UNIX® AND LINUX® SYSTEM ADMINISTRATION HANDBOOK

FIFTH EDITION

EVI NEMETH • GARTH SNYDER • TRENT R. HEIN
BEN WHALEY • DAN MACKIN
with James Garnett, Fabrizio Branca, and Adrian Mouat

UNIX® AND LINUX® SYSTEM ADMINISTRATION HANDBOOK

FIFTH EDITION

This page intentionally left blank

UNIX® AND LINUX® SYSTEM ADMINISTRATION HANDBOOK

FIFTH EDITION

Evi Nemeth
Garth Snyder
Trent R. Hein
Ben Whaley
Dan Mackin

with James Garnett, Fabrizio Branca, and Adrian Mouat

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

Ubuntu is a registered trademark of Canonical Limited, and is used with permission.

Debian is a registered trademark of Software in the Public Interest Incorporated.

CentOS is a registered trademark of Red Hat Inc., and is used with permission.

FreeBSD is a registered trademark of The FreeBSD Foundation, and is used with permission.

The Linux Tux logo was created by Larry Ewing, lewing@isc.tamu.edu.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the web: informit.com

Library of Congress Control Number: 2017945559

Copyright © 2018 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions/.

ISBN-13: 978-0-13-427755-4
ISBN-10: 0-13-427755-4

ScoutAutomatedPrintCode

Table of Contents

Tribute to Evi **xl**

Preface **xlii**

Foreword **xliv**

Acknowledgments **xlvi**

SECTION ONE: BASIC ADMINISTRATION

Chapter 1 Where to Start **3**

Essential duties of a system administrator 4
- Controlling access. 4
- Adding hardware. 4
- Automating tasks. 4
- Overseeing backups 4
- Installing and upgrading software 5
- Monitoring. 5
- Troubleshooting 5
- Maintaining local documentation 5
- Vigilantly monitoring security 6
- Tuning performance. 6
- Developing site policies 6
- Working with vendors 6
- Fire fighting 6

Suggested background 7
Linux distributions 8
Example systems used in this book 9
    Example Linux distributions 10
    Example UNIX distribution 11
Notation and typographical conventions 12
Units 13
Man pages and other on-line documentation 14
    Organization of the man pages 14
    man: read man pages 15
    Storage of man pages 15
Other authoritative documentation 16
    System-specific guides 16
    Package-specific documentation 16
    Books 17
    RFC publications 17
Other sources of information 18
    Keeping current 18
    HowTos and reference sites 19
    Conferences 19
Ways to find and install software 19
    Determining if software is already installed 21
    Adding new software 22
    Building software from source code 23
    Installing from a web script 24
Where to host 25
Specialization and adjacent disciplines 26
    DevOps 26
    Site reliability engineers 27
    Security operations engineers 27
    Network administrators 27
    Database administrators 27
    Network operations center (NOC) engineers 27
    Data center technicians 28
    Architects 28
Recommended reading 28
    System administration and DevOps 28
    Essential tools 29

Chapter 2 Booting and System Management Daemons 30

Boot process overview . . . 30
System firmware . . . 32
  BIOS vs. UEFI . . . 32
  Legacy BIOS . . . 33
  UEFI . . . 33
Boot loaders . . . 35
GRUB: the GRand Unified Boot loader . . . 35
  GRUB configuration . . . 36
  The GRUB command line . . . 37
  Linux kernel options . . . 38
The FreeBSD boot process . . . 39
  The BIOS path: **boot0** . . . 39
  The UEFI path . . . 39
  loader configuration . . . 40
  loader commands . . . 40
System management daemons . . . 41
  Responsibilities of **init** . . . 41
  Implementations of **init** . . . 42
  Traditional **init** . . . 43
  systemd vs. the world . . . 43
  inits judged and assigned their proper punishments . . . 44
systemd in detail . . . 44
  Units and unit files . . . 45
  systemctl: manage **systemd** . . . 46
  Unit statuses . . . 47
  Targets . . . 49
  Dependencies among units . . . 50
  Execution order . . . 51
  A more complex unit file example . . . 52
  Local services and customizations . . . 53
  Service and startup control caveats . . . 54
  systemd logging . . . 56
FreeBSD **init** and startup scripts . . . 57
Reboot and shutdown procedures . . . 59
  Shutting down physical systems . . . 59
  Shutting down cloud systems . . . 59
Stratagems for a nonbooting system . . . 60
  Single-user mode . . . 61
  Single-user mode on FreeBSD . . . 62
  Single-user mode with GRUB . . . 62
  Recovery of cloud systems . . . 62

Chapter 3 Access Control and Rootly Powers 65

Standard UNIX access control . . . 66
Filesystem access control . . . 66
Process ownership . . . 67
The root account . . . 67
Setuid and setgid execution . . . 68
Management of the root account . . . 69
Root account login . . . 69
su: substitute user identity . . . 70
sudo: limited **su** . . . 70
Example configuration . . . 71
sudo pros and cons . . . 72
sudo vs. advanced access control . . . 73
Typical setup . . . 74
Environment management . . . 74
sudo without passwords . . . 75
Precedence . . . 75
sudo without a control terminal . . . 76
Site-wide **sudo** configuration . . . 76
Disabling the root account . . . 78
System accounts other than root . . . 78
Extensions to the standard access control model . . . 79
Drawbacks of the standard model . . . 80
PAM: Pluggable Authentication Modules . . . 80
Kerberos: network cryptographic authentication . . . 81
Filesystem access control lists . . . 81
Linux capabilities . . . 82
Linux namespaces . . . 82
Modern access control . . . 83
Separate ecosystems . . . 84
Mandatory access control . . . 84
Role-based access control . . . 85
SELinux: Security-Enhanced Linux . . . 85
AppArmor . . . 87
Recommended reading . . . 89

Chapter 4 Process Control 90

Components of a process . . . 90
PID: process ID number . . . 91
PPID: parent PID . . . 91
UID and EUID: real and effective user ID . . . 92
GID and EGID: real and effective group ID . . . 92
Niceness . . . 93
Control terminal . . . 93

The life cycle of a process . . . 93
Signals . . . 94
kill: send signals . . . 97
Process and thread states . . . 97
ps: monitor processes . . . 98
Interactive monitoring with **top** . . . 101
nice and **renice**: influence scheduling priority . . . 102
The **/proc** filesystem . . . 104
strace and **truss**: trace signals and system calls . . . 105
Runaway processes . . . 107
Periodic processes . . . 109
cron: schedule commands . . . 109
The format of crontab files . . . 110
Crontab management . . . 112
Other crontabs . . . 112
cron access control . . . 113
systemd timers . . . 113
Structure of **systemd** timers . . . 114
systemd timer example . . . 114
systemd time expressions . . . 116
Transient timers . . . 117
Common uses for scheduled tasks . . . 118
Sending mail . . . 118
Cleaning up a filesystem . . . 118
Rotating a log file . . . 118
Running batch jobs . . . 118
Backing up and mirroring . . . 119

Chapter 5 The Filesystem 120

Pathnames . . . 122
Filesystem mounting and unmounting . . . 122
Organization of the file tree . . . 125
File types . . . 126
Regular files . . . 129
Directories . . . 129
Hard links . . . 129
Character and block device files . . . 130
Local domain sockets . . . 131
Named pipes . . . 131
Symbolic links . . . 131

File attributes 132
The permission bits 132
The setuid and setgid bits 133
The sticky bit 134
ls: list and inspect files 134
chmod: change permissions 136
chown and **chgrp**: change ownership and group 137
umask: assign default permissions 138
Linux bonus flags 139
Access control lists 140
A cautionary note 141
ACL types 141
Implementation of ACLs 142
Linux ACL support 142
FreeBSD ACL support 143
POSIX ACLs 143
Interaction between traditional modes and ACLs 144
POSIX access determination 146
POSIX ACL inheritance 146
NFSv4 ACLs 147
NFSv4 entities for which permissions can be specified 148
NFSv4 access determination 149
ACL inheritance in NFSv4 149
NFSv4 ACL viewing 150
Interactions between ACLs and modes 151
NFSv4 ACL setup 151

Chapter 6 Software Installation and Management 153

Operating system installation 154
Installing from the network 154
Setting up PXE 155
Using kickstart, the automated installer for Red Hat and CentOS 156
Setting up a kickstart configuration file 156
Building a kickstart server 158
Pointing kickstart at your config file 158
Automating installation for Debian and Ubuntu 159
Netbooting with Cobbler, the open source Linux provisioning server 161
Automating FreeBSD installation 161
Managing packages 162
Linux package management systems 164
rpm: manage RPM packages 164
dpkg: manage **.deb** packages 166

High-level Linux package management systems . . . 166
Package repositories . . . 167
RHN: the Red Hat Network . . . 169
APT: the Advanced Package Tool . . . 169
Repository configuration . . . 170
An example **/etc/apt/sources.list** file . . . 171
Creation of a local repository mirror . . . 172
APT automation . . . 173
yum: release management for RPM . . . 174
FreeBSD software management . . . 175
The base system . . . 175
pkg: the FreeBSD package manager . . . 176
The ports collection . . . 177
Software localization and configuration . . . 178
Organizing your localization . . . 179
Structuring updates . . . 179
Limiting the field of play . . . 180
Testing . . . 180
Recommended reading . . . 181

Chapter 7 Scripting and the Shell 182

Scripting philosophy . . . 183
Write microscripts . . . 183
Learn a few tools well . . . 184
Automate all the things . . . 184
Don't optimize prematurely . . . 185
Pick the right scripting language . . . 186
Follow best practices . . . 187
Shell basics . . . 189
Command editing . . . 190
Pipes and redirection . . . 190
Variables and quoting . . . 192
Environment variables . . . 193
Common filter commands . . . 194
cut: separate lines into fields . . . 194
sort: sort lines . . . 194
uniq: print unique lines . . . 195
wc: count lines, words, and characters . . . 196
tee: copy input to two places . . . 196
head and **tail**: read the beginning or end of a file . . . 196
grep: search text . . . 197

sh scripting . . . 198
Execution . . . 198
From commands to scripts . . . 199
Input and output . . . 201
Spaces in filenames . . . 202
Command-line arguments and functions . . . 203
Control flow . . . 205
Loops . . . 207
Arithmetic . . . 209
Regular expressions . . . 209
The matching process . . . 210
Literal characters . . . 210
Special characters . . . 210
Example regular expressions . . . 211
Captures . . . 213
Greediness, laziness, and catastrophic backtracking . . . 213
Python programming . . . 215
The passion of Python 3 . . . 215
Python 2 or Python 3? . . . 216
Python quick start . . . 216
Objects, strings, numbers, lists, dictionaries, tuples, and files . . . 218
Input validation example . . . 220
Loops . . . 221
Ruby programming . . . 223
Installation . . . 223
Ruby quick start . . . 224
Blocks . . . 225
Symbols and option hashes . . . 227
Regular expressions in Ruby . . . 227
Ruby as a filter . . . 229
Library and environment management for Python and Ruby . . . 229
Finding and installing packages . . . 229
Creating reproducible environments . . . 230
Multiple environments . . . 231
virtualenv: virtual environments for Python . . . 232
RVM: the Ruby enVironment Manager . . . 232
Revision control with Git . . . 235
A simple Git example . . . 236
Git caveats . . . 239
Social coding with Git . . . 239
Recommended reading . . . 241
Shells and shell scripting . . . 241
Regular expressions . . . 241
Python . . . 242
Ruby . . . 242

Chapter 8 User Management 243

Account mechanics . . . 244
The **/etc/passwd** file . . . 245
    Login name . . . 245
    Encrypted password . . . 246
    UID (user ID) number . . . 248
    Default GID (group ID) number . . . 249
    GECOS field . . . 249
    Home directory . . . 250
    Login shell . . . 250
The Linux **/etc/shadow** file . . . 250
FreeBSD's **/etc/master.passwd** and **/etc/login.conf** files . . . 252
    The **/etc/master.passwd** file . . . 252
    The **/etc/login.conf** file . . . 253
The **/etc/group** file . . . 254
Manual steps for adding users . . . 255
    Editing the **passwd** and **group** files . . . 256
    Setting a password . . . 257
    Creating the home directory and installing startup files . . . 257
    Setting home directory permissions and ownerships . . . 259
    Configuring roles and administrative privileges . . . 259
    Finishing up . . . 260
Scripts for adding users: **useradd**, **adduser**, and **newusers** . . . 260
    useradd on Linux . . . 261
    adduser on Debian and Ubuntu . . . 262
    adduser on FreeBSD . . . 262
    newusers on Linux: adding in bulk . . . 263
Safe removal of a user's account and files . . . 264
User login lockout . . . 265
Risk reduction with PAM . . . 266
Centralized account management . . . 266
    LDAP and Active Directory . . . 267
    Application-level single sign-on systems . . . 267
    Identity management systems . . . 268

Chapter 9 Cloud Computing 270

The cloud in context . . . 271
Cloud platform choices . . . 273
    Public, private, and hybrid clouds . . . 273
    Amazon Web Services . . . 274
    Google Cloud Platform . . . 275
    DigitalOcean . . . 275

Cloud service fundamentals 276
Access to the cloud 277
Regions and availability zones 278
Virtual private servers 279
Networking 280
Storage 281
Identity and authorization 281
Automation 282
Serverless functions 282
Clouds: VPS quick start by platform 283
Amazon Web Services 283
aws: control AWS subsystems 284
Creating an EC2 instance 284
Viewing the console log 286
Stopping and terminating instances 287
Google Cloud Platform 288
Setting up **gcloud** 288
Running an instance on GCE 288
DigitalOcean 289
Cost control 291
Recommended Reading 293

Chapter 10 Logging 294

Log locations 296
Files not to manage 298
How to view logs in the **systemd** journal 298
The **systemd** journal 299
Configuring the **systemd** journal 300
Adding more filtering options for **journalctl** 301
Coexisting with syslog 301
Syslog 302
Reading syslog messages 303
Rsyslog architecture 304
Rsyslog versions 304
Rsyslog configuration 305
Modules 306
sysklogd syntax 307
Legacy directives 311
RainerScript 312
Config file examples 314
Basic rsyslog configuration 314
Network logging client 315
Central logging host 316
Syslog message security 317
Syslog configuration debugging 318

Kernel and boot-time logging . . . 318
Management and rotation of log files . . . 319
logrotate: cross-platform log management . . . 319
newsyslog: log management on FreeBSD . . . 321
Management of logs at scale . . . 321
The ELK stack . . . 321
Graylog . . . 322
Logging as a service . . . 323
Logging policies . . . 323

Chapter 11 Drivers and the Kernel 325

Kernel chores for system administrators . . . 326
Kernel version numbering . . . 327
Linux kernel versions . . . 327
FreeBSD kernel versions . . . 328
Devices and their drivers . . . 328
Device files and device numbers . . . 329
Challenges of device file management . . . 330
Manual creation of device files . . . 331
Modern device file management . . . 331
Linux device management . . . 331
Sysfs: a window into the souls of devices . . . 332
udevadm: explore devices . . . 333
Rules and persistent names . . . 334
FreeBSD device management . . . 337
Devfs: automatic device file configuration . . . 337
devd: higher-level device management . . . 338
Linux kernel configuration . . . 339
Tuning Linux kernel parameters . . . 339
Building a custom kernel . . . 341
If it ain't broke, don't fix it . . . 341
Setting up to build the Linux kernel . . . 341
Configuring kernel options . . . 342
Building the kernel binary . . . 343
Adding a Linux device driver . . . 344
FreeBSD kernel configuration . . . 344
Tuning FreeBSD kernel parameters . . . 344
Building a FreeBSD kernel . . . 345
Loadable kernel modules . . . 346
Loadable kernel modules in Linux . . . 346
Loadable kernel modules in FreeBSD . . . 348
Booting . . . 348
Linux boot messages . . . 349
FreeBSD boot messages . . . 353

Booting alternate kernels in the cloud . . . 355
Kernel errors . . . 356
Linux kernel errors . . . 356
FreeBSD kernel panics . . . 359
Recommended reading . . . 359

Chapter 12 Printing 360
CUPS printing . . . 361
Interfaces to the printing system . . . 361
The print queue . . . 362
Multiple printers and queues . . . 363
Printer instances . . . 363
Network printer browsing . . . 363
Filters . . . 364
CUPS server administration . . . 365
Network print server setup . . . 365
Printer autoconfiguration . . . 366
Network printer configuration . . . 367
Printer configuration examples . . . 367
Service shutoff . . . 368
Other configuration tasks . . . 368
Troubleshooting tips . . . 369
Print daemon restart . . . 369
Log files . . . 369
Direct printing connections . . . 370
Network printing problems . . . 370
Recommended reading . . . 371

SECTION TWO: NETWORKING

Chapter 13 TCP/IP Networking 375
TCP/IP and its relationship to the Internet . . . 375
Who runs the Internet? . . . 376
Network standards and documentation . . . 376
Networking basics . . . 378
IPv4 and IPv6 . . . 379
Packets and encapsulation . . . 381
Ethernet framing . . . 382
Maximum transfer unit . . . 382

Packet addressing 384
Hardware (MAC) addressing 384
IP addressing 385
Hostname "addressing" 385
Ports 385
Address types 386
IP addresses: the gory details 387
IPv4 address classes 387
IPv4 subnetting 388
Tricks and tools for subnet arithmetic 390
CIDR: Classless Inter-Domain Routing 391
Address allocation 392
Private addresses and network address translation (NAT) 392
IPv6 addressing 394
IPv6 address notation 395
IPv6 prefixes 396
Automatic host numbering 397
Stateless address autoconfiguration 397
IPv6 tunneling 398
IPv6 information sources 398
Routing 398
Routing tables 399
ICMP redirects 401
IPv4 ARP and IPv6 neighbor discovery 401
DHCP: the Dynamic Host Configuration Protocol 402
DHCP software 403
DHCP behavior 404
ISC's DHCP software 404
Security issues 406
IP forwarding 406
ICMP redirects 407
Source routing 407
Broadcast pings and other directed broadcasts 407
IP spoofing 408
Host-based firewalls 408
Virtual private networks 409
Basic network configuration 410
Hostname and IP address assignment 411
Network interface and IP configuration 412
Routing configuration 414
DNS configuration 415
System-specific network configuration 416

