go home" confident.

3. And if you've never heard of ICMP redirects, run, do not walk, to your nearest book shill and get a copy of *The TCP/IP Guide* by Charles M. Kozierok (No Starch Press, 2005). Once you have it, *read* it.

4. For the youngsters: line noise, or interference, causes random junk to appear in your terminal session. Random junk *other* than what you typed, that is.

5. You know, the hardware manual. The booklet you pitched with the server's shipping box.

6. All right, “even more insecure.” Happy?

7. This might not predate dinosaurs, but it was before spam *and* before the web. I miss that golden age.

6 KERNEL GAMES

1. Yes, I could make any number of editorial comments here, but they’re all too easy. I do have *some* standards, you know.

2. In truth, listening is optional.

3. Which anyone using a built-in floppy drive outside a lab fully deserves.

4. Actually, I probably won’t bother, as I’ll be shutting down the test host. But you get the idea.

5. You’ll discover that, actually, you failed your new kernel. But whatever.

7 THE NETWORK

1. Yes, I know about IPv4 Network Address Translation, where not all IP addresses are unique. NAT is a lie, and lying to your network is a good route to trouble—ask anyone who uses NAT on a really large scale. But even with NAT, if you’re on the public internet, your network has one or more unique IP addresses.

2. If my current research on reformatting and reinstalling users bears fruit, however, I will be certain to publish my results.

3. I’m not going to name sites accessible only via IPv4, because if I do, those sites will add IPv6 half an hour after this book reaches the printer.

4. I could go into history here, but suffice it to say: if someone tries to explain Class A, Class B, or Class C addresses to you, plug your ears and scream at them not to contaminate your brain with information made obsolete more than two decades ago.

5. Except, for some reason, many embedded devices. Put your Internet of Things behind a firewall, and don’t allow them general internet access!

6. Yes, Ethernet works over long distances if you have private fiber and multi-million-dollar switches, but if you have those you know why you’re the exception.

8 CONFIGURING NETWORKING

1. Each connection technology *traditionally* uses up-to-date cables plugged into previous-generation patch panels or vice versa. Those CAT6 cables in the CAT5 patch panel are part of a long custom of sysadmin outrage.

2. Reducing the internet's popularity might not be a bad thing, mind you . . .

3. Because your life is still worth living.

4. Some readers have already replaced their cruddy hardware before considering software optimizations. These readers may perceive this comment as unwarranted. I sincerely, wholeheartedly, and without reservation apologize to all three of you.

9 SECURING YOUR SYSTEM

1. For interactive use, that is. Never, never, *never* program in *any* C shell. Read Tom Christiansen's classic paper "Csh Programming Considered Harmful" for a full explanation.

2. Some even survived the experience.

3. Sadly, shells don't come with tachometers.

4. It's not funny enough to balance out intruders penetrating your server, of course, but it can provide a brief moment of joy in an otherwise ghastly day.

10 DISKS, PARTITIONING, AND GEOM

1. As this particular geom is part of a virtual drive, anything it says about the underlying hardware is a bald-faced lie meant to reassure you.

2. Bert.

3. In the quarter century since then, the BSD community has spent innumerable work-hours explaining and then justifying that decision. Learn from our pain. Don't port your OS to commodity hardware.

4. Experienced sysadmins should start to feel sympathetic dread right about here.

11 THE UNIX FILE SYSTEM

1. I *used* a custom block size several times, but most often I didn't *need* it and it hurt performance.

2. Again.

3. It's probably sysadmin error, but you'll probably blame the hardware.

4. In the first edition of this book, I said using fsdb(8) and clri(8) was like climbing Mount Everest in sandals and shorts. Really, it's like you're carrying your climbing guide too, except he's a chubby author who eats too much gelato and wears a heavy coat because Everest is even colder than his native Michigan. And he's live-tweeting your every misstep.

5. One could increase the reserved space percentage to make a filesystem appear extra full, thus emphasizing your manager's urgency in ramming the new disk through Purchasing. But that would be wrong.

12 THE Z FILE SYSTEM

1. Proactively adding a performance-boosting SLOG or L2ARC is a valid solution for administrative problems, like soothing the boss.

2. Probably after several meetings about why the database server is sooo blasted slow.

3. If someone is so daft as to request an account on my systems, I treat them with all the respect they deserve: none.

4. Note that I have enough self-respect not to say "scrub a scrub." Barely enough, but enough.

13 FOREIGN FILESYSTEMS

1. While formatting a flash drive touches comparatively few sectors and is probably easier on your drive than copying a large file to it, if I didn't include this warning, I'd get complaint letters. So here it is.

2. Yes, Bert put his files in a directory called *cdfiles*. I'm not totally sure he knows the difference between UDF and ISO 9660.

3. It's also a sysadmin rite of passage, so don't feel *too* bad when it happens. Just feel bad enough never to do it again.

4. Why is there no safeguard against shooting yourself in the foot like this?

Well, Unix feels that anyone dumb enough to do this doesn't deserve to be its friend. Various people keep trying to put Unix in therapy for this type of antisocial behavior, but it just isn't interested.

5. For Bert, of course, I'd add the read-only option.

14 EXPLORING /ETC

1. You think your home network is friendly? Oh, really. You really trust every device on it? Media players? Tablets? Televisions? The kids' toys? Your networked stove would stab you in the back as soon as look at you.

2. Having weird crap in *make.conf* during a system upgrade will make people laugh at you when you ask for help. But commercial software support techs do that, too, so you're probably used to it.

15 MAKING YOUR SYSTEM USEFUL

1. For decades, I threatened violence on anyone who named their software *NG*, or *Next Generation*. The name's designed to go obsolete. I've reluctantly concluded that I just don't have enough violence to go around.