Linux networking . . . 417
NetworkManager . . . 417
ip: manually configure a network . . . 418
Debian and Ubuntu network configuration . . . 419
Red Hat and CentOS network configuration . . . 419
Linux network hardware options . . . 421
Linux TCP/IP options . . . 422
Security-related kernel variables . . . 424
FreeBSD networking . . . 425
ifconfig: configure network interfaces . . . 425
FreeBSD network hardware configuration . . . 426
FreeBSD boot-time network configuration . . . 426
FreeBSD TCP/IP configuration . . . 427
Network troubleshooting . . . 428
ping: check to see if a host is alive . . . 429
traceroute: trace IP packets . . . 431
Packet sniffers . . . 434
tcpdump: command-line packet sniffer . . . 435
Wireshark and TShark: **tcpdump** on steroids . . . 436
Network monitoring . . . 437
SmokePing: gather ping statistics over time . . . 437
iPerf: track network performance . . . 437
Cacti: collect and graph data . . . 438
Firewalls and NAT . . . 440
Linux **iptables**: rules, chains, and tables . . . 440
iptables rule targets . . . 441
iptables firewall setup . . . 442
A complete example . . . 442
Linux NAT and packet filtering . . . 444
IPFilter for UNIX systems . . . 445
Cloud networking . . . 448
AWS's virtual private cloud (VPC) . . . 448
Subnets and routing tables . . . 449
Security groups and NACLs . . . 450
A sample VPC architecture . . . 451
Creating a VPC with Terraform . . . 452
Google Cloud Platform networking . . . 455
DigitalOcean networking . . . 456
Recommended reading . . . 457
History . . . 457
Classics and bibles . . . 458
Protocols . . . 458

Chapter 14 Physical Networking 459

Ethernet: the Swiss Army knife of networking . . . 460
Ethernet signaling . . . 460
Ethernet topology . . . 461
Unshielded twisted-pair cabling . . . 462
Optical fiber . . . 464
Ethernet connection and expansion . . . 465
Hubs . . . 465
Switches . . . 465
VLAN-capable switches . . . 466
Routers . . . 467
Autonegotiation . . . 467
Power over Ethernet . . . 468
Jumbo frames . . . 468
Wireless: Ethernet for nomads . . . 469
Wireless standards . . . 469
Wireless client access . . . 470
Wireless infrastructure and WAPs . . . 470
Wireless topology . . . 471
Small money wireless . . . 472
Big money wireless . . . 472
Wireless security . . . 473
SDN: software-defined networking . . . 473
Network testing and debugging . . . 474
Building wiring . . . 475
UTP cabling options . . . 475
Connections to offices . . . 475
Wiring standards . . . 475
Network design issues . . . 476
Network architecture vs. building architecture . . . 477
Expansion . . . 477
Congestion . . . 478
Maintenance and documentation . . . 478
Management issues . . . 478
Recommended vendors . . . 479
Cables and connectors . . . 479
Test equipment . . . 480
Routers/switches . . . 480
Recommended reading . . . 480

Chapter 15 IP Routing 481

Packet forwarding: a closer look. 482
Routing daemons and routing protocols. 485
    Distance-vector protocols 486
    Link-state protocols 487
    Cost metrics. 487
    Interior and exterior protocols 488
Protocols on parade 488
    RIP and RIPng: Routing Information Protocol 488
    OSPF: Open Shortest Path First 489
    EIGRP: Enhanced Interior Gateway Routing Protocol 490
    BGP: Border Gateway Protocol 490
Routing protocol multicast coordination 490
Routing strategy selection criteria 490
Routing daemons 492
    routed: obsolete RIP implementation 492
    Quagga: mainstream routing daemon 493
    XORP: router in a box 494
Cisco routers 494
Recommended reading 496

Chapter 16 DNS: The Domain Name System 498

DNS architecture 499
    Queries and responses 499
    DNS service providers 500
DNS for lookups 500
    resolv.conf: client resolver configuration 500
    nsswitch.conf: who do I ask for a name? 501
The DNS namespace. 502
    Registering a domain name. 503
    Creating your own subdomains 503
How DNS works 503
    Name servers 504
    Authoritative and caching-only servers 505
    Recursive and nonrecursive servers 505
    Resource records. 506
    Delegation 506
    Caching and efficiency 508
    Multiple answers and round robin DNS load balancing. 508
    Debugging with query tools 509
The DNS database. 512
    Parser commands in zone files 512
    Resource records 513
    The SOA record. 516

NS records 518
A records 519
AAAA records 519
PTR records 520
MX records 521
CNAME records 522
SRV records 523
TXT records. 524
SPF, DKIM, and DMARC records 525
DNSSEC records 525
The BIND software 525
Components of BIND 525
Configuration files 526
The `include` statement 527
The `options` statement 528
The `acl` statement 534
The (TSIG) `key` statement 534
The `server` statement 535
The `masters` statement 535
The `logging` statement 536
The `statistics-channels` statement 536
The `zone` statement 536
Configuring the primary server for a zone 537
Configuring a secondary server for a zone 538
Setting up the root server hints 539
Setting up a forwarding zone 539
The `controls` statement for **rndc** 540
Split DNS and the `view` statement 541
BIND configuration examples 543
The localhost zone 543
A small security company 544
Zone file updating 547
Zone transfers 548
Dynamic updates 549
DNS security issues 551
Access control lists in BIND, revisited 552
Open resolvers 553
Running in a **chroot**ed jail 554
Secure server-to-server communication with TSIG and TKEY 554
Setting up TSIG for BIND 555
DNSSEC 557
DNSSEC policy 558
DNSSEC resource records 558
Turning on DNSSEC 560
Key pair generation 560

Zone signing 562
The DNSSEC chain of trust 564
DNSSEC key rollover 565
DNSSEC tools 566
ldns tools, nlnetlabs.nl/projects/ldns 566
dnssec-tools.org 566
RIPE tools, ripe.net 567
OpenDNSSEC, opendnssec.org 567
Debugging DNSSEC 567
BIND debugging 568
Logging in BIND 568
Channels 569
Categories 570
Log messages 570
Sample BIND logging configuration 573
Debug levels in BIND 573
Name server control with **rndc** 574
Command-line querying for lame delegations 575
Recommended reading 576
Books and other documentation 577
On-line resources 577
The RFCs 577

Chapter 17 Single Sign-On 578

Core SSO elements 579
LDAP: “lightweight” directory services 580
Uses for LDAP 580
The structure of LDAP data 581
OpenLDAP: the traditional open source LDAP server 582
389 Directory Server: alternative open source LDAP server 583
LDAP Querying 584
Conversion of **passwd** and **group** files to LDAP 585
Using directory services for login 586
Kerberos 586
Linux Kerberos configuration for AD integration 587
FreeBSD Kerberos configuration for AD integration 587
sssd: the System Security Services Daemon 589
nsswitch.conf: the name service switch 590
PAM: cooking spray or authentication wonder? 590
PAM configuration 591
PAM example 592
Alternative approaches 594
NIS: the Network Information Service 594
rsync: transfer files securely 594
Recommended reading 595

Chapter 18 Electronic Mail 596

Mail system architecture 597
User agents 597
Submission agents 598
Transport agents 598
Local delivery agents 599
Message stores 599
Access agents 599
Anatomy of a mail message 600
The SMTP protocol 603
You had me at EHLO 604
SMTP error codes 604
SMTP authentication 604
Spam and malware 605
Forgeries 606
SPF and Sender ID 606
DKIM 607
Message privacy and encryption 607
Mail aliases 608
Getting aliases from files 610
Mailing to files 611
Mailing to programs 611
Building the hashed alias database 612
Email configuration 612
sendmail 613
The switch file 614
Starting **sendmail** 615
Mail queues 616
sendmail configuration 617
The **m4** preprocessor 617
The **sendmail** configuration pieces 618
A configuration file built from a sample **.mc** file 619
Configuration primitives 620
Tables and databases 620
Generic macros and features 621
`OSTYPE` macro 621
`DOMAIN` macro 621
`MAILER` macro 622
`FEATURE` macro 622
`use_cw_file` feature 622
`redirect` feature 623
`always_add_domain` feature 623
`access_db` feature 623
`virtusertable` feature 624

ldap_routing feature . . . 624
Masquerading features . . . 625
MAIL_HUB and SMART_HOST macros . . . 626
Client configuration . . . 626
m4 configuration options . . . 627
Spam-related features in **sendmail** . . . 628
Relay control . . . 629
User or site blacklisting . . . 630
Throttles, rates, and connection limits . . . 631
Security and **sendmail** . . . 632
Ownerships . . . 633
Permissions . . . 634
Safer mail to files and programs . . . 634
Privacy options . . . 635
Running a **chroot**ed **sendmail** (for the truly paranoid) . . . 636
Denial of service attacks . . . 636
TLS: Transport Layer Security . . . 637
sendmail testing and debugging . . . 638
Queue monitoring . . . 638
Logging . . . 639
Exim . . . 640
Exim installation . . . 640
Exim startup . . . 642
Exim utilities . . . 642
Exim configuration language . . . 643
Exim configuration file . . . 644
Global options . . . 645
Options . . . 645
Lists . . . 646
Macros . . . 647
Access control lists (ACLs) . . . 647
Content scanning at ACL time . . . 650
Authenticators . . . 651
Routers . . . 652
The accept router . . . 653
The dnslookup router . . . 653
The manualroute router . . . 653
The redirect router . . . 654
Per-user filtering through **.forward** files . . . 655
Transports . . . 655
The appendfile transport . . . 655
The smtp transport . . . 656
Retry configuration . . . 656
Rewriting configuration . . . 657
Local scan function . . . 657

Logging. 657
Debugging 658
Postfix. 658
Postfix architecture 659
Receiving mail 659
Managing mail-waiting queues. 660
Sending mail 660
Security 661
Postfix commands and documentation 661
Postfix configuration 661
What to put in **main.cf**. 662
Basic settings 662
Null client. 662
Use of **postconf** 663
Lookup tables 663
Local delivery 664
Virtual domains. 665
Virtual alias domains 666
Virtual mailbox domains 667
Access control 667
Access tables. 669
Authentication of clients and encryption 670
Debugging 670
Looking at the queue 671
Soft-bouncing 671
Recommended reading 672
sendmail references 672
Exim references. 672
Postfix references 672
RFCs 673

Chapter 19 Web Hosting 674

HTTP: the Hypertext Transfer Protocol 674
Uniform Resource Locators (URLs) 675
Structure of an HTTP transaction 676
HTTP requests 677
HTTP responses 677
Headers and the message body 678
curl: HTTP from the command line 679
TCP connection reuse 680
HTTP over TLS. 681
Virtual hosts. 681

Web software basics 682
Web servers and HTTP proxy software 683
Load balancers 684
Caches 686
Browser caches 687
Proxy cache 688
Reverse proxy cache 688
Cache problems 688
Cache software 689
Content delivery networks 689
Languages of the web 691
Ruby 691
Python 691
Java 691
Node.js 691
PHP 692
Go 692
Application programming interfaces (APIs) 692
Web hosting in the cloud 694
Build versus buy 694
Platform-as-a-Service 695
Static content hosting 695
Serverless web applications 696
Apache **httpd** 696
httpd in use 697
httpd configuration logistics 698
Virtual host configuration 699
HTTP basic authentication 701
Configuring TLS 702
Running web applications within Apache 702
Logging 703
NGINX 704
Installing and running NGINX 704
Configuring NGINX 705
Configuring TLS for NGINX 708
Load balancing with NGINX 708
HAProxy 710
Health checks 711
Server statistics 712
Sticky sessions 712
TLS termination 713
Recommended reading 714

SECTION THREE: STORAGE

Chapter 20 Storage 717

I just want to add a disk! 718
Linux recipe.....719
FreeBSD recipe.....720
Storage hardware.....721
Hard disks.....722
Hard disk reliability.....723
Failure modes and metrics.....723
Drive types.....724
Warranties and retirement.....725
Solid state disks.....725
Rewritability limits.....726
Flash memory and controller types.....726
Page clusters and pre-erasing.....727
SSD reliability.....727
Hybrid drives.....728
Advanced Format and 4KiB blocks.....729
Storage hardware interfaces.....730
The SATA interface.....730
The PCI Express interface.....730
The SAS interface.....731
USB.....732
Attachment and low-level management of drives.....733
Installation verification at the hardware level.....733
Disk device files.....734
Ephemeral device names.....735
Formatting and bad block management.....735
ATA secure erase.....737
hdparm and **camcontrol**: set disk and interface parameters.....738
Hard disk monitoring with SMART.....738
The software side of storage: peeling the onion.....739
Elements of a storage system.....740
The Linux device mapper.....742
Disk partitioning.....742
Traditional partitioning.....744
MBR partitioning.....745
GPT: GUID partition tables.....746
Linux partitioning.....746
FreeBSD partitioning.....747

Logical volume management 747
Linux logical volume management 748
Volume snapshots 750
Filesystem resizing 751
FreeBSD logical volume management 753
RAID: redundant arrays of inexpensive disks 753
Software vs. hardware RAID 753
RAID levels 754
Disk failure recovery 756
Drawbacks of RAID 5 757
mdadm: Linux software RAID 758
Creating an array 758
mdadm.conf: document array configuration 760
Simulating a failure 761
Filesystems 762
Traditional filesystems: UFS, ext4, and XFS 763
Filesystem terminology 764
Filesystem polymorphism 765
Filesystem formatting 766
fsck: check and repair filesystems 766
Filesystem mounting 767
Setup for automatic mounting 768
USB drive mounting 770
Swapping recommendations 770
Next-generation filesystems: ZFS and Btrfs 772
Copy-on-write 772
Error detection 772
Performance 773
ZFS: all your storage problems solved 773
ZFS on Linux 774
ZFS architecture 774
Example: disk addition 775
Filesystems and properties 776
Property inheritance 777
One filesystem per user 778
Snapshots and clones 779
Raw volumes 780
Storage pool management 781
Btrfs: "ZFS lite" for Linux 783
Btrfs vs. ZFS 783
Setup and storage conversion 784
Volumes and subvolumes 786
Volume snapshots 787
Shallow copies 788

Data backup strategy 788
Recommended reading 790

Chapter 21 The Network File System 791

Meet network file services 791
The competition 792
Issues of state 792
Performance concerns 793
Security 793
The NFS approach 794
Protocol versions and history 794
Remote procedure calls 795
Transport protocols 795
State 796
Filesystem exports 796
File locking 797
Security concerns 798
Identity mapping in version 4 799
Root access and the nobody account 800
Performance considerations in version 4 801
Server-side NFS 801
Linux **exports** 802
FreeBSD **exports** 804
nfsd: serve files 806
Client-side NFS 807
Mounting remote filesystems at boot time 810
Restricting exports to privileged ports 810
Identity mapping for NFS version 4 810
nfsstat: dump NFS statistics 811
Dedicated NFS file servers 812
Automatic mounting 812
Indirect maps 814
Direct maps 814
Master maps 815
Executable maps 815
Automount visibility 816
Replicated filesystems and **automount** 816
Automatic automounts (V3; all but Linux) 817
Specifics for Linux 817
Recommended reading 818

Chapter 22 SMB 819

Samba: SMB server for UNIX. 820
Installing and configuring Samba. 821
File sharing with local authentication 822
File sharing with accounts authenticated by Active Directory. 822
Configuring shares 823
Sharing home directories. 823
Sharing project directories 824
Mounting SMB file shares 825
Browsing SMB file shares. 826
Ensuring Samba security 826
Debugging Samba. 827
Querying Samba's state with **smbstatus** 827
Configuring Samba logging. 828
Managing character sets 829
Recommended reading 829

SECTION FOUR: OPERATIONS

Chapter 23 Configuration Management 833

Configuration management in a nutshell 834
Dangers of configuration management 834
Elements of configuration management 835
Operations and parameters 835
Variables 837
Facts 838
Change handlers 838
Bindings 838
Bundles and bundle repositories 839
Environments 839
Client inventory and registration 840
Popular CM systems compared 841
Terminology 842
Business models 842
Architectural options 843
Language options 845
Dependency management options. 846
General comments on Chef. 848
General comments on Puppet 849
General comments on Ansible and Salt. 850
YAML: a rant 850

Introduction to Ansible . . . 852
Ansible example . . . 853
Client setup . . . 855
Client groups . . . 857
Variable assignments . . . 858
Dynamic and computed client groups . . . 859
Task lists . . . 860
`state` parameters . . . 862
Iteration . . . 862
Interaction with Jinja . . . 863
Template rendering . . . 863
Bindings: plays and playbooks . . . 864
Roles . . . 866
Recommendations for structuring the configuration base . . . 868
Ansible access options . . . 869
Introduction to Salt . . . 871
Minion setup . . . 873
Variable value binding for minions . . . 874
Minion matching . . . 876
Salt states . . . 877
Salt and Jinja . . . 878
State IDs and dependencies . . . 880
State and execution functions . . . 882
Parameters and names . . . 883
State binding to minions . . . 886
Highstates . . . 886
Salt formulas . . . 887
Environments . . . 888
Documentation roadmap . . . 892
Ansible and Salt compared . . . 893
Deployment flexibility and scalability . . . 893
Built-in modules and extensibility . . . 894
Security . . . 894
Miscellaneous . . . 895
Best practices . . . 895
Recommended reading . . . 899

Chapter 24 Virtualization 900

Virtual vernacular . . . 901
Hypervisors . . . 901
Full virtualization . . . 901
Paravirtualization . . . 902
Hardware-assisted virtualization . . . 902
Paravirtualized drivers . . . 902

Modern virtualization 903
Type 1 vs. type 2 hypervisors....903
Live migration....904
Virtual machine images....904
Containerization....904
Virtualization with Linux....905
Xen....906
Xen guest installation....907
KVM....908
KVM guest installation....909
FreeBSD bhyve....910
VMware....910
VirtualBox....911
Packer....911
Vagrant....913
Recommended reading....914

Chapter 25 Containers 915

Background and core concepts....916
Kernel support....917
Images....917
Networking....918
Docker: the open source container engine....919
Basic architecture....919
Installation....921
Client setup....921
The container experience....922
Volumes....926
Data volume containers....927
Docker networks....927
Namespaces and the bridge network....928
Network overlays....930
Storage drivers....930
dockerd option editing....930
Image building....932
Choosing a base image....933
Building from a **Dockerfile**....933
Composing a derived **Dockerfile**....934
Registries....936

Containers in practice 937
Logging. 938
Security advice 939
Restrict access to the daemon 939
Use TLS 940
Run processes as unprivileged users 940
Use a read-only root filesystem. 941
Limit capabilities. 941
Secure images 941
Debugging and troubleshooting 942
Container clustering and management 942
A synopsis of container management software. 944
Kubernetes 944
Mesos and Marathon 946
Docker Swarm. 947
AWS EC2 Container Service 947
Recommended reading 948

Chapter 26 Continuous Integration and Delivery 949

CI/CD essentials 951
Principles and practices 951
Use revision control 952
Build once, deploy often 952
Automate end-to-end 952
Build every integration commit 952
Share responsibility. 953
Build fast, fix fast 953
Audit and verify. 953
Environments 953
Feature flags. 955
Pipelines. 955
The build process. 956
Testing 957
Deployment. 959
Zero-downtime deployment techniques 960
Jenkins: the open source automation server 961
Basic Jenkins concepts 962
Distributed builds 963
Pipeline as code. 963

CI/CD in practice 964
UlsahGo, a trivial web application 966
Unit testing UlsahGo 966
Taking first steps with the Jenkins Pipeline....968
Building a DigitalOcean image 970
Provisioning a single system for testing....972
Testing the droplet 975
Deploying UlsahGo to a pair of droplets and a load balancer 976
Concluding the demonstration pipeline 977
Containers and CI/CD....978
Containers as a build environment 979
Container images as build artifacts 979
Recommended reading 980

Chapter 27 Security 981

Elements of security....983
How security is compromised....983
Social engineering 983
Software vulnerabilities 984
Distributed denial-of-service attacks (DDoS) 985
Insider abuse 986
Network, system, or application configuration errors....986
Basic security measures 987
Software updates 987
Unnecessary services 988
Remote event logging....989
Backups 989
Viruses and worms 989
Root kits....990
Packet filtering....991
Passwords and multifactor authentication 991
Vigilance....991
Application penetration testing....992
Passwords and user accounts 992
Password changes 993
Password vaults and password escrow....993
Password aging 995
Group logins and shared logins 996
User shells 996
Rootly entries 996

Security power tools....996
Nmap: network port scanner....996
Nessus: next-generation network scanner....998
Metasploit: penetration testing software....999
Lynis: on-box security auditing....999
John the Ripper: finder of insecure passwords....1000
Bro: the programmable network intrusion detection system....1000
Snort: the popular network intrusion detection system....1001
OSSEC: host-based intrusion detection....1002
OSSEC basic concepts....1002
OSSEC installation....1003
OSSEC configuration....1004
Fail2Ban: brute-force attack response system....1004
Cryptography primer....1005
Symmetric key cryptography....1005
Public key cryptography....1006
Public key infrastructure....1007
Transport Layer Security....1009
Cryptographic hash functions....1009
Random number generation....1011
Cryptographic software selection....1012
The **openssl** command....1012
Preparing keys and certificates....1013
Debugging TLS servers....1014
PGP: Pretty Good Privacy....1014
Kerberos: a unified approach to network security....1015
SSH, the Secure SHell....1016
OpenSSH essentials....1016
The **ssh** client....1018
Public key authentication....1019
The **ssh-agent**....1020
Host aliases in **~/.ssh/config**....1022
Connection multiplexing....1023
Port forwarding....1023
sshd: the OpenSSH server....1024
Host key verification with SSHFP....1026
File transfers....1027
Alternatives for secure logins....1027
Firewalls....1027
Packet-filtering firewalls....1028
Filtering of services....1028
Stateful inspection firewalls....1029
Firewalls: safe?....1029

Virtual private networks (VPNs) . . . 1030
IPsec tunnels . . . 1030
All I need is a VPN, right? . . . 1031
Certifications and standards . . . 1031
Certifications . . . 1031
Security standards . . . 1032
ISO 27001:2013 . . . 1032
PCI DSS . . . 1033
NIST 800 series . . . 1033
The Common Criteria . . . 1034
OWASP: the Open Web Application Security Project . . . 1034
CIS: the Center for Internet Security . . . 1034
Sources of security information . . . 1034
SecurityFocus.com, the BugTraq mailing list, and the OSS mailing list . . . 1035
Schneier on Security . . . 1035
The Verizon Data Breach Investigations Report . . . 1035
The SANS Institute . . . 1035
Distribution-specific security resources . . . 1036
Other mailing lists and web sites . . . 1036
When your site has been attacked . . . 1037
Recommended reading . . . 1038

Chapter 28 Monitoring 1040

An overview of monitoring . . . 1041
Instrumentation . . . 1042
Data types . . . 1042
Intake and processing . . . 1043
Notifications . . . 1043
Dashboards and UIs . . . 1044
The monitoring culture . . . 1044
The monitoring platforms . . . 1045
Open source real-time platforms . . . 1046
Nagios and Icinga . . . 1046
Sensu . . . 1047
Open source time-series platforms . . . 1047
Graphite . . . 1047
Prometheus . . . 1048
InfluxDB . . . 1049
Munin . . . 1049
Open source charting platforms . . . 1049
Commercial monitoring platforms . . . 1050
Hosted monitoring platforms . . . 1051

Data collection . . . 1051
StatsD: generic data submission protocol . . . 1052
Data harvesting from command output . . . 1054
Network monitoring . . . 1055
Systems monitoring . . . 1056
Commands for systems monitoring. . . . 1057
collectd: generalized system data harvester . . . 1057
sysdig and **dtrace**: execution tracers . . . 1058
Application monitoring . . . 1059
Log monitoring . . . 1059
Supervisor + Munin: a simple option for limited domains . . . 1060
Commercial application monitoring tools . . . 1060
Security monitoring . . . 1061
System integrity verification . . . 1061
Intrusion detection monitoring . . . 1062
SNMP: the Simple Network Management Protocol . . . 1063
SNMP organization . . . 1064
SNMP protocol operations . . . 1065
Net-SNMP: tools for servers . . . 1065
Tips and tricks for monitoring . . . 1068
Recommended reading . . . 1069

Chapter 29 Performance Analysis 1070

Performance tuning philosophy . . . 1071
Ways to improve performance . . . 1073
Factors that affect performance . . . 1074
Stolen CPU cycles . . . 1075
Analysis of performance problems . . . 1076
System performance checkup . . . 1077
Taking stock of your equipment . . . 1077
Gathering performance data . . . 1079
Analyzing CPU usage . . . 1079
Understanding how the system manages memory . . . 1081
Analyzing memory usage . . . 1082
Analyzing disk I/O . . . 1084
fio: testing storage subsystem performance . . . 1085
sar: collecting and reporting statistics over time . . . 1086
Choosing a Linux I/O scheduler . . . 1086
perf: profiling Linux systems in detail . . . 1087
Help! My server just got really slow! . . . 1088
Recommended reading . . . 1090

Chapter 30 Data Center Basics 1091

Racks. 1092
Power 1092
Rack power requirements 1093
kVA vs. kW 1094
Energy efficiency. 1095
Metering. 1095
Cost. 1096
Remote control 1096
Cooling and environment 1096
Cooling load estimation. 1097
Roof, walls, and windows 1097
Electronic gear. 1097
Light fixtures 1098
Operators 1098
Total heat load 1098
Hot aisles and cold aisles 1098
Humidity 1100
Environmental monitoring 1100
Data center reliability tiers 1101
Data center security 1102
Location 1102
Perimeter 1102
Facility access 1102
Rack access 1103
Tools 1103
Recommended reading 1104

Chapter 31 Methodology, Policy, and Politics 1105

The grand unified theory: DevOps. 1106
DevOps is CLAMS 1107
Culture 1107
Lean 1108
Automation 1109
Measurement. 1110
Sharing 1110
System administration in a DevOps world 1110
Ticketing and task management systems 1111
Common functions of ticketing systems. 1112
Ticket ownership. 1112
User acceptance of ticketing systems 1113
Sample ticketing systems 1114
Ticket dispatching. 1114

Local documentation maintenance 1115
Infrastructure as code 1116
Documentation standards 1116
Environment separation 1118
Disaster management 1119
Risk assessment 1119
Recovery planning 1120
Staffing for a disaster 1121
Security incidents 1122
IT policies and procedures 1122
The difference between policies and procedures 1123
Policy best practices 1124
Procedures 1124
Service level agreements 1125
Scope and descriptions of services 1125
Queue prioritization policies 1126
Conformance measurements 1127
Compliance: regulations and standards 1127
Legal issues 1131
Privacy 1131
Policy enforcement 1132
Control = liability 1132
Software licenses 1133
Organizations, conferences, and other resources 1133
Recommended reading 1135

Index 1136
A Brief History of System Administration 1166
Colophon 1176
About the Contributors 1178
Past Contributors 1179
About the Authors 1180

Tribute to Evi

Every field has an avatar who defines and embodies that space. For system administration, that person is Evi Nemeth.

This is the 5th edition of a book that Evi led as an author for almost three decades. Although Evi wasn't able to physically join us in writing this edition, she's with us in spirit and, in some cases, in the form of text and examples that have endured. We've gone to great efforts to maintain Evi's extraordinary style, candor, technical depth, and attention to detail.

An accomplished mathematician and cryptographer, Evi's professional days were spent (most recently) as a computer science professor at the University of Colorado at Boulder. How system administration came into being, and Evi's involvement in it, is detailed in the last chapter of this book, *A Brief History of System Administration*.

Throughout her career, Evi looked forward to retiring and sailing the world. In 2001, she did exactly that: she bought a sailboat *(Wonderland)* and set off on an adventure. Across the years, Evi kept us entertained with stories of amazing islands, cool new people, and other sailing escapades. We produced two editions of this book with Evi anchoring as close as possible to shoreline establishments so that she could camp on their Wi-Fi networks and upload chapter drafts.

Never one to decline an intriguing venture, Evi signed on in June 2013 as crew for the historic schooner *Nina* for a sail across the Tasman Sea. The *Nina* disappeared shortly thereafter in a bad storm, and we haven't heard from Evi since. She was living her dream.

Evi taught us much more than system administration. Even in her 70s, she ran circles around all of us. She was always the best at building a network, configuring

a server, debugging a kernel, splitting wood, frying chicken, baking a quiche, or quaffing an occasional glass of wine. With Evi by your side, anything was achievable.

It's impossible to encapsulate all of Evi's wisdom here, but these tenets have stuck with us:

- Be conservative in what you send and liberal in what you receive.[1]
- Be liberal in who you hire, but fire early.
- Don't use weasel words.
- Undergraduates are the secret superpower.
- You can never use too much red ink.
- You don't really understand something until you've implemented it.
- It's always time for sushi.
- Be willing to try something twice.
- Always use **sudo**.

We're sure some readers will write in to ask what, exactly, some of the guidance above really means. We've left that as an exercise for the reader, as Evi would have. You can hear her behind you now, saying "Try it yourself. See how it works."

Smooth sailing, Evi. We miss you.

1. This tenet is also known as Postel's Law, named in honor of Jon Postel, who served as Editor of the RFC series from 1969 until his death in 1998.

Preface

Modern technologists are masters at the art of searching Google for answers. If another system administrator has already encountered (and possibly solved) a problem, chances are you can find their write-up on the Internet. We applaud and encourage this open sharing of ideas and solutions.

If great information is already available on the Internet, why write another edition of this book? Here's how this book helps system administrators grow:

- We offer philosophy, guidance, and context for applying technology appropriately. As with the blind men and the elephant, it's important to understand any given problem space from a variety of angles. Valuable perspectives include background on adjacent disciplines such as security, compliance, DevOps, cloud computing, and software development life cycles.
- We take a hands-on approach. Our purpose is to summarize our collective perspective on system administration and to recommend approaches that stand the test of time. This book contains numerous war stories and a wealth of pragmatic advice.
- This is not a book about how to run UNIX or Linux at home, in your garage, or on your smartphone. Instead, we describe the management of production environments such as businesses, government offices, and universities. These environments have requirements that are different from (and far outstrip) those of a typical hobbyist.
- We teach you how to be a professional. Effective system administration requires both technical and "soft" skills. It also requires a sense of humor.

The organization of this book

This book is divided into four large chunks: Basic Administration, Networking, Storage, and Operations.

Basic Administration presents a broad overview of UNIX and Linux from a system administrator's perspective. The chapters in this section cover most of the facts and techniques needed to run a stand-alone system.

The Networking section describes the protocols used on UNIX systems and the techniques used to set up, extend, and maintain networks and Internet-facing servers. High-level network software is also covered here. Among the featured topics are the Domain Name System, electronic mail, single sign-on, and web hosting.

The Storage section tackles the challenges of storing and managing data. This section also covers subsystems that allow file sharing on a network, such as the Network File System and the Windows-friendly SMB protocol.

The Operations section addresses the key topics that a system administrator faces on a daily basis when managing production environments. These topics include monitoring, security, performance, interactions with developers, and the politics of running a system administration group.

Our contributors

We're delighted to welcome James Garnett, Fabrizio Branca, and Adrian Mouat as contributing authors for this edition. These contributors' deep knowledge of a variety of areas has greatly enriched the content of this book.

Contact information

Please send suggestions, comments, and bug reports to ulsah@book.admin.com. We do answer mail, but please be patient; it is sometimes a few days before one of us is able to respond. Because of the volume of email that this alias receives, we regret that we are unable to answer technical questions.

To view a copy of our current bug list and other late-breaking information, visit our web site, admin.com.

We hope you enjoy this book, and we wish you the best of luck with your adventures in system administration!

Garth Snyder
Trent R. Hein
Ben Whaley
Dan Mackin

July 2017

Foreword

In 1942, Winston Churchill described an early battle of WWII: "this is not the end—it is not even the beginning of the end—but it is, perhaps, the end of the beginning." I was reminded of these words when I was approached to write this Foreword for the fifth edition of *UNIX and Linux System Administration Handbook*. The loss at sea of Evi Nemeth has been a great sadness for the UNIX community, but I'm pleased to see her legacy endure in the form of this book and in her many contributions to the field of system administration.

The way the world got its Internet was, originally, through UNIX. A remarkable departure from the complex and proprietary operating systems of its day, UNIX was minimalistic, tools-driven, portable, and widely used by people who wanted to share their work with others. What we today call open source software was already pervasive—but nameless—in the early days of UNIX and the Internet. Open source was just how the technical and academic communities did things, because the benefits so obviously outweighed the costs.

Detailed histories of UNIX, Linux, and the Internet have been lovingly presented elsewhere. I bring up these high-level touchpoints only to remind us all that the modern world owes much to open source software and to the Internet, and that the original foundation for this bounty was UNIX.

As early UNIX and Internet companies fought to hire the most brilliant people and to deliver the most innovative features, software portability was often sacrificed. Eventually, system administrators had to know a little bit about a lot of things because no two UNIX-style operating systems (then, or now) were entirely alike. As a working UNIX system administrator in the mid-1980s and later, I had to know not just shell scripting and Sendmail configuration but also kernel device drivers. It was also important to know how to fix a filesystem with an octal debugger. Fun times!

Out of that era came the first edition of this book and all the editions that followed it. In the parlance of the times, we called the authors "Evi and crew" or perhaps "Evi and her kids." Because of my work on Cron and BIND, Evi spent a week or two with me (and my family, and my workplace) every time an edition of this book was in progress to make sure she was saying enough, saying nothing wrong, and hopefully, saying something unique and useful about each of those programs. Frankly, being around Evi was exhausting, especially when she was curious about something, or on a deadline, or in my case, both. That having been said, I miss Evi terribly and I treasure every memory and every photograph of her.

In the decades of this book's multiple editions, much has changed. It has been fascinating to watch this book evolve along with UNIX itself. Every new edition omitted some technologies that were no longer interesting or relevant to make room for new topics that were just becoming important to UNIX administrators, or that the authors thought soon would be.

It's hard to believe that we ever spent dozens of kilowatts of power on truck-sized computers whose capabilities are now dwarfed by an Android smartphone. It's equally hard to believe that we used to run hundreds or thousands of individual server and desktop computers with now-antiquated technologies like **rdist**. In those years, various editions of this book helped people like me (and like Evi herself) cope with heterogeneous and sometimes proprietary computers that were each *real* rather than virtualized, and which each had to be *maintained* rather than being reinstalled (or in Docker, rebuilt) every time something needed patching or upgrading.

We adapt, or we exit. The "Evi kids" who carry on Evi's legacy have adapted, and they are back in this fifth edition to tell you what you need to know about how modern UNIX and Linux computers work and how you can make them work the way you want them to. Evi's loss marks the end of an era, but it's also sobering to consider how many aspects of system administration have passed into history alongside her. I know dozens of smart and successful technologists who will never dress cables in the back of an equipment rack, hear the tone of a modem, or see an RS-232 cable. This edition is for those whose systems live in the cloud or in virtualized data centers; those whose administrative work largely takes the form of automation and configuration source code; those who collaborate closely with developers, network engineers, compliance officers, and all the other worker bees who inhabit the modern hive.

You hold in your hand the latest, best edition of a book whose birth and evolution have precisely tracked the birth and evolution of the UNIX and Internet community. Evi would be extremely proud of her kids, both because of this book, and because of who they have each turned out to be. I am proud to know them.

Paul Vixie
La Honda, California
June 2017

Acknowledgments

Many people contributed to this project, bestowing everything from technical reviews and constructive suggestions to overall moral support. The following individuals deserve special thanks for hanging in there with us:

- Jason Carolan
- Randy Else
- Steve Gaede
- Asif Khan
- Sam Leathers
- Ned McClain
- Beth McElroy
- Paul Nelson
- Tim O'Reilly
- Madhuri Peri
- Dave Roth
- Peter Sankauskas
- Deepak Singh
- Paul Vixie

Our editor at Pearson, Mark Taub, deserves huge thanks for his wisdom, patient support, and gentle author herding throughout the production of this book. It's safe to say this edition would not have come to fruition without him.

Mary Lou Nohr has been our relentless behind-the-scenes copy editor for over 20 years. When we started work on this edition, Mary Lou was headed for well-deserved retirement. After a lot of begging and guilt-throwing, she agreed to join us for an encore. (Both Mary Lou Nohr and Evi Nemeth appear on the cover. Can you find them?)