2. Removal might do terrible things, but nothing that the package system can predict.

3. The answer to "Should I make my flunky do it?" is always "yes."

16 CUSTOMIZING SOFTWARE WITH PORTS

1. 28,000 ports. 62-odd categories. Some categories have 9 members. You do the math.

2. Most of this nonexportable software is available from non-US sources and can be downloaded anywhere in the world. Meanwhile, ex-KGB cryptographers without these regulations will happily provide strong crypto to anyone at low, low rates. Mind you, they charge extra for crypto without obvious backdoors.

3. Some software is sufficiently insulting by its nature and doesn't need additional help.

4. I haven't looked to see whether LibreOffice can support SNMP, but I see no reason why it wouldn't.

5. I expect the successor tool to be called *detonation* in Japanese, and then we'll have *smoking crater* in Aramaic.

6. I'm not entirely sure why the installer has a 1MB swap partition, but whatever.

17 ADVANCED SOFTWARE MANAGEMENT

1. Or, perhaps, the libraries you *believe* are on your system aren't the same as the libraries that actually *are* on your system. Never rule out your own failure until you conclusively identify the problem!

2. While most readers of this book will be sysadmins, you can tell your users to buy this book and read this section. They won't, but maybe they'll shut up and leave you alone.

3. OSF/1 is tied to defunct hardware (the awesome Alpha processor), while SVR4 is now so ancient nobody uses the feature any more. SCO Unix is hiding somewhere in shame.

4. Until you discover dtrace(1), that is.

5. Again.

18 UPGRADING FREEBSD

1. "In the data center, nobody can hear your power supply scream."

2. No, "annoying users" isn't a regular project for FreeBSD developers. It's a fringe benefit. Entirely different.

3. Various FreeBSD developers have spent the last several releases working toward packaging the base system so that the packaging tools can handle upgrades. I expect that the release of this book will prompt them to immediately solve the remaining problems and obsolete this section.

4. Note that the while CVS is gone, CTM is still alive. More than one FreeBSD developer begged me not to document it, so I won't. I await your bank transfers, gentlemen.

5. This is the voice of experience. Don't do it. Really.

6. Unless things go wrong. Then everybody cares—a *lot*.

19 ADVANCED SECURITY FEATURES

1. Instead, I order flunkies to drive in, fix the problem, and apologize for me. Problem solved.

2. After learning these things, your own screams might wake you up at night for a few years. But you'll get over it.

3. This really happened. And before you ask, no, I wasn't the recipient! A friend gave me my high-end graphics workstations. Really. And they're long obsolete now anyway. Plus, the statute of limitations is a thing.

4. Well, most of them, anyway. Quite a few. A few, at least. Oh, never mind.

5. Hi, Brad and Lucy!

6. Yes, changing `secure` to `insecure` improves security. Go figure.

7. I'd say *intruder* here except that the person in question was Ken Thompson, one of the creators of Unix and C. He had a miraculous ability to log into any Unix system, anywhere in the world, including systems developed years after he stopped working on Unix. Search out Thompson's paper "Reflections on Trusting Trust."

20 SMALL SYSTEM SERVICES

1. These links are leftovers from the days when disk space was really, really expensive. They made great sense in the 1980s. Consider this the next time you create any software.

2. It's good for him. It'll build character.

3. The most important time of all, of course, is the "time to go home."

4. I preferred fortune `-o`, but FreeBSD sadly purged the offensive fortune database.

5. Answering the question "When will it be fixed?" with "However long it takes to fix, plus however much time I waste talking to you" never goes over well.

6. We're right: you don't know how good you have it.

7. Whatever that is.

8. Unless, of course, you'd like to try installing someone else's server configuration file as the new IOS on your Cisco router. Be sure to tell the Cisco support tech to activate the phone recorder before you describe your problem; he'll want to share this one with his coworkers.

21 SYSTEM PERFORMANCE AND MONITORING

1. Once you manage dozens or hundreds of servers, you'll also find yourself installing Catci, Zabbix, Graphite, or one of their kin to monitor performance. You wanted to manage yet another application, right?

2. I won't describe it anywhere, actually. If you want to know the horribly intimate details of FreeBSD's virtual memory system, read the latest edition of "The Design and Implementation of the FreeBSD Operating System."

3. My desktop has 32GB but uses only about 4. Yes, I'm compensating for something. The 1990s.

4. Besides, sysadmins already claimed "selfish" and "cranky" for themselves.

5. Being selfish doesn't count as a good reason to `renice -20` your processes. Or so I've been told.

22 JAILS

1. This book could be described as "a not even nearly comprehensive list of things for sysadmins to worry about." This right here is the only counterexample.

2. Note that I used "sense of privacy" and not actual "privacy."

23 THE FRINGE OF FREEBSD

1. "My own code" is #8 on my list of Reasons I Shriek Obscenities in the Office.

2. Because, sadly, at some time, we all need to override the override's override.

3. Just for the record: if you have a sharp stick and the proper attitude, you can have my passphrases.

24 PROBLEM REPORTS AND PANICS

1. Also, do *try* not to swear. Much.

2. I'm increasingly convinced that the word *panic* was chosen to describe the sysadmin, not the system.

AFTERWORD

1. Yes, this includes a man who took his FreeBSD laptop on a cruise to Antarctica. With his wife. On their 20th anniversary. He's never mentioned whether his wife threw the laptop overboard or not, but if so, he's lucky he didn't follow it into the briny deeps.

2. They won't *deliberately* come for you in your sleep, though. It's a global group—they don't know when you're sleeping.