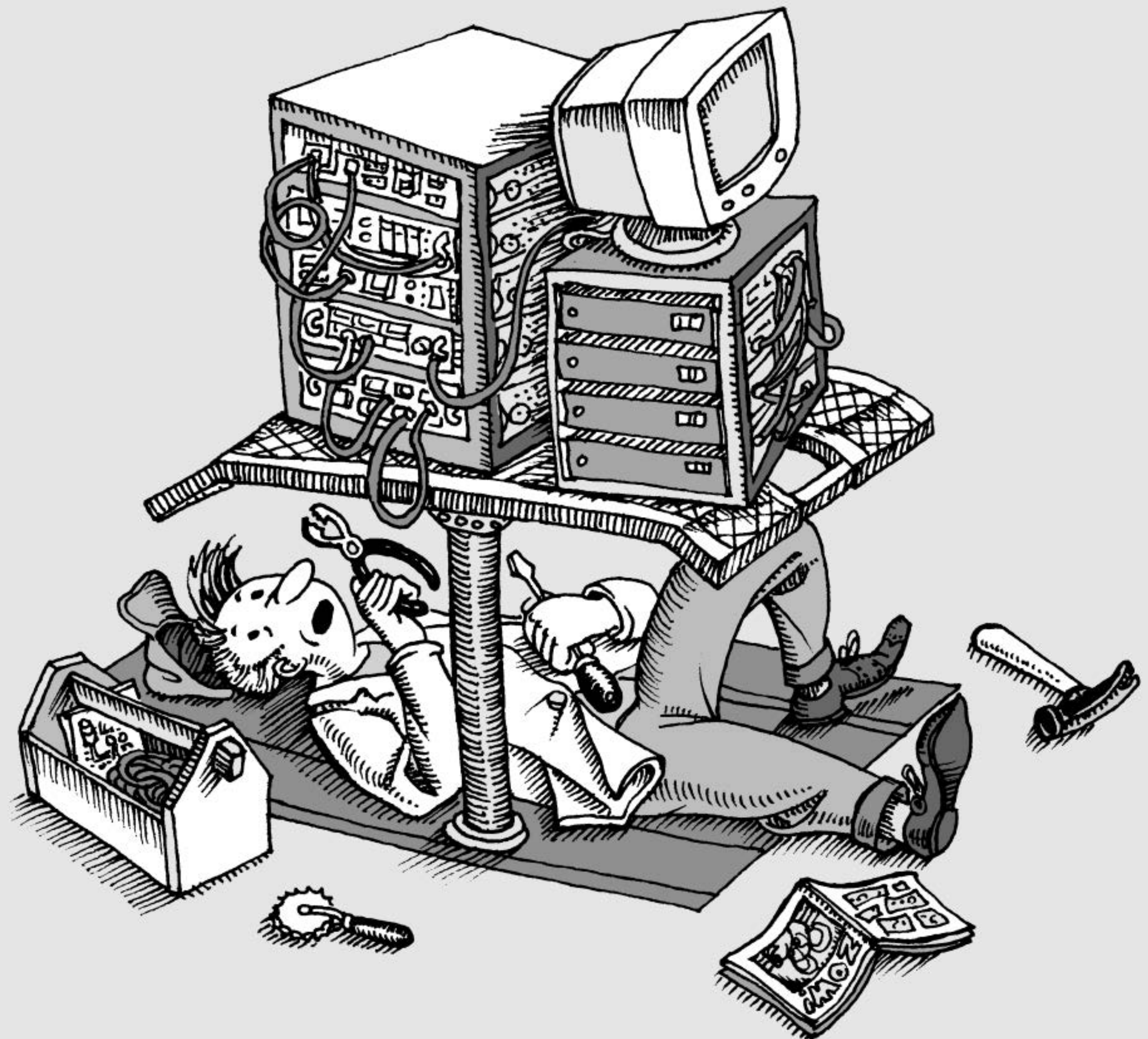
We've had a fantastic team of technical reviewers. Three dedicated souls reviewed the entire book: Jonathan Corbet, Pat Parseghian, and Jennine Townsend. We greatly appreciate their tenacity and tactfulness.

This edition's awesome cartoons and cover were conceived and executed by Lisa Haney. Her portfolio is on-line at lisahaney.com.

Last but not least, special thanks to Laszlo Nemeth for his willingness to support the continuation of this series.

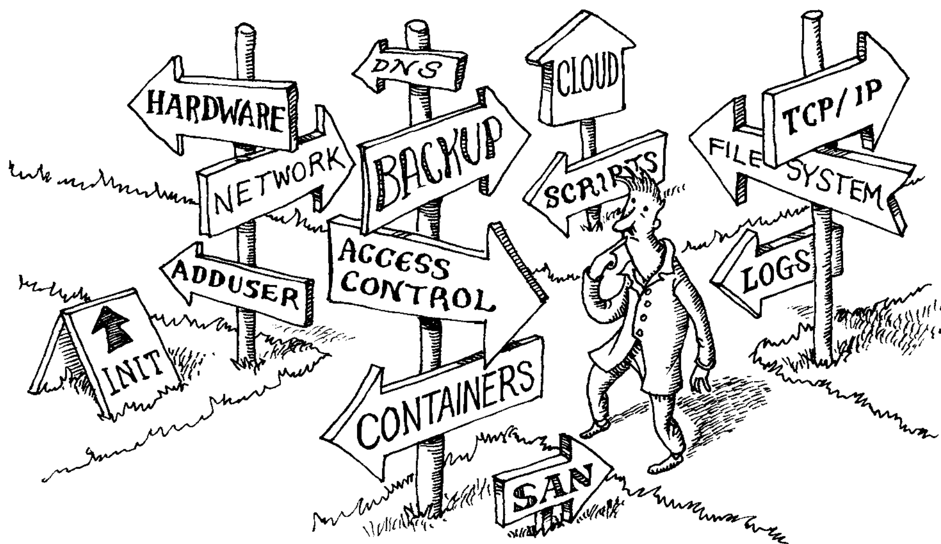
SECTION ONE

BASIC ADMINISTRATION



This page intentionally left blank

1 Where to Start



We've designed this book to occupy a specific niche in the vast ecosystem of man pages, blogs, magazines, books, and other reference materials that address the needs of UNIX and Linux system administrators.

First, it's an orientation guide. It reviews the major administrative systems, identifies the different pieces of each, and explains how they work together. In the many cases where you must choose among various implementations of a concept, we describe the advantages and drawbacks of the most popular options.

Second, it's a quick-reference handbook that summarizes what you need to know to perform common tasks on a variety of common UNIX and Linux systems. For example, the **ps** command, which shows the status of running processes, supports more than 80 command-line options on Linux systems. But a few combinations of options satisfy the majority of a system administrator's needs; we summarize them on page 98.

Finally, this book focuses on the administration of enterprise servers and networks. That is, *serious, professional* system administration. It's easy to set up a single system; harder to keep a distributed, cloud-based platform running smoothly in the face of viral popularity, network partitions, and targeted attacks. We describe techniques

and rules of thumb that help you recover systems from adversity, and we help you choose solutions that scale as your empire grows in size, complexity, and heterogeneity.

We don't claim to do all of this with perfect objectivity, but we think we've made our biases fairly clear throughout the text. One of the interesting things about system administration is that reasonable people can have dramatically different notions of what constitutes the most appropriate solution. We offer our subjective opinions to you as raw data. Decide for yourself how much to accept and how much of our comments apply to your environment.

1.1 Essential duties of a system administrator

The sections below summarize some of the main tasks that administrators are expected to perform. These duties need not necessarily be carried out by a single person, and at many sites the work is distributed among the members of a team. However, at least one person should understand all the components and ensure that every task is performed correctly.

Controlling access

See Chapters 8, 17, and 23 for information about user account provisioning.

The system administrator creates accounts for new users, removes the accounts of inactive users, and handles all the account-related issues that come up in between (e.g., forgotten passwords and lost key pairs). The process of actually adding and removing accounts is typically automated by a configuration management system or centralized directory service.

Adding hardware

Administrators who work with physical hardware (as opposed to cloud or hosted systems) must install it and configure it to be recognized by the operating system. Hardware support chores might range from the simple task of adding a network interface card to configuring a specialized external storage array.

Automating tasks

See Chapter 7, Scripting and the Shell, for information about scripting and automation.

Using tools to automate repetitive and time-consuming tasks increases your efficiency, reduces the likelihood of errors caused by humans, and improves your ability to respond rapidly to changing requirements. Administrators strive to reduce the amount of manual labor needed to keep systems functioning smoothly. Familiarity with scripting languages and automation tools is a large part of the job.

Overseeing backups

See page 788 for some tips on performing backups.

Backing up data and restoring it successfully when required are important administrative tasks. Although backups are time consuming and boring, the frequency of real-world disasters is simply too high to allow the job to be disregarded.

Operating systems and some individual software packages provide well-established tools and techniques to facilitate backups. Backups must be executed on a regular schedule and restores must be tested periodically to ensure that they are functioning correctly.

Installing and upgrading software

See Chapter 6 for information about software management.

Software must be selected, installed, and configured, often on a variety of operating systems. As patches and security updates are released, they must be tested, reviewed, and incorporated into the local environment without endangering the stability of production systems.

See Chapter 26 for information about software deployment and continuous delivery.

The term "software delivery" refers to the process of releasing updated versions of software—especially software developed in-house—to downstream users. "Continuous delivery" takes this process to the next level by automatically releasing software to users at a regular cadence as it is developed. Administrators help implement robust delivery processes that meet the requirements of the enterprise.

Monitoring

See Chapter 28 for information about monitoring.

Working around a problem is usually faster than taking the time to document and report it, and users internal to an organization often follow the path of least resistance. External users are more likely to voice their complaints publicly than to open a support inquiry. Administrators can help to prevent both of these outcomes by detecting problems and fixing them before public failures occur.

Some monitoring tasks include ensuring that web services respond quickly and correctly, collecting and analyzing log files, and keeping tabs on the availability of server resources such as disk space. All of these are excellent opportunities for automation, and a slew of open source and commercial monitoring systems can help sysadmins with these tasks.

Troubleshooting

See page 428 for an introduction to network troubleshooting.

Networked systems fail in unexpected and sometimes spectacular fashion. It's the administrator's job to play mechanic by diagnosing problems and calling in subject-matter experts as needed. Finding the source of a problem is often more challenging than resolving it.

Maintaining local documentation

See page 1115 for suggestions regarding documentation.

Administrators choose vendors, write scripts, deploy software, and make many other decisions that may not be immediately obvious or intuitive to others. Thorough and accurate documentation is a blessing for team members who would otherwise need to reverse-engineer a system to resolve problems in the middle of the night. A lovingly crafted network diagram is more useful than many paragraphs of text when describing a design.

Vigilantly monitoring security

See Chapter 27 for more information about security.

Administrators are the first line of defense for protecting network-attached systems. The administrator must implement a security policy and set up procedures to prevent systems from being breached. This responsibility might include only a few basic checks for unauthorized access, or it might involve an elaborate network of traps and auditing programs, depending on the context. System administrators are cautious by nature and are often the primary champions of security across a technical organization.

Tuning performance

See Chapter 29 for more information about performance.

UNIX and Linux are general purpose operating systems that are well suited to almost any conceivable computing task. Administrators can tailor systems for optimal performance in accord with the needs of users, the available infrastructure, and the services the systems provide. When a server is performing poorly, it is the administrator's job to investigate its operation and identify areas that need improvement.

Developing site policies

See the sections starting on page 17 for information about local policy-making.

For legal and compliance reasons, most sites need policies that govern the acceptable use of computer systems, the management and retention of data, the privacy and security of networks and systems, and other areas of regulatory interest. System administrators often help organizations develop sensible policies that meet the letter and intent of the law and yet still promote progress and productivity.

Working with vendors

Most sites rely on third parties to provide a variety of ancillary services and products related to their computing infrastructure. These providers might include software developers, cloud infrastructure providers, hosted software-as-a-service (SaaS) shops, help-desk support staff, consultants, contractors, security experts, and platform or infrastructure vendors. Administrators may be tasked with selecting vendors, assisting with contract negotiations, and implementing solutions once the paperwork has been completed.

Fire fighting

Although helping other people with their various problems is rarely included in a system administrator's job description, these tasks claim a measurable portion of most administrators' workdays. System administrators are bombarded with problems ranging from "It worked yesterday and now it doesn't! What did you change?" to "I spilled coffee on my keyboard! Should I pour water on it to wash it out?"

In most cases, your response to these issues affects your perceived value as an administrator far more than does any actual technical skill you might possess. You can either howl at the injustice of it all, or you can delight in the fact that a single

well-handled trouble ticket scores more brownie points than five hours of midnight debugging. Your choice!

1.2 Suggested background

We assume in this book that you have a certain amount of Linux or UNIX experience. In particular, you should have a general concept of how the system looks and feels from a user's perspective since we do not review that material. Several good books can get you up to speed; see *Recommended reading* on page 28.

We love well-designed graphical interfaces. Unfortunately, GUI tools for system administration on UNIX and Linux remain rudimentary in comparison with the richness of the underlying software. In the real world, administrators must be comfortable using the command line.

For text editing, we strongly recommend learning **vi** (now seen more commonly in its enhanced form, **vim**), which is standard on all systems. It is simple, powerful, and efficient. Mastering **vim** is perhaps the single best productivity enhancement available to administrators. Use the **vimtutor** command for an excellent, interactive introduction.

Alternatively, GNU's **nano** is a simple and low-impact "starter editor" that has on-screen prompts. Use it discreetly; professional administrators may be visibly distressed if they witness a peer running **nano**.

See Chapter 7 for an introduction to scripting.

Although administrators are not usually considered software developers, industry trends are blurring the lines between these functions. Capable administrators are usually polyglot programmers who don't mind picking up a new language when the need arises.

For new scripting projects, we recommend Bash (aka **bash**, aka **sh**), Ruby, or Python. Bash is the default command shell on most UNIX and Linux systems. It is primitive as a programming language, but it serves well as the duct tape in an administrative tool box. Python is a clever language with a highly readable syntax, a large developer community, and libraries that facilitate many common tasks. Ruby developers describe the language as "a joy to work with" and "beautiful to behold." Ruby and Python are similar in many ways, and we've found them to be equally functional for administration. The choice between them is mostly a matter of personal preference.

We also suggest that you learn **expect**, which is not a programming language so much as a front end for driving interactive programs. It's an efficient glue technology that can replace some complex scripting and is easy to learn.

Chapter 7, *Scripting and the Shell*, summarizes the most important things to know about scripting for Bash, Python, and Ruby. It also reviews regular expressions (text matching patterns) and some shell idioms that are useful for sysadmins.

1.3 Linux distributions

A Linux distribution comprises the Linux kernel, which is the core of the operating system, and packages that make up all the commands you can run on the system. All distributions share the same kernel lineage, but the format, type, and number of packages differ quite a bit. Distributions also vary in their focus, support, and popularity. There continue to be hundreds of independent Linux distributions, but our sense is that distributions derived from the Debian and Red Hat lineages will predominate in production environments in the years ahead.

By and large, the differences among Linux distributions are not cosmically significant. In fact, it is something of a mystery why so many different distributions exist, each claiming "easy installation" and "a massive software library" as its distinguishing features. It's hard to avoid the conclusion that people just like to make new Linux distributions.

Most major distributions include a relatively painless installation procedure, a desktop environment, and some form of package management. You can try them out easily by starting up a cloud instance or a local virtual machine.

See Chapter 25, Containers, for more information about Docker and containers.

Much of the insecurity of general-purpose operating systems derives from their complexity. Virtually all leading distributions are cluttered with scores of unused software packages; security vulnerabilities and administrative anguish often come along for the ride. In response, a relatively new breed of minimalist distributions has been gaining traction. CoreOS is leading the charge against the status quo and prefers to run all software in containers. Alpine Linux is a lightweight distribution that is used as the basis of many public Docker images. Given this reductionist trend, we expect the footprint of Linux to shrink over the coming years.

By adopting a distribution, you are making an investment in a particular vendor's way of doing things. Instead of looking only at the features of the installed software, it's wise to consider how your organization and that vendor are going to work with each other. Some important questions to ask are:

- Is this distribution going to be around in five years?
- Is this distribution going to stay on top of the latest security patches?
- Does this distribution have an active community and sufficient documentation?
- If I have problems, will the vendor talk to me, and how much will that cost?

Table 1.1 lists some of the most popular mainstream distributions.

The most viable distributions are not necessarily the most corporate. For example, we expect Debian Linux (OK, OK, Debian GNU/Linux!) to remain viable for a long time despite the fact that Debian is not a company, doesn't sell anything, and offers no enterprise-level support. Debian benefits from a committed group of contributors and from the enormous popularity of the Ubuntu distribution, which is based on it.

A comprehensive list of distributions, including many non-English distributions, can be found at lwn.net/Distributions or distrowatch.com.

Table 1.1 Most popular general-purpose Linux distributions

Distribution	Web site	Comments
Arch	archlinux.org	For those who fear not the command line
CentOS	centos.org	Free analog of Red Hat Enterprise
CoreOS	coreos.com	Containers, containers everywhere
Debian	debian.org	Free as in freedom, most GNUish distro
Fedora	fedoraproject.org	Test bed for Red Hat Linux
Kali	kali.org	For penetration testers
Linux Mint	linuxmint.com	Ubuntu-based, desktop-friendly
openSUSE	opensuse.org	Free analog of SUSE Linux Enterprise
openWRT	openwrt.org	Linux for routers and embedded devices
Oracle Linux	oracle.com	Oracle-supported version of RHEL
RancherOS	rancher.com	20MiB, everything in containers
Red Hat Enterprise	redhat.com	Reliable, slow-changing, commercial
Slackware	slackware.com	Grizzled, long-surviving distro
SUSE Linux Enterprise	suse.com	Strong in Europe, multilingual
Ubuntu	ubuntu.com	Cleaned-up version of Debian

1.4 Example systems used in this book

We have chosen three popular Linux distributions and one UNIX variant as our primary examples for this book: Debian GNU/Linux, Ubuntu Linux, Red Hat Enterprise Linux (and its dopplegänger CentOS), and FreeBSD. These systems are representative of the overall marketplace and account collectively for a substantial portion of installations in use at large sites today.

Information in this book generally applies to all of our example systems unless a specific attribution is given. Details particular to one system are marked with a logo:

Debian GNU/Linux 9.0 “Stretch”

Ubuntu® 17.04 “Zesty Zapus”

RHEL Red Hat® Enterprise Linux® 7.1 and CentOS® 7.1

FreeBSD® 11.0

Most of these marks belong to the vendors that release the corresponding software and are used with the kind permission of their respective owners. However, the vendors have not reviewed or endorsed the contents of this book.

We repeatedly attempted and failed to obtain permission from Red Hat to use their famous red fedora logo, so you're stuck with yet another technical acronym. At least this one is in the margins.

The paragraphs below provide a bit more detail about each of the example systems.

Example Linux distributions

Information that's specific to Linux but not to any particular distribution is marked with the Tux penguin logo shown at left.

Debian (pronounced *deb-ian*, named after the late founder Ian Murdock and his wife Debra), is one of the oldest and most well-regarded distributions. It is a non-commercial project with more than a thousand contributors worldwide. Debian maintains an ideological commitment to community development and open access, so there's never any question about which parts of the distribution are free or redistributable.

Debian defines three releases that are maintained simultaneously: stable, targeting production servers; unstable, with current packages that may have bugs and security vulnerabilities; and testing, which is somewhere in between.

Ubuntu is based on Debian and maintains Debian's commitment to free and open source software. The business behind Ubuntu is Canonical Ltd., founded by entrepreneur Mark Shuttleworth.

Canonical offers a variety of editions of Ubuntu targeting the cloud, the desktop, and bare metal. There are even releases intended for phones and tablets. Ubuntu version numbers derive from the year and month of release, so version 16.10 is from October, 2016. Each release also has an alliterative code name such as Vivid Vervet or Wily Werewolf.

Two versions of Ubuntu are released annually: one in April and one in October. The April releases in even-numbered years are long-term support (LTS) editions that promise five years of maintenance updates. These are the releases recommended for production use.

RHEL

Red Hat has been a dominant force in the Linux world for more than two decades, and its distributions are widely used in North America and beyond. By the numbers, Red Hat, Inc., is the most successful open source software company in the world.

Red Hat Enterprise Linux, often shortened to RHEL, targets production environments at large enterprises that require support and consulting services to keep their systems running smoothly. Somewhat paradoxically, RHEL is open source but requires a license. If you're not willing to pay for the license, you're not going to be running Red Hat.

Red Hat also sponsors Fedora, a community-based distribution that serves as an incubator for bleeding-edge software not considered stable enough for RHEL.

Fedora is used as the initial test bed for software and configurations that later find their way to RHEL.

CentOS is virtually identical to Red Hat Enterprise Linux, but free of charge. The CentOS Project (centos.org) is owned by Red Hat and employs its lead developers. However, they operate separately from the Red Hat Enterprise Linux team. The CentOS distribution lacks Red Hat's branding and a few proprietary tools, but is in other respects equivalent.

CentOS is an excellent choice for sites that want to deploy a production-oriented distribution without paying tithes to Red Hat. A hybrid approach is also feasible: front-line servers can run Red Hat Enterprise Linux and avail themselves of Red Hat's excellent support, even as nonproduction systems run CentOS. This arrangement covers the important bases in terms of risk and support while also minimizing cost and administrative complexity.

CentOS aspires to full binary and bug-for-bug compatibility with Red Hat Enterprise Linux. Rather than repeating "Red Hat and CentOS" ad nauseam, we generally mention only one or the other in this book. The text applies equally to Red Hat and CentOS unless we note otherwise.

Other popular distributions are also Red Hat descendants. Oracle sells a rebranded and customized version of CentOS to customers of its enterprise database software. Amazon Linux, available to Amazon Web Services users, was initially derived from CentOS and still shares many of its conventions.

Most administrators will encounter a Red Hat-like system at some point in their careers, and familiarity with its nuances is helpful even if it isn't the system of choice at your site.

Example UNIX distribution

The popularity of UNIX has been waning for some time, and most of the stalwart UNIX distributions (e.g., Solaris, HP-UX, and AIX) are no longer in common use. The open source descendants of BSD are exceptions to this trend and continue to enjoy a cult following, particularly among operating system experts, free software evangelists, and security-minded administrators. In other words, some of the world's foremost operating system authorities rely on the various BSD distributions. Apple's macOS has a BSD heritage.

FreeBSD, first released in late 1993, is the most widely used of the BSD derivatives. It commands a 70% market share among BSD variants according to some usage statistics. Users include major Internet companies such as WhatsApp, Google, and Netflix.

Unlike Linux, FreeBSD is a complete operating system, not just a kernel. Both the kernel and userland software are licensed under the permissive BSD License, a fact that encourages development by and additions from the business community.

1.5 Notation and typographical conventions

In this book, filenames, commands, and literal arguments to commands are shown in boldface. Placeholders (e.g., command arguments that should not be taken literally) are in italics. For example, in the command

```
cp file directory
```

you're supposed to replace *file* and *directory* with the names of an actual file and an actual directory.

Excerpts from configuration files and terminal sessions are shown in a code font. Sometimes, we annotate sessions with the **bash** comment character # and italic text. For example:

```
$ grep Bob /pub/phonelist  # Look up Bob's phone number
Bob Knowles 555-2834
Bob Smith 555-2311
```

We use $ to denote the shell prompt for a normal, unprivileged user, and # for the root user. When a command is specific to a distribution or family of distributions, we prefix the prompt with the distribution name. For example:

```
$ sudo su - root        # Become root
# passwd                # Change root's password
debian# dpkg -l         # List installed packages on Debian and Ubuntu
```

This convention is aligned with the one used by standard UNIX and Linux shells.

Outside of these specific cases, we have tried to keep special fonts and formatting conventions to a minimum as long as we could do so without compromising intelligibility. For example, we often talk about entities such as the daemon group with no special formatting at all.

We use the same conventions as the manual pages for command syntax:

- Anything between square brackets ("[" and "]") is optional.
- Anything followed by an ellipsis ("…") can be repeated.
- Curly braces ("{" and "}") mean that you should select one of the items separated by vertical bars ("|").

For example, the specification

```
bork [ -x ] { on | off } filename ...
```

would match any of the following commands:

```
bork on /etc/passwd
bork -x off /etc/passwd /etc/smartd.conf
bork off /usr/lib/tmac
```

We use shell-style globbing characters for pattern matching:

- A star (*) matches zero or more characters.
- A question mark (?) matches one character.
- A tilde or "twiddle" (~) means the home directory of the current user.
- ~*user* means the home directory of *user*.

For example, we might refer to the startup script directories **/etc/rc0.d**, **/etc/rc1.d**, and so on with the shorthand pattern **/etc/rc*.d**.

Text within quotation marks often has a precise technical meaning. In these cases, we ignore the normal rules of U.S. English and put punctuation outside the quotes so that there can be no confusion about what's included and what's not.

1.6 Units

Metric prefixes such as kilo-, mega-, and giga- are defined as powers of 10; one megabuck is $1,000,000. However, computer types have long poached these prefixes and used them to refer to powers of 2. For example, one "megabyte" of memory is really 2^{20} or 1,048,576 bytes. The stolen units have even made their way into formal standards such as the JEDEC Solid State Technology Association's Standard 100B.01, which recognizes the prefixes as denoting powers of 2 (albeit with some misgivings).

In an attempt to restore clarity, the International Electrotechnical Commission has defined a set of numeric prefixes (kibi-, mebi-, gibi-, and so on, abbreviated Ki, Mi, and Gi) based explicitly on powers of 2. Those units are always unambiguous, but they are just starting to be widely used. The original kilo-series prefixes are still used in both senses.

Context helps with decoding. RAM is always denominated in powers of 2, but network bandwidth is always a power of 10. Storage space is usually quoted in power-of-10 units, but block and page sizes are in fact powers of 2.

In this book, we use IEC units for powers of 2, metric units for powers of 10, and metric units for rough values and cases in which the exact basis is unclear, undocumented, or impossible to determine. In command output and in excerpts from configuration files, or where the delineation is not important, we leave the original values and unit designators. We abbreviate bit as b and byte as B. Table 1.2 on the next page shows some examples.

The abbreviation K, as in "8KB of RAM!", is not part of any standard. It's a computerese adaptation of the metric abbreviation k, for kilo-, and originally meant 1,024 as opposed to 1,000. But since the abbreviations for the larger metric prefixes are already upper case, the analogy doesn't scale. Later, people became confused about the distinction and started using K for factors of 1,000, too.

Most of the world doesn't consider this to be an important matter and, like the use of imperial units in the United States, metric prefixes are likely to be misused for

Table 1.2 Unit decoding examples

Example	Meaning
1kB file	A file that contains 1,000 bytes
4KiB SSD pages	SSD pages that contain 4,096 bytes
8KB of memory	Not used in this book; see note on page 13
100MB file size limit	Nominally 10^8 bytes; in context, ambiguous
100MB disk partition	Nominally 10^8 bytes; in context, probably 99,999,744 bytes [a]
1GiB of RAM	1,073,741,824 bytes of memory
1 Gb/s Ethernet	A network that transmits 1,000,000,000 bits per second
6TB hard disk	A hard disk that stores about 6,000,000,000,000 bytes

a. That is, 10^8 rounded down to the nearest whole multiple of the disk's 512-byte block size

the foreseeable future. Ubuntu maintains a helpful units policy, though we suspect it has not been widely adopted even at Canonical; see wiki.ubuntu.com/UnitsPolicy for some additional details.

1.7 Man pages and other on-line documentation

The manual pages, usually called "man pages" because they are read with the **man** command, constitute the traditional "on-line" documentation. (Of course, these days all documentation is on-line in some form or another.) Program-specific man pages come along for the ride when you install new software packages. Even in the age of Google, we continue to consult man pages as an authoritative resource because they are accessible from the command line, typically include complete details on a program's options, and show helpful examples and related commands.

Man pages are concise descriptions of individual commands, drivers, file formats, or library routines. They do not address more general topics such as "How do I install a new device?" or "Why is this system so damn slow?"

Organization of the man pages

FreeBSD and Linux divide the man pages into sections. Table 1.3 shows the basic schema. Other UNIX variants sometimes define the sections slightly differently.

The exact structure of the sections isn't important for most topics because **man** finds the appropriate page wherever it is stored. Just be aware of the section definitions when a topic with the same name appears in multiple sections. For example, **passwd** is both a command and a configuration file, so it has entries in both section 1 and section 5.

Table 1.3 Sections of the man pages

Section	Contents
1	User-level commands and applications
2	System calls and kernel error codes
3	Library calls
4	Device drivers and network protocols
5	Standard file formats
6	Games and demonstrations
7	Miscellaneous files and documents
8	System administration commands
9	Obscure kernel specs and interfaces

man: read man pages

See page 193 to learn about environment variables.

man *title* formats a specific manual page and sends it to your terminal through **more**, **less**, or whatever program is specified in your PAGER environment variable. *title* is usually a command, device, filename, or name of a library routine. The sections of the manual are searched in roughly numeric order, although sections that describe commands (sections 1 and 8) are usually searched first.

The form **man** *section title* gets you a man page from a particular section. Thus, on most systems, **man sync** gets you the man page for the **sync** command, and **man 2 sync** gets you the man page for the **sync** system call.

man -k *keyword* or **apropos** *keyword* prints a list of man pages that have *keyword* in their one-line synopses. For example:

```
$ man -k translate
objcopy (1)        - copy and translate object files
dcgettext (3)      - translate message
tr (1)             - translate or delete characters
snmptranslate (1)  - translate SNMP OID values into useful information
tr (1p)            - translate characters
...
```

The keywords database can become outdated. If you add additional man pages to your system, you may need to rebuild this file with **makewhatis** (Red Hat and FreeBSD) or **mandb** (Ubuntu).

Storage of man pages

nroff input for man pages (i.e., the man page source code) is stored in directories under **/usr/share/man** and compressed with **gzip** to save space. The **man** command knows how to decompress them on the fly.

man maintains a cache of formatted pages in **/var/cache/man** or **/usr/share/man** if the appropriate directories are writable; however, this is a security risk. Most systems preformat the man pages once at installation time (see **catman**) or not at all.

The **man** command can search several man page repositories to find the manual pages you request. On Linux systems, you can find out the current default search path with the **manpath** command. This path (from Ubuntu) is typical:

```
ubuntu$ manpath
/usr/local/man:/usr/local/share/man:/usr/share/man
```

If necessary, you can set your MANPATH environment variable to override the default path:

```
$ export MANPATH=/home/share/localman:/usr/share/man
```

Some systems let you set a custom system-wide default search path for man pages, which can be useful if you need to maintain a parallel tree of man pages such as those generated by OpenPKG. To distribute local documentation in the form of man pages, however, it is simpler to use your system's standard packaging mechanism and to put man pages in the standard man directories. See Chapter 6, *Software Installation and Management*, for more details.

1.8 Other authoritative documentation

Man pages are just a small part of the official documentation. Most of the rest, unfortunately, is scattered about on the web.

System-specific guides

Major vendors have their own dedicated documentation projects. Many continue to produce useful book-length manuals, including administration and installation guides. These are generally available on-line and as downloadable PDF files. Table 1.4 shows where to look.

Although this documentation is helpful, it's not the sort of thing you keep next to your bed for light evening reading (though some vendors' versions would make useful sleep aids). We generally Google for answers before turning to vendor docs.

Package-specific documentation

Most of the important software packages in the UNIX and Linux world are maintained by individuals or by third parties such as the Internet Systems Consortium and the Apache Software Foundation. These groups write their own documentation. The quality runs the gamut from embarrassing to spectacular, but jewels such as *Pro Git* from git-scm.com/book make the hunt worthwhile.

Table 1.4 Where to find OS vendors' proprietary documentation

OS	URL	Comments
Debian	debian.org/doc	Admin handbook lags behind the current version
Ubuntu	help.ubuntu.com	User oriented, see "server guide" for LTS releases
RHEL	redhat.com/docs	Comprehensive docs for administrators
CentOS	wiki.centos.org	Includes tips, HowTos, and FAQs
FreeBSD	freebsd.org/docs.html	See the *FreeBSD Handbook* for sysadmin info

Supplemental documents include white papers (technical reports), design rationales, and book- or pamphlet-length treatments of particular topics. These supplemental materials are not limited to describing just one command, so they can adopt a tutorial or procedural approach. Many pieces of software have both a man page and a long-form article. For example, the man page for **vim** tells you about the command-line arguments that **vim** understands, but you have to turn to an in-depth treatment to learn how to actually edit a file.

Most software projects have user and developer mailing lists and IRC channels. This is the first place to visit if you have questions about a specific configuration issue or if you encounter a bug.

Books

The O'Reilly books are favorites in the technology industry. The business began with *UNIX in a Nutshell* and now includes a separate volume on just about every important UNIX and Linux subsystem and command. O'Reilly also publishes books on network protocols, programming languages, Microsoft Windows, and other non-UNIX tech topics. All the books are reasonably priced, timely, and focused.

Many readers turn to O'Reilly's Safari Books Online, a subscription service that offers unlimited electronic access to books, videos, and other learning resources. Content from many publishers is included—not just O'Reilly—and you can choose from an immense library of material.

RFC publications

Request for Comments documents describe the protocols and procedures used on the Internet. Most of these are relatively detailed and technical, but some are written as overviews. The phrase "reference implementation" applied to software usually translates to "implemented by a trusted source according to the RFC specification."

RFCs are absolutely authoritative, and many are quite useful for system administrators. See page 376 for a more complete description of these documents. We refer to various RFCs throughout this book.

1.9 Other sources of information

The sources discussed in the previous section are peer reviewed and written by authoritative sources, but they're hardly the last word in UNIX and Linux administration. Countless blogs, discussion forums, and news feeds are available on the Internet.

It should go without saying, but Google is a system administrator's best friend. Unless you're looking up the details of a specific command or file format, Google or an equivalent search engine should be the first resource you consult for any sysadmin question. Make it a habit; if nothing else, you'll avoid the delay and humiliation of having your questions in an on-line forum answered with a link to Google.[1] *When stuck, search the web.*

Keeping current

Operating systems and the tools and techniques that support them change rapidly. Read the sites in Table 1.5 with your morning coffee to keep abreast of industry trends.

Table 1.5 Resources for keeping up to date

Web site	Description
darkreading.com	Security news, trends, and discussion
devopsreactions.tumblr.com	Sysadmin humor in animated GIF form
linux.com	A Linux Foundation site; forum, good for new users
linuxfoundation.org	Nonprofit fostering OSS, employer of Linus Torvalds
lwn.net	High-quality, timely articles on Linux and OSS
lxer.com	Linux news aggregator
securityfocus.com	Vulnerability reports and security-related mailing lists
@SwiftOnSecurity	Infosec opinion from Taylor Swift (parody account)
@nixcraft	Tweets about UNIX and Linux administration
everythingsysadmin.com	Blog of Thomas Limoncelli, respected sysadmin[a]
sysadvent.blogspot.com	Advent for sysadmins with articles each December
oreilly.com/topics	Learning resources from O'Reilly on many topics
schneier.com	Blog of Bruce Schneier, privacy and security expert

a. See also Tom's collection of April Fools' Day RFCs at rfc-humor.com

Social media are also useful. Twitter and reddit in particular have strong, engaged communities with a lot to offer, though the signal-to-noise ratio can sometimes be quite bad. On reddit, join the sysadmin, linux, linuxadmin, and netsec subreddits.

1. Or worse yet, a link to Google through lmgtfy.com

HowTos and reference sites

The sites listed in Table 1.6 contain guides, tutorials, and articles about how to accomplish specific tasks on UNIX and Linux.

Table 1.6 Task-specific forums and reference sites

Web site	Description
wiki.archlinux.org	Articles and guides for Arch Linux; many are more general
askubuntu.com	Q&A for Ubuntu users and developers
digitalocean.com	Tutorials on many OSS, development, and sysadmin topics [a]
kernel.org	Official Linux kernel site
serverfault.com	Collaboratively edited database of sysadmin questions [b]
serversforhackers.com	High-quality videos, forums, and articles on administration

a. See digitalocean.com/community/tutorials
b. Also see the sister site stackoverflow.com, which is dedicated to programming but useful for sysadmins

Stack Overflow and Server Fault, both listed in Table 1.6 (and both members of the Stack Exchange group of sites), warrant a closer look. If you're having a problem, chances are that somebody else has already seen it and asked for help on one of these sites. The reputation-based Q&A format used by the Stack Exchange sites has proved well suited to the kinds of problems that sysadmins and programmers encounter. It's worth creating an account and joining this large community.

Conferences

Industry conferences are a great way to network with other professionals, keep tabs on technology trends, take training classes, gain certifications, and learn about the latest services and products. The number of conferences pertinent to administration has exploded in recent years. Table 1.7 on the next page highlights some of the most prominent ones.

Meetups (meetup.com) are another way to network and engage with like-minded people. Most urban areas in the United States and around the world have a Linux user group or DevOps meetup that sponsors speakers, discussions, and hack days.

1.10 Ways to find and install software

Chapter 6, *Software Installation and Management,* addresses software provisioning in detail. But for the impatient, here's a quick primer on how to find out what's installed on your system and how to obtain and install new software.

Modern operating systems divide their contents into packages that can be installed independently of one another. The default installation includes a range of starter packages that you can expand and contract according to your needs. When adding

Table 1.7 Conferences relevant to system administrators

Conference	Location	When	Description
LISA	Varies	Q4	Large Installation System Administration
Monitorama	Portland	June	Monitoring tools and techniques
OSCON	Varies (US/EU)	Q2 or Q3	Long-running O'Reilly OSS conference
SCALE	Pasadena	Jan	Southern California Linux Expo
DefCon	Las Vegas	July	Oldest and largest hacker convention
Velocity	Global	Varies	O'Reilly conference on web operations
BSDCan	Ottawa	May/June	Everything BSD from novices to gurus
re:Invent	Las Vegas	Q4	AWS cloud computing conference
VMWorld	Varies (US/EU)	Q3 or Q4	Virtualization and cloud computing
LinuxCon	Global	Varies	The future of Linux
RSA	San Francisco	Q1 or Q2	Enterprise cryptography and infosec
DevOpsDays	Global	Varies	A range of topics on bridging the gap between development and ops teams
QCon	Global	Varies	A conference for software developers

software, don your security hat and remember that additional software creates additional attack surface. Only install what's necessary.

Add-on software is often provided in the form of precompiled packages as well, although the degree to which this is a mainstream approach varies widely among systems. Most software is developed by independent groups that release the software in the form of source code. Package repositories then pick up the source code, compile it appropriately for the conventions in use on the systems they serve, and package the resulting binaries. It's usually easier to install a system-specific binary package than to fetch and compile the original source code. However, packagers are sometimes a release or two behind the current version.

The fact that two systems use the same package format doesn't necessarily mean that packages for the two systems are interchangeable. Red Hat and SUSE both use RPM, for example, but their filesystem layouts are somewhat different. It's best to use packages designed for your particular system if they are available.

Our example systems provide excellent package management systems that include tools for accessing and searching hosted software repositories. Distributors aggressively maintain these repositories on behalf of the community, to facilitate patching and software updates. Life is good.

When the packaged format is insufficient, administrators must install software the old-fashioned way: by downloading a **tar** archive of the source code and manually configuring, compiling, and installing it. Depending on the software and the operating system, this process can range from trivial to nightmarish.

In this book, we generally assume that optional software is already installed rather than torturing you with boilerplate instructions for installing every package. If there's a potential for confusion, we sometimes mention the exact names of the packages needed to complete a particular project. For the most part, however, we don't repeat installation instructions since they tend to be similar from one package to the next.

Determining if software is already installed

For a variety of reasons, it can be a bit tricky to determine which package contains the software you actually need. Rather than starting at the package level, it's easier to use the shell's **which** command to find out if a relevant binary is already in your search path. For example, the following command reveals that the GNU C compiler has already been installed on this machine:

```
ubuntu$ which gcc
/usr/bin/gcc
```

If **which** can't find the command you're looking for, try **whereis**; it searches a broader range of system directories and is independent of your shell's search path.

Another alternative is the incredibly useful **locate** command, which consults a precompiled index of the filesystem to locate filenames that match a particular pattern.

FreeBSD includes **locate** as part of the base system. In Linux, the current implementation of **locate** is in the **mlocate** package. On Red Hat and CentOS, install the **mlocate** package with **yum**; see page 174.

locate can find any type of file; it is not specific to commands or packages. For example, if you weren't sure where to find the **signal.h** include file, you could try

```
freebsd$ locate signal.h
/usr/include/machine/signal.h
/usr/include/signal.h
/usr/include/sys/signal.h
...
```

locate's database is updated periodically by the **updatedb** command (in FreeBSD, **locate.updatedb**), which runs periodically out of **cron**. Therefore, the results of a **locate** don't always reflect recent changes to the filesystem.

See Chapter 6 for more information about package management.

If you know the name of the package you're looking for, you can also use your system's packaging utilities to check directly for the package's presence. For example, on a Red Hat system, the following command checks for the presence (and installed version) of the Python interpreter:

```
redhat$ rpm -q python
python-2.7.5-18.el7_1.1.x86_64
```

You can also find out which package a particular file belongs to:

```
redhat$ rpm -qf /etc/httpd
httpd-2.4.6-31.el7.centos.x86_64

freebsd$ pkg which /usr/local/sbin/httpd
/usr/local/sbin/httpd was installed by package apache24-2.4.12

ubuntu$ dpkg-query -S /etc/apache2
apache2: /etc/apache2
```

Adding new software

If you do need to install additional software, you first need to determine the canonical name of the relevant software package. For example, you'd need to translate "I want to install **locate**" to "I need to install the **mlocate** package," or translate "I need **named**" to "I have to install BIND." A variety of system-specific indexes on the web can help with this, but Google is usually just as effective. For example, a search for "locate command" takes you directly to several relevant discussions.

The following examples show the installation of the **tcpdump** command on each of our example systems. **tcpdump** is a packet capture tool that lets you view the raw packets being sent to and from the system on the network.

Debian and Ubuntu use APT, the Debian Advanced Package Tool:

```
ubuntu# sudo apt-get install tcpdump
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  tcpdump
0 upgraded, 1 newly installed, 0 to remove and 81 not upgraded.
Need to get 0 B/360 kB of archives.
After this operation, 1,179 kB of additional disk space will be used.
Selecting previously unselected package tcpdump.
(Reading database ... 63846 files and directories currently installed.)
Preparing to unpack .../tcpdump_4.6.2-4ubuntu1_amd64.deb ...
Unpacking tcpdump (4.6.2-4ubuntu1) ...
Processing triggers for man-db (2.7.0.2-5) ...
Setting up tcpdump (4.6.2-4ubuntu1) ...
```

The Red Hat and CentOS version is

```
redhat# sudo yum install tcpdump
Loaded plugins: fastestmirror
Determining fastest mirrors
 * base: mirrors.xmission.com
 * epel: linux.mirrors.es.net
 * extras: centos.arvixe.com
 * updates: repos.lax.quadranet.com
```

```
Resolving Dependencies
--> Running transaction check
---> Package tcpdump.x86_64 14:4.5.1-2.el7 will be installed
--> Finished Dependency Resolution
tcpdump-4.5.1-2.el7.x86_64.rpm                    | 387 kB  00:00
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
  Installing : 14:tcpdump-4.5.1-2.el7.x86_64   1/1
  Verifying  : 14:tcpdump-4.5.1-2.el7.x86_64   1/1
Installed:
  tcpdump.x86_64 14:4.5.1-2.el7
Complete!
```

The package manager for FreeBSD is **pkg**.

```
freebsd# sudo pkg install -y tcpdump
Updating FreeBSD repository catalogue...
Fetching meta.txz:            100%    944 B   0.9kB/s    00:01
Fetching packagesite.txz:     100%    5 MiB   5.5MB/s    00:01
Processing entries: 100%
FreeBSD repository update completed. 24632 packages processed.
All repositories are up-to-date.
The following 2 package(s) will be affected (of 0 checked):

New packages to be INSTALLED:
     tcpdump: 4.7.4
     libsmi: 0.4.8_1

The process will require 17 MiB more space.
2 MiB to be downloaded.
Fetching tcpdump-4.7.4.txz:   100%  301 KiB 307.7kB/s    00:01
Fetching libsmi-0.4.8_1.txz:  100%    2 MiB   2.0MB/s    00:01
Checking integrity... done (0 conflicting)
[1/2] Installing libsmi-0.4.8_1...
[1/2] Extracting libsmi-0.4.8_1: 100%
[2/2] Installing tcpdump-4.7.4...
[2/2] Extracting tcpdump-4.7.4: 100%
```

Building software from source code

As an illustration, here's how you build a version of **tcpdump** from the source code.

The first chore is to identify the code. Software maintainers sometimes keep an index of releases on the project's web site that are downloadable as tarballs. For open source projects, you're most likely to find the code in a Git repository.

The **tcpdump** source is kept on GitHub. Clone the repository in the **/tmp** directory, create a branch of the tagged version you want to build, then unpack, configure, build, and install it:

```
redhat$ cd /tmp
redhat$ git clone https://github.com/the-tcpdump-group/tcpdump.git
<status messages as repository is cloned>
redhat$ cd tcpdump
redhat$ git checkout tags/tcpdump-4.7.4 -b tcpdump-4.7.4
Switched to a new branch 'tcpdump-4.7.4'
redhat$ ./configure
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for gcc... gcc
checking whether the C compiler works... yes
...
redhat$ make
<several pages of compilation output>
redhat$ sudo make install
<files are moved in to place>
```

This **configure**/**make**/**make install** sequence is common to most software written in C and works on all UNIX and Linux systems. It's always a good idea to check the package's **INSTALL** or **README** file for specifics. You must have the development environment and any package-specific prerequisites installed. (In the case of **tcpdump**, **libpcap** and its libraries are prerequisites.)

You'll often need to tweak the build configuration, so use **./configure --help** to see the options available for each particular package. Another useful **configure** option is **--prefix=***directory*, which lets you compile the software for installation somewhere other than **/usr/local**, which is usually the default.

Installing from a web script

Cross-platform software bundles increasingly offer an expedited installation process that's driven by a shell script you download from the web with **curl**, **fetch**, or **wget**.[2] For example, to set up a machine as a Salt client, you can run the following commands:

```
$ curl -o /tmp/saltboot -sL https://bootstrap.saltstack.com
$ sudo sh /tmp/saltboot
```

The bootstrap script investigates the local environment, then downloads, installs, and configures an appropriate version of the software. This type of installation is particularly common in cases where the process itself is somewhat complex, but the vendor is highly motivated to make things easy for users. (Another good example is RVM; see page 232.)

2. These are all simple HTTP clients that download the contents of a URL to a local file or, optionally, print the contents to their standard output.

See Chapter 6 for more information about package installation.

This installation method is perfectly fine, but it raises a couple of issues that are worth mentioning. To begin with, it leaves no proper record of the installation for future reference. If your operating system offers a packagized version of the software, it's usually preferable to install the package instead of running a web installer. Packages are easy to track, upgrade, and remove. (On the other hand, most OS-level packages are out of date. You probably won't end up with the most current version of the software.)

See page 1007 for details on HTTPS's chain of trust.

Be very suspicious if the URL of the boot script is not secure (that is, it does not start with https:). Unsecured HTTP is trivial to hijack, and installation URLs are of particular interest to hackers because they know you're likely to run, as root, whatever code comes back. By contrast, HTTPS validates the identity of the server through a cryptographic chain of trust. Not foolproof, but reliable enough.

A few vendors publicize an HTTP installation URL that automatically redirects to an HTTPS version. This is dumb and is in fact no more secure than straight-up HTTP. There's nothing to prevent the initial HTTP exchange from being intercepted, so you might never reach the vendor's redirect. However, the existence of such redirects does mean it's worth trying your own substitution of https for http in insecure URLs. More often than not, it works just fine.

The shell accepts script text on its standard input, and this feature enables tidy, one-line installation procedures such as the following:

```
$ curl -L https://badvendor.com | sudo sh
```

However, there's a potential issue with this construction in that the root shell still runs even if **curl** outputs a partial script and then fails—say, because of a transient network glitch. The end result is unpredictable and potentially not good.

We are not aware of any documented cases of problems attributable to this cause. Nevertheless, it is a plausible failure mode. More to the point, piping the output of **curl** to a shell has entered the collective sysadmin unconscious as a prototypical rookie blunder, so if you must do it, at least keep it on the sly.

The fix is easy: just save the script to a temporary file, then run the script in a separate step after the download successfully completes.

1.11 Where to host

Operating systems and software can be hosted in private data centers, at co-location facilities, on a cloud platform, or on some combination of these. Most burgeoning startups choose the cloud. Established enterprises are likely to have existing data centers and may run a private cloud internally.

The most practical choice, and our recommendation for new projects, is a public cloud provider. These facilities offer numerous advantages over data centers:

- No capital expenses and low initial operating costs
- No need to install, secure, and manage hardware
- On-demand adjustment of storage, bandwidth, and compute capacity
- Ready-made solutions for common ancillary needs such as databases, load balancers, queues, monitoring, and more
- Cheaper and simpler implementation of highly available/redundant systems

Early cloud systems acquired a reputation for inferior security and performance, but these are no longer major concerns. These days, most of our administration work is in the cloud. See Chapter 9 for a general introduction to this space.

Our preferred cloud platform is the leader in the space: Amazon Web Services (AWS). Gartner, a leading technology research firm, found that AWS is ten times the size of all competitors combined. AWS innovates rapidly and offers a much broader array of services than does any other provider. It also has a reputation for excellent customer service and supports a large and engaged community. AWS offers a free service tier to cut your teeth on, including a year's use of a low powered cloud server.

Google Cloud Platform (GCP) is aggressively improving and marketing its products. Some claim that its technology is unmatched by other providers. GCP's growth has been slow, in part due to Google's reputation for dropping support for popular offerings. However, its customer-friendly pricing terms and unique features are appealing differentiators.

DigitalOcean is a simpler service with a stated goal of high performance. Its target market is developers, whom it woos with a clean API, low pricing, and extremely fast boot times. DigitalOcean is a strong proponent of open source software, and their tutorials and guides for popular Internet technologies are some of the best available.

1.12 Specialization and adjacent disciplines

System administrators do not exist in a vacuum; a team of experts is required to build and maintain a complex network. This section describes some of the roles with which system administrators overlap in skills and scope. Some administrators choose to specialize in one or more of these areas.

Your goal as a system administrator, or as a professional working in any of these related areas, is to achieve the objectives of the organization. Avoid letting politics or hierarchy interfere with progress. The best administrators solve problems and share information freely with others.

DevOps

See page 1106 for more comments on DevOps.

DevOps is not so much a specific function as a culture or operational philosophy. It aims to improve the efficiency of building and delivering software, especially at

large sites that have many interrelated services and teams. Organizations with a DevOps practice promote integration among engineering teams and may draw little or no distinction between development and operations. Experts who work in this area seek out inefficient processes and replace them with small shell scripts or large and unwieldy Chef repositories.

Site reliability engineers

Site reliability engineers value uptime and correctness above all else. Monitoring networks, deploying production software, taking pager duty, planning future expansion, and debugging outages all lie within the realm of these availability crusaders. Single points of failure are site reliability engineers' nemeses.

Security operations engineers

Security operations engineers focus on the practical, day-to-day side of an information security program. These folks install and operate tools that search for vulnerabilities and monitor for attacks on the network. They also participate in attack simulations to gauge the effectiveness of their prevention and detection techniques.

Network administrators

Network administrators design, install, configure, and operate networks. Sites that operate data centers are most likely to employ network administrators; that's because these facilities have a variety of physical switches, routers, firewalls, and other devices that need management. Cloud platforms also offer a variety of networking options, but these usually don't require a dedicated administrator because most of the work is handled by the provider.

Database administrators

Database administrators (sometimes known as DBAs) are experts at installing and managing database software. They manage database schemas, perform installations and upgrades, configure clustering, tune settings for optimal performance, and help users formulate efficient queries. DBAs are usually wizards with one or more query languages and have experience with both relational and nonrelational (NoSQL) databases.

Network operations center (NOC) engineers

NOC engineers monitor the real-time health of large sites and track incidents and outages. They troubleshoot tickets from users, perform routine upgrades, and coordinate actions among other teams. They can most often be found watching a wall of monitors that show graphs and measurements.

Data center technicians

Data center technicians work with hardware. They receive new equipment, track equipment inventory and life cycles, install servers in racks, run cabling, maintain power and air conditioning, and handle the daily operations of a data center. As a system administrator, it's in your best interest to befriend data center technicians and bribe them with coffee, caffeinated soft drinks, and alcoholic beverages.

Architects

Systems architects have deep expertise in more than one area. They use their experience to design distributed systems. Their job descriptions may include defining security zones and segmentation, eliminating single points of failure, planning for future growth, ensuring connectivity among multiple networks and third parties, and other site-wide decision making. Good architects are technically proficient and generally prefer to implement and test their own designs.

1.13 Recommended reading

Abbott, Martin L., and Michael T. Fisher. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise (2nd Edition)*. Addison-Wesley Professional, 2015.

Gancarz, Mike. *Linux and the Unix Philosophy*. Boston: Digital Press, 2003.

Limoncelli, Thomas A., and Peter Salus. *The Complete April Fools' Day RFCs*. Peer-to-Peer Communications LLC. 2007. Engineering humor. You can read this collection on-line for free at rfc-humor.com.

Raymond, Eric S. *The Cathedral & The Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol, CA: O'Reilly Media, 2001.

Salus, Peter H. *The Daemon, the GNU & the Penguin: How Free and Open Software is Changing the World*. Reed Media Services, 2008. This fascinating history of the open source movement by UNIX's best-known historian is also available at groklaw.com under the Creative Commons license. The URL for the book itself is quite long; look for a current link at groklaw.com or try this compressed equivalent: tinyurl.com/d6u7j.

Siever, Ellen, Stephen Figgins, Robert Love, and Arnold Robbins. *Linux in a Nutshell (6th Edition)*. Sebastopol, CA: O'Reilly Media, 2009.

System administration and DevOps

Kim, Gene, Kevin Behr, and George Spafford. *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win.* Portland, OR: IT Revolution Press, 2014. A guide to the philosophy and mindset needed to run a modern IT organization, written as a narrative. An instant classic.

Kim, Gene, Jez Humble, Patrick Debois, and John Willis. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations.* Portland, OR: IT Revolution Press, 2016.

Limoncelli, Thomas A., Christina J. Hogan, and Strata R. Chalup. *The Practice of System and Network Administration (2nd Edition).* Reading, MA: Addison-Wesley, 2008. This is a good book with particularly strong coverage of the policy and procedural aspects of system administration. The authors maintain a system administration blog at everythingsysadmin.com.

Limoncelli, Thomas A., Christina J. Hogan, and Strata R. Chalup. *The Practice of Cloud System Administration.* Reading, MA: Addison-Wesley, 2014. From the same authors as the previous title, now with a focus on distributed systems and cloud computing.

Essential tools

Blum, Richard, and Christine Bresnahan. *Linux Command Line and Shell Scripting Bible (3rd Edition).* Wiley, 2015.

Dougherty, Dale, and Arnold Robins. *Sed & Awk (2nd Edition).* Sebastopol, CA: O'Reilly Media, 1997. Classic O'Reilly book on the powerful, indispensable text processors **sed** and **awk**.

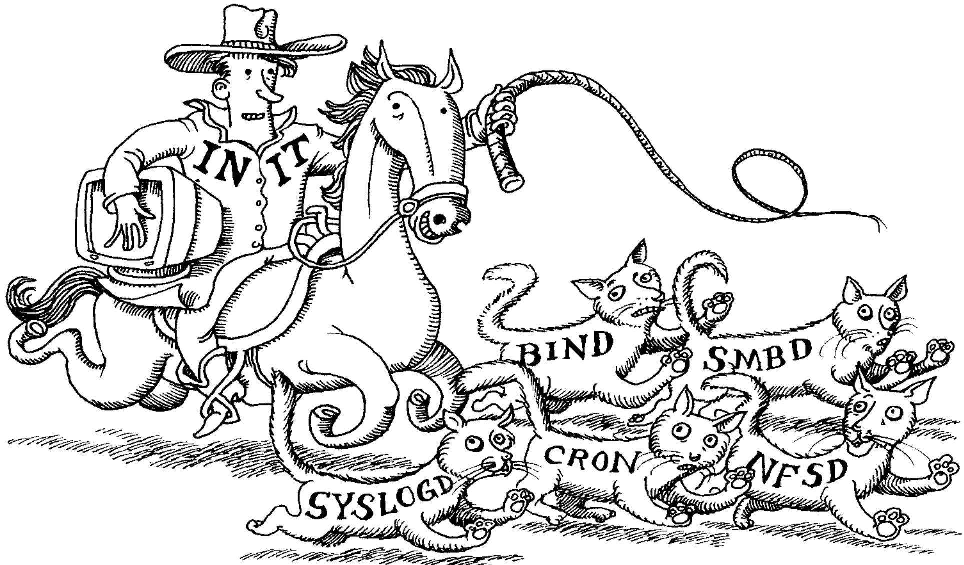
Kim, Peter. *The Hacker Playbook 2: Practical Guide To Penetration Testing.* CreateSpace Independent Publishing Platform, 2015.

Neil, Drew. *Practical Vim: Edit Text at the Speed of Thought*. Pragmatic Bookshelf, 2012.

Shotts, William E. *The Linux Command Line: A Complete Introduction.* San Francisco, CA: No Starch Press, 2012.

Sweigart, Al. *Automate the Boring Stuff with Python: Practical Programming for Total Beginners.* San Francisco, CA: No Starch Press, 2015.

2 Booting and System Management Daemons



"Booting" is the standard term for "starting up a computer." It's a shortened form of the word "bootstrapping," which derives from the notion that the computer has to "pull itself up by its own bootstraps."

The boot process consists of a few broadly defined tasks:

- Finding, loading, and running bootstrapping code
- Finding, loading, and running the OS kernel
- Running startup scripts and system daemons
- Maintaining process hygiene and managing system state transitions

The activities included in that last bullet point continue as long as the system remains up, so the line between bootstrapping and normal operation is inherently a bit blurry.

2.1 Boot process overview

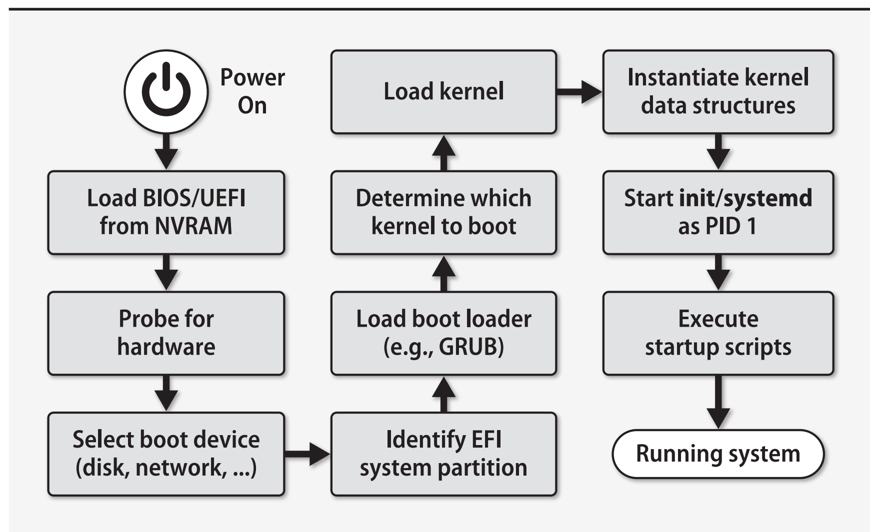
Startup procedures have changed a lot in recent years. The advent of modern (UEFI) BIOSs has simplified the early stages of booting, at least from a conceptual standpoint. In later stages, most Linux distributions now use a system manager daemon called **systemd** instead of the traditional UNIX **init**. **systemd** streamlines the boot

process by adding dependency management, support for concurrent startup processes, and a comprehensive approach to logging, among other features.

Boot management has also changed as systems have migrated into the cloud. The drift toward virtualization, cloud instances, and containerization has reduced the need for administrators to touch physical hardware. Instead, we now have image management, APIs, and control panels.

During bootstrapping, the kernel is loaded into memory and begins to execute. A variety of initialization tasks are performed, and the system is then made available to users. The general overview of this process is shown in Exhibit A.

Exhibit A Linux & UNIX boot process



Administrators have little direct, interactive control over most of the steps required to boot a system. Instead, admins can modify bootstrap configurations by editing config files for the system startup scripts or by changing the arguments the boot loader passes to the kernel.

Before the system is fully booted, filesystems must be checked and mounted and system daemons started. These procedures are managed by a series of shell scripts (sometimes called "**init** scripts") or unit files that are run in sequence by **init** or parsed by **systemd**. The exact layout of the startup scripts and the manner in which they are executed varies among systems. We cover the details later in this chapter.

2.2 System firmware

When a machine is powered on, the CPU is hardwired to execute boot code stored in ROM. On virtualized systems, this “ROM” may be imaginary, but the concept remains the same.

The system firmware typically knows about all the devices that live on the motherboard, such as SATA controllers, network interfaces, USB controllers, and sensors for power and temperature.[1] In addition to allowing hardware-level configuration of these devices, the firmware lets you either expose them to the operating system or disable and hide them.

On physical (as opposed to virtualized) hardware, most firmware offers a user interface. However, it’s generally crude and a bit tricky to access. You need control of the computer and console, and must press a particular key immediately after powering on the system. Unfortunately, the identity of the magic key varies by manufacturer; see if you can glimpse a cryptic line of instructions at the instant the system first powers on.[2] Barring that, try Delete, Control, F6, F8, F10, or F11. For the best chance of success, tap the key several times, then hold it down.

During normal bootstrapping, the system firmware probes for hardware and disks, runs a simple set of health checks, and then looks for the next stage of bootstrapping code. The firmware UI lets you designate a boot device, usually by prioritizing a list of available options (e.g., “try to boot from the DVD drive, then a USB drive, then a hard disk”).

In most cases, the system’s disk drives populate a secondary priority list. To boot from a particular drive, you must both set it as the highest-priority disk and make sure that “hard disk” is enabled as a boot medium.

BIOS vs. UEFI

Traditional PC firmware was called the BIOS, for Basic Input/Output System. Over the last decade, however, BIOS has been supplanted by a more formalized and modern standard, the Unified Extensible Firmware Interface (UEFI). You’ll often see UEFI referred to as “UEFI BIOS,” but for clarity, we’ll reserve the term BIOS for the legacy standard in this chapter. Most systems that implement UEFI can fall back to a legacy BIOS implementation if the operating system they’re booting doesn’t support UEFI.

UEFI is the current revision of an earlier standard, EFI. References to the name EFI persist in some older documentation and even in some standard terms, such as “EFI system partition.” In all but the most technically explicit situations, you can treat these terms as equivalent.

1. Virtual systems pretend to have this same set of devices.
2. You might find it helpful to disable the monitor’s power management features temporarily.

UEFI support is pretty much universal on new PC hardware these days, but plenty of BIOS systems remain in the field. Moreover, virtualized environments often adopt BIOS as their underlying boot mechanism, so the BIOS world isn't in danger of extinction just yet.

As much as we'd prefer to ignore BIOS and just talk about UEFI, it's likely that you'll encounter both types of systems for years to come. UEFI also builds-in several accommodations to the old BIOS regime, so a working knowledge of BIOS can be quite helpful for deciphering the UEFI documentation.

Legacy BIOS

Partitioning is a way to subdivide physical disks. See page 742 for a more detailed discussion.

Traditional BIOS assumes that the boot device starts with a record called the MBR (Master Boot Record). The MBR includes both a first-stage boot loader (aka "boot block") and a primitive disk partitioning table. The amount of space available for the boot loader is so small (less than 512 bytes) that it's not able to do much other than load and run a second-stage boot loader.

Neither the boot block nor the BIOS is sophisticated enough to read any type of standard filesystem, so the second-stage boot loader must be kept somewhere easy to find. In one typical scenario, the boot block reads the partitioning information from the MBR and identifies the disk partition marked as "active." It then reads and executes the second-stage boot loader from the beginning of that partition. This scheme is known as a volume boot record.

Alternatively, the second-stage boot loader can live in the dead zone that lies between the MBR and the beginning of the first disk partition. For historical reasons, the first partition doesn't start until the 64th disk block, so this zone normally contains at least 32KB of storage: still not a lot, but enough to store a filesystem driver. This storage scheme is commonly used by the GRUB boot loader; see page 35.

To effect a successful boot, all components of the boot chain must be properly installed and compatible with one another. The MBR boot block is OS-agnostic, but because it assumes a particular location for the second stage, there may be multiple versions that can be installed. The second-stage loader is generally knowledgeable about operating systems and filesystems (it may support several of each), and usually has configuration options of its own.

UEFI

See page 746 for more information about GPT partitions.

The UEFI specification includes a modern disk partitioning scheme known as GPT (GUID Partition Table, where GUID stands for "globally unique identifier"). UEFI also understands FAT (File Allocation Table) filesystems, a simple but functional layout that originated in MS-DOS. These features combine to define the concept of an EFI System Partition (ESP). At boot time, the firmware consults the GPT partition table to identify the ESP. It then reads the configured target application directly from a file in the ESP and executes it.

Because the ESP is just a generic FAT filesystem, it can be mounted, read, written, and maintained by any operating system. No "mystery meat" boot blocks are required anywhere on the disk.[3]

In fact, no boot loader at all is technically required. The UEFI boot target can be a UNIX or Linux kernel that has been configured for direct UEFI loading, thus effecting a loader-less bootstrap. In practice, though, most systems still use a boot loader, partly because that makes it easier to maintain compatibility with legacy BIOSes.

UEFI saves the pathname to load from the ESP as a configuration parameter. With no configuration, it looks for a standard path, usually **/efi/boot/bootx64.efi** on modern Intel systems. A more typical path on a configured system (this one for Ubuntu and the GRUB boot loader) would be **/efi/ubuntu/grubx64.efi**. Other distributions follow a similar convention.

UEFI defines standard APIs for accessing the system's hardware. In this respect, it's something of a miniature operating system in its own right. It even provides for UEFI-level add-on device drivers, which are written in a processor-independent language and stored in the ESP. Operating systems can use the UEFI interface, or they can take over direct control of the hardware.

Because UEFI has a formal API, you can examine and modify UEFI variables (including boot menu entries) on a running system. For example, **efibootmgr -v** shows the following summary of the boot configuration:

```
$ efibootmgr -v
BootCurrent: 0004
BootOrder: 0000,0001,0002,0004,0003
Boot0000* EFI DVD/CDROM PciRoot(0x0)/Pci(0x1f,0x2)/Sata(1,0,0)
Boot0001* EFI Hard Drive PciRoot(0x0)/Pci(0x1f,0x2)/Sata(0,0,0)
Boot0002* EFI Network PciRoot(0x0)/Pci(0x5,0x0)/MAC(001c42fb5baf,0)
Boot0003* EFI Internal Shell MemoryMapped(11,0x7ed5d000,0x7f0dcfff)/
    FvFile(c57ad6b7-0515-40a8-9d21-551652854e37)
Boot0004* ubuntu HD(1,GPT,020c8d3e-fd8c-4880-9b61-
    ef4cffc3d76c,0x800,0x100000)/File(\EFI\ubuntu\shimx64.efi)
```

efibootmgr lets you change the boot order, select the next configured boot option, or even create and destroy boot entries. For example, to set the boot order to try the system drive before trying the network, and to ignore other boot options, we could use the command

```
$ sudo efibootmgr -o 0004,0002
```

Here, we're specifying the options `Boot0004` and `Boot0002` from the output above.

The ability to modify the UEFI configuration from user space means that the firmware's configuration information is mounted read/write—a blessing and a curse. On

3. Truth be told, UEFI does maintain an MBR-compatible record at the beginning of each disk to facilitate interoperability with BIOS systems. BIOS systems can't see the full GPT-style partition table, but they at least recognize the disk as having been formatted. Be careful not to run MBR-specific administrative tools on GPT disks. They may think they understand the disk layout, but they do not.

systems (typically, those with **systemd**) that allow write access by default, **rm -rf /** can be enough to permanently destroy the system at the firmware level; in addition to removing files, **rm** also removes variables and other UEFI information accessible through **/sys**.[4] Yikes! Don't try this at home!

2.3 Boot loaders

Most bootstrapping procedures include the execution of a boot loader that is distinct from both the BIOS/UEFI code and the OS kernel. It's also separate from the initial boot block on a BIOS system, if you're counting steps.

The boot loader's main job is to identify and load an appropriate operating system kernel. Most boot loaders can also present a boot-time user interface that lets you select which of several possible kernels or operating systems to invoke.

Another task that falls to the boot loader is the marshaling of configuration arguments for the kernel. The kernel doesn't have a command line per se, but its startup option handling will seem eerily similar from the shell. For example, the argument **single** or **-s** usually tells the kernel to enter single-user mode instead of completing the normal boot process.

Such options can be hard-wired into the boot loader's configuration if you want them used on every boot, or they can be provided on the fly through the boot loader's UI.

In the next few sections, we discuss GRUB (the Linux world's predominant boot loader) and the boot loaders used with FreeBSD.

2.4 GRUB: the GRand Unified Boot loader

GRUB, developed by the GNU Project, is the default boot loader on most Linux distributions. The GRUB lineage has two main branches: the original GRUB, now called GRUB Legacy, and the newer, extra-crispy GRUB 2, which is the current standard. Make sure you know which GRUB you're dealing with, as the two versions are quite different.

GRUB 2 has been the default boot manager for Ubuntu since version 9.10, and it recently became the default for RHEL 7. All our example Linux distributions use it as their default. In this book we discuss only GRUB 2, and we refer to it simply as GRUB.

FreeBSD has its own boot loader (covered in more detail starting on page 39). However, GRUB is perfectly happy to boot FreeBSD, too. This might be an advantageous configuration if you're planning to boot multiple operating systems on a single computer. Otherwise, the FreeBSD boot loader is more than adequate.

4. See goo.gl/QMSiSG (link to Phoronix article) for some additional details.

GRUB configuration

See page 49 for more about operating modes.

GRUB lets you specify parameters such as the kernel to boot (specified as a GRUB "menu entry") and the operating mode to boot into.

Since this configuration information is needed at boot time, you might imagine that it would be stored somewhere strange, such as the system's NVRAM or the disk blocks reserved for the boot loader. In fact, GRUB understands most of the filesystems in common use and can usually find its way to the root filesystem on its own. This feat lets GRUB read its configuration from a regular text file.

The config file is called **grub.cfg**, and it's usually kept in **/boot/grub** (**/boot/grub2** in Red Hat and CentOS) along with a selection of other resources and code modules that GRUB might need to access at boot time.[5] Changing the boot configuration is a simple matter of updating the **grub.cfg** file.

Although you can create the **grub.cfg** file yourself, it's more common to generate it with the **grub-mkconfig** utility, which is called **grub2-mkconfig** on Red Hat and CentOS and wrapped as **update-grub** on Debian and Ubuntu. In fact, most distributions assume that **grub.cfg** can be regenerated at will, and they do so automatically after updates. If you don't take steps to prevent this, your handcrafted **grub.cfg** file will get clobbered.

As with all things Linux, distributions configure **grub-mkconfig** in a variety of ways. Most commonly, the configuration is specified in **/etc/default/grub** in the form of **sh** variable assignments. Table 2.1 shows some of the commonly modified options.

Table 2.1 Common GRUB configuration options from /etc/default/grub

Shell variable name	Contents or function
GRUB_BACKGROUND	Background image [a]
GRUB_CMDLINE_LINUX	Kernel parameters to add to menu entries for Linux [b]
GRUB_DEFAULT	Number or title of the default menu entry
GRUB_DISABLE_RECOVERY	Prevents the generation of recovery mode entries
GRUB_PRELOAD_MODULES	List of GRUB modules to be loaded as early as possible
GRUB_TIMEOUT	Seconds to display the boot menu before autoboot

a. The background image must be a **.png**, **.tga**, **.jpg**, or **.jpeg** file.
b. Table 2.3 on page 38 lists some of the available options.

5. If you're familiar with UNIX filesystem conventions (see Chapter 5, *The Filesystem*, starting on page 120), you might wonder why **/boot/grub** isn't named something more standard-looking such as **/var/lib/grub** or **/usr/local/etc/grub**. The reason is that the filesystem drivers used at boot time are somewhat simplified. Boot loaders can't handle advanced features such as mount points as they traverse the filesystem. Everything in **/boot** should be a simple file or directory.

After editing **/etc/default/grub**, run **update-grub** or **grub2-mkconfig** to translate your configuration into a proper **grub.cfg** file. As part of the configuration-building process, these commands inventory the system's bootable kernels, so they can be useful to run after you make kernel changes even if you haven't explicitly changed the GRUB configuration.

You may need to edit the **/etc/grub.d/40_custom** file to change the order in which kernels are listed in the boot menu (after you create a custom kernel, for example), set a boot password, or change the names of boot menu items. As usual, run **update-grub** or **grub2-mkconfig** after making changes.

As an example, here's a **40_custom** file that invokes a custom kernel on an Ubuntu system:

```
#!/bin/sh

exec tail -n +3 $0

# This file provides an easy way to add custom menu entries. Just type
# the menu entries you want to add after this comment. Be careful not to
# change the 'exec tail' line above.

menuentry 'My Awesome Kernel' {
    set root='(hd0,msdos1)'
    linux   /awesome_kernel root=UUID=XXX-XXX-XXX ro quiet
    initrd  /initrd.img-awesome_kernel
}
```

See page 122 for more information about mounting filesystems.

In this example, GRUB loads the kernel from **/awesome_kernel**. Kernel paths are relative to the boot partition, which historically was mounted as **/boot** but with the advent of UEFI now is likely an unmounted EFI System Partition. Use **gpart show** and **mount** to examine your disk and determine the state of the boot partition.

The GRUB command line

GRUB supports a command-line interface for editing config file entries on the fly at boot time. To enter command-line mode, type **c** at the GRUB boot screen.

From the command line, you can boot operating systems that aren't listed in the **grub.cfg** file**,** display system information, and perform rudimentary filesystem testing. Anything that can be done through **grub.cfg** can also be done through the command line.

Press the <Tab> key to see a list of possible commands. Table 2.2 on the next page shows some of the more useful ones.

Table 2.2 GRUB commands

Cmd	Function
boot	Boots the system from the specified kernel image
help	Gets interactive help for a command
linux	Loads a Linux kernel
reboot	Reboots the system
search	Searches devices by file, filesystem label, or UUID
usb	Tests USB support

For detailed information about GRUB and its command-line options, refer to the official manual at gnu.org/software/grub/manual.

Linux kernel options

See Chapter 11 for more about kernel parameters.

Kernel startup options typically modify the values of kernel parameters, instruct the kernel to probe for particular devices, specify the path to the **init** or **systemd** process, or designate a particular root device. Table 2.3 shows a few examples.

Table 2.3 Examples of kernel boot time options

Option	Meaning
`debug`	Turns on kernel debugging
`init=/bin/bash`	Starts only the **bash** shell; useful for emergency recovery
`root=/dev/foo`	Tells the kernel to use **/dev/foo** as the root device
`single`	Boots to single-user mode

When specified at boot time, kernel options are not persistent. Edit the appropriate kernel line in **/etc/grub.d/40_custom** or **/etc/defaults/grub** (the variable named `GRUB_CMDLINE_LINUX`) to make the change permanent across reboots.

Security patches, bug fixes, and features are all regularly added to the Linux kernel. Unlike other software packages, however, new kernel releases typically do not replace old ones. Instead, the new kernels are installed side by side with the previous versions so that you can return to an older kernel in the event of problems.

This convention helps administrators back out of an upgrade if a kernel patch breaks their system, although it also means that the boot menu tends to get cluttered with old versions of the kernel. Try choosing a different kernel if your system won't boot after an update.

2.5 The FreeBSD boot process

FreeBSD's boot system is a lot like GRUB in that the final-stage boot loader (called **loader**) uses a filesystem-based configuration file, supports menus, and offers an interactive, command-line-like mode. **loader** is the final common pathway for both the BIOS and UEFI boot paths.

The BIOS path: boot0

As with GRUB, the full **loader** environment is too large to fit in an MBR boot block, so a chain of progressively more sophisticated preliminary boot loaders get **loader** up and running on a BIOS system.

GRUB bundles all of these components under the umbrella name "GRUB," but in FreeBSD, the early boot loaders are part of a separate system called **boot0** that's used only on BIOS systems. **boot0** has options of its own, mostly because it stores later stages of the boot chain in a volume boot record (see *Legacy BIOS* on page 33) rather than in front of the first disk partition.

For that reason, the MBR boot record needs a pointer to the partition it should use to continue the boot process. Normally, all this is automatically set up for you as part of the FreeBSD installation process, but if you should ever need to adjust the configuration, you can do so with the **boot0cfg** command.

The UEFI path

On UEFI systems, FreeBSD creates an EFI system partition and installs boot code there under the path **/boot/bootx64.efi**.[6] This is the default path that UEFI systems check at boot time (at least on modern PC platforms), so no firmware-level configuration should be needed other than ensuring that device boot priorities are properly set.

By default, FreeBSD doesn't keep the EFI system partition mounted after booting. You can inspect the partition table with **gpart** to identify it:

```
$ gpart show
=>         40  134217648  ada0  GPT  (64G)
           40       1600     1  efi  (800K)
         1640  127924664     2  freebsd-ufs  (61G)
    127926304    6291383     3  freebsd-swap  (3.0G)
    134217687          1        - free -  (512B)
```

See page 122 for more information about mounting filesystems.

Although you can mount the ESP if you're curious to see what's in it (use **mount**'s **-t msdos** option), the whole filesystem is actually just a copy of an image found in **/boot/boot1.efifat** on the root disk. No user-serviceable parts inside.

6. Don't confuse the **/boot** directory in the EFI system partition with the **/boot** directory in the FreeBSD root filesystem. They are separate and serve different purposes, although of course both are bootstrapping related.

If the ESP partition gets damaged or removed, you can re-create it by setting up the partition with **gpart** and then copying in the filesystem image with **dd**:

```
$ sudo dd if=/boot/boot1.efifat of=/dev/ada0p1
```

Once the first-stage UEFI boot loader finds a partition of type `freebsd-ufs`,[7] it loads a UEFI version of the **loader** software from **/boot/loader.efi**. From there, booting proceeds as under BIOS, with **loader** determining the kernel to load, the kernel parameters to set, and so on.

loader configuration

loader is actually a scripting environment, and the scripting language is Forth.[8] There's a bunch of Forth code stored under **/boot** that controls **loader**'s operations, but it's designed to be self-contained—you needn't learn Forth.

The Forth scripts execute **/boot/loader.conf** to obtain the values of configuration variables, so that's where your customizations should go. Don't confuse this file with **/boot/defaults/loader.conf**, which contains the configuration defaults and isn't intended for modification. Fortunately, the variable assignments in **loader.conf** are syntactically similar to standard **sh** assignments.

The man pages for **loader** and **loader.conf** give the dirt on all the boot loader options and the configuration variables that control them. Some of the more interesting options include those for protecting the boot menu with a password, changing the splash screen displayed at boot, and passing kernel options.

loader commands

loader understands a variety of interactive commands. For example, to locate and boot an alternate kernel, you'd use a sequence of commands like this:

```
Type '?' for a list of commands, 'help' for more detailed help.
OK ls
/
 d .snap
 d dev
...
d rescue
l home
...
OK unload
OK load /boot/kernel/kernel.old
/boot/kernel/kernel.old text=0xf8f898 data=0x124 ... b077]
OK boot
```

7. As of FreeBSD 10.1, it is now possible to use ZFS as the root partition on a UEFI system.

8. This is a remarkable and interesting fact if you're a historian of programming languages, and unimportant otherwise.

Here, we listed the contents of the (default) root filesystem, unloaded the default kernel (**/boot/kernel/kernel**), loaded an older kernel (**/boot/kernel/kernel.old**), and then continued the boot process.

See **man loader** for complete documentation of the available commands.

2.6 System management daemons

Once the kernel has been loaded and has completed its initialization process, it creates a complement of "spontaneous" processes in user space. They're called spontaneous processes because the kernel starts them autonomously—in the normal course of events, new processes are created only at the behest of existing processes.

Most of the spontaneous processes are really part of the kernel implementation. They don't necessarily correspond to programs in the filesystem. They're not configurable, and they don't require administrative attention. You can recognize them in **ps** listings (see page 98) by their low PIDs and by the brackets around their names (for example, `[pagedaemon]` on FreeBSD or `[kdump]` on Linux).

The exception to this pattern is the system management daemon. It has process ID 1 and usually runs under the name **init**. The system gives **init** a couple of special privileges, but for the most part it's just a user-level program like any other daemon.

Responsibilities of init

init has multiple functions, but its overarching goal is to make sure the system runs the right complement of services and daemons at any given time.

To serve this goal, **init** maintains a notion of the mode in which the system should be operating. Some commonly defined modes:[9]

- Single-user mode, in which only a minimal set of filesystems is mounted, no services are running, and a root shell is started on the console
- Multiuser mode, in which all customary filesystems are mounted and all configured network services have been started, along with a window system and graphical login manager for the console
- Server mode, similar to multiuser mode, but with no GUI running on the console

Every mode is associated with a defined complement of system services, and the initialization daemon starts or stops services as needed to bring the system's actual state into line with the currently active mode. Modes can also have associated milepost tasks that run whenever the mode begins or ends.

9. Don't take these mode names or descriptions too literally; they're just examples of common operating modes that most systems define in one way or another.

As an example, **init** normally takes care of many different startup chores as a side effect of its transition from bootstrapping to multiuser mode. These may include

- Setting the name of the computer
- Setting the time zone
- Checking disks with **fsck**
- Mounting filesystems
- Removing old files from the **/tmp** directory
- Configuring network interfaces
- Configuring packet filters
- Starting up other daemons and network services

init has very little built-in knowledge about these tasks. In simply runs a set of commands or scripts that have been designated for execution in that particular context.

Implementations of init

Today, three very different flavors of system management processes are in widespread use:

- An **init** styled after the **init** from AT&T's System V UNIX, which we refer to as "traditional **init**." This was the predominant **init** used on Linux systems until the debut of **systemd**.
- An **init** variant that derives from BSD UNIX and is used on most BSD-based systems, including FreeBSD, OpenBSD, and NetBSD. This one is just as tried-and-true as the SysV **init** and has just as much claim to being called "traditional," but for clarity we refer to it as "BSD **init**." This variant is quite simple in comparison with SysV-style **init**. We discuss it separately starting on page 57.
- A more recent contender called **systemd** which aims to be one-stop-shopping for all daemon- and state-related issues. As a consequence, **systemd** carves out a significantly larger territory than any historical version of **init**. That makes it somewhat controversial, as we discuss below. Nevertheless, all our example Linux distributions have now adopted **systemd**.

Although these implementations are the predominant ones today, they're far from being the only choices. Apple's macOS, for example, uses a system called **launchd**. Until it adopted **systemd**, Ubuntu used another modern **init** variant called Upstart.

On Linux systems, you can theoretically replace your system's default **init** with whichever variant you prefer. But in practice, **init** is so fundamental to the operation of the system that a lot of add-on software is likely to break. If you can't abide **systemd**, standardize on a distribution that doesn't use it.

Traditional init

In the traditional **init** world, system modes (e.g., single-user or multiuser) are known as "run levels." Most run levels are denoted by a single letter or digit.

Traditional **init** has been around since the early 80s, and grizzled folks in the anti-**systemd** camp often cite the principle, "If it ain't broke, don't fix it." That said, traditional **init** does have a number of notable shortcomings.

To begin with, the traditional **init** on its own is not really powerful enough to handle the needs of a modern system. Most systems that use it actually have a standard and fixed **init** configuration that never changes. That configuration points to a second tier of shell scripts that do the actual work of changing run levels and letting administrators make configuration changes.

The second layer of scripts maintains yet a third layer of daemon- and system-specific scripts, which are cross-linked to run-level-specific directories that indicate what services are supposed to be running at what run level. It's all a bit hackish and unsightly.

More concretely, this system has no general model of dependencies among services, so it requires that all startups and takedowns be run in a numeric order that's maintained by the administrator. Later actions can't run until everything ahead of them has finished, so it's impossible to execute actions in parallel, and the system takes a long time to change states.

systemd vs. the world

Few issues in the Linux space have been more hotly debated than the migration from traditional **init** to **systemd**. For the most part, complaints center on **systemd**'s seemingly ever-increasing scope.

systemd takes all the **init** features formerly implemented with sticky tape, shell script hacks, and the sweat of administrators and formalizes them into a unified field theory of how services should be configured, accessed, and managed.

See Chapter 6, Software Installation and Management, for more information about package management.

Much like a package management system, **systemd** defines a robust dependency model, not only among services but also among "targets," **systemd**'s term for the operational modes that traditional **init** calls run levels. **systemd** not only manages processes in parallel, but also manages network connections (**networkd**), kernel log entries (**journald**), and logins (**logind**).

The anti-**systemd** camp argues that the UNIX philosophy is to keep system components small, simple, and modular. A component as fundamental as **init**, they say, should not have monolithic control over so many of the OS's other subsystems. **systemd** not only breeds complexity, but also introduces potential security weaknesses and muddies the distinction between the OS platform and the services that run on top of it.

systemd has also received criticism for imposing new standards and responsibilities on the Linux kernel, for its code quality, for the purported unresponsiveness of its developers to bug reports, for the functional design of its basic features, and for looking at people funny. We can't fairly address these issues here, but you may find it informative to peruse the *Arguments against systemd* section at without-systemd.org, the Internet's premier **systemd** hate site.

inits judged and assigned their proper punishments

The architectural objections to **systemd** outlined above are all reasonable points. **systemd** does indeed display most of the telltale stigmata of an overengineered software project.

In practice, however, many administrators quite like **systemd**, and we fall squarely into this camp. Ignore the controversy for a bit and give **systemd** a chance to win your love. Once you've become accustomed to it, you will likely find yourself appreciating its many merits.

At the very least, keep in mind that the traditional **init** that **systemd** displaces was no national treasure. If nothing else, **systemd** delivers some value just by eliminating a few of the unnecessary differences among Linux distributions.

The debate really doesn't matter anymore because the **systemd** coup is over. The argument was effectively settled when Red Hat, Debian, and Ubuntu switched. Many other Linux distributions are now adopting **systemd**, either by choice or by being dragged along, kicking and screaming, by their upstream distributions.

Traditional **init** still has a role to play when a distribution either targets a small installation footprint or doesn't need **systemd**'s advanced process management functions. There's also a sizable population of revanchists who disdain **systemd** on principle, so some Linux distributions are sure to keep traditional **init** alive indefinitely as a form of protest theater.

Nevertheless, we don't think that traditional **init** has enough of a future to merit a detailed discussion in this book. For Linux, we mostly limit ourselves to **systemd**. We also discuss the mercifully simple system used by FreeBSD, starting on page 57.

2.7 SYSTEMD IN DETAIL

The configuration and control of system services is an area in which Linux distributions have traditionally differed the most from one another. **systemd** aims to standardize this aspect of system administration, and to do so, it reaches further into the normal operations of the system than any previous alternative.

systemd is not a single daemon but a collection of programs, daemons, libraries, technologies, and kernel components. A post on the **systemd** blog at 0pointer.de/blog notes that a full build of the project generates 69 different binaries. Think of it as a scrumptious buffet at which you are forced to consume everything.

Since **systemd** depends heavily on features of the Linux kernel, it's a Linux-only proposition. You won't see it ported to BSD or to any other variant of UNIX within the next five years.

Units and unit files

An entity that is managed by **systemd** is known generically as a unit. More specifically, a unit can be "a service, a socket, a device, a mount point, an automount point, a swap file or partition, a startup target, a watched filesystem path, a timer controlled and supervised by **systemd**, a resource management slice, a group of externally created processes, or a wormhole into an alternate universe."[10] OK, we made up the part about the alternate universe, but that still covers a lot of territory.

Within **systemd**, the behavior of each unit is defined and configured by a unit file. In the case of a service, for example, the unit file specifies the location of the executable file for the daemon, tells **systemd** how to start and stop the service, and identifies any other units that the service depends on.

*See page 594 for more information about **rsync**.*

We explore the syntax of unit files in more detail soon, but here's a simple example from an Ubuntu system as an appetizer. This unit file is **rsync.service**; it handles startup of the **rsync** daemon that mirrors filesystems.

```
[Unit]
Description=fast remote file copy program daemon
ConditionPathExists=/etc/rsyncd.conf

[Service]
ExecStart=/usr/bin/rsync --daemon --no-detach

[Install]
WantedBy=multi-user.target
```

If you recognize this as the file format used by MS-DOS **.ini** files, you are well on your way to understanding both **systemd** and the anguish of the **systemd** haters.

Unit files can live in several different places. **/usr/lib/systemd/system** is the main place where packages deposit their unit files during installation; on some systems, the path is **/lib/systemd/system** instead. The contents of this directory are considered stock, so you shouldn't modify them. Your local unit files and customizations can go in **/etc/systemd/system**. There's also a unit directory in **/run/systemd/system** that's a scratch area for transient units.

systemd maintains a telescopic view of all these directories, so they're pretty much equivalent. If there's any conflict, the files in **/etc** have the highest priority.

10. Mostly quoted from the **systemd.unit** man page

By convention, unit files are named with a suffix that varies according to the type of unit being configured. For example, service units have a **.service** suffix and timers use **.timer**. Within the unit file, some sections (e.g., `[Unit]`) apply generically to all kinds of units, but others (e.g., `[Service]`) can appear only in the context of a particular unit type.

systemctl: manage systemd

See page 235 for more information about Git.

systemctl is an all-purpose command for investigating the status of **systemd** and making changes to its configuration. As with Git and several other complex software suites, **systemctl**'s first argument is typically a subcommand that sets the general agenda, and subsequent arguments are specific to that particular subcommand. The subcommands could be top-level commands in their own right, but for consistency and clarity, they're bundled into the **systemctl** omnibus.

Running **systemctl** without any arguments invokes the default **list-units** subcommand, which shows all loaded and active services, sockets, targets, mounts, and devices. To show only loaded and active services, use the **--type=service** qualifier:

```
$ systemctl list-units --type=service
UNIT                     LOAD   ACTIVE  SUB      DESCRIPTION
accounts-daemon.service  loaded active  running  Accounts Service
...
wpa_supplicant.service   loaded active  running  WPA supplicant
```

It's also sometimes helpful to see all the installed unit files, regardless of whether or not they're active:

```
$ systemctl list-unit-files --type=service
UNIT FILE                   STATE
...
cron.service                enabled
cryptdisks-early.service    masked
cryptdisks.service          masked
cups-browsed.service        enabled
cups.service                disabled
...
wpa_supplicant.service      disabled
x11-common.service          masked

188 unit files listed.
```

For subcommands that act on a particular unit (e.g., **systemctl status**) **systemctl** can usually accept a unit name without a unit-type suffix (e.g., **cups** instead of **cups.service**). However, the default unit type with which simple names are fleshed out varies by subcommand.

Table 2.4 shows the most common and useful **systemctl** subcommands. See the **systemctl** man page for a complete list.

Table 2.4 Commonly used systemctl subcommands

Subcommand	Function
list-unit-files [*pattern*]	Shows installed units; optionally matching *pattern*
enable *unit*	Enables *unit* to activate at boot
disable *unit*	Prevents *unit* from activating at boot
isolate *target*	Changes operating mode to *target*
start *unit*	Activates *unit* immediately
stop *unit*	Deactivates *unit* immediately
restart *unit*	Restarts (or starts, if not running) *unit* immediately
status *unit*	Shows *unit*'s status and recent log entries
kill *pattern*	Sends a signal to units matching *pattern*
reboot	Reboots the computer
daemon-reload	Reloads unit files and **systemd** configuration

Unit statuses

In the output of **systemctl list-unit-files** above, we can see that **cups.service** is disabled. We can use **systemctl status** to find out more details:

```
$ sudo systemctl status -l cups
cups.service - CUPS Scheduler
   Loaded: loaded (/lib/systemd/system/cups.service; disabled; vendor
     preset: enabled)
   Active: inactive (dead) since Sat 2016-12-12 00:51:40 MST; 4s ago
     Docs: man:cupsd(8)
 Main PID: 10081 (code=exited, status=0/SUCCESS)

Dec 12 00:44:39 ulsah systemd[1]: Started CUPS Scheduler.
Dec 12 00:44:45 ulsah systemd[1]: Started CUPS Scheduler.
Dec 12 00:51:40 ulsah systemd[1]: Stopping CUPS Scheduler...
Dec 12 00:51:40 ulsah systemd[1]: Stopped CUPS Scheduler.
```

Here, **systemctl** shows us that the service is currently inactive (`dead`) and tells us when the process died. (Just a few seconds ago; we disabled it for this example.) It also shows (in the section marked `Loaded`) that the service defaults to being enabled at startup, but that it is presently disabled.

The last four lines are recent log entries. By default, the log entries are condensed so that each entry takes only one line. This compression often makes entries unreadable, so we included the **-l** option to request full entries. It makes no difference in this case, but it's a useful habit to acquire.

Table 2.5 on the next page shows the statuses you'll encounter most frequently when checking up on units.

Table 2.5 Unit file statuses

State	Meaning
`bad`	Some kind of problem within **systemd**; usually a bad unit file
`disabled`	Present, but not configured to start autonomously
`enabled`	Installed and runnable; will start autonomously
`indirect`	Disabled, but has peers in `Also` clauses that may be enabled
`linked`	Unit file available through a symlink
`masked`	Banished from the **systemd** world from a logical perspective
`static`	Depended upon by another unit; has no install requirements

The `enabled` and `disabled` states apply only to unit files that live in one of **systemd's system** directories (that is, they are not linked in by a symbolic link) and that have an `[Install]` section in their unit files. "Enabled" units should perhaps really be thought of as "installed," meaning that the directives in the `[Install]` section have been executed and that the unit is wired up to its normal activation triggers. In most cases, this state causes the unit to be activated automatically once the system is bootstrapped.

Likewise, the `disabled` state is something of a misnomer because the only thing that's actually disabled is the normal activation path. You can manually activate a unit that is `disabled` by running **systemctl start**; **systemd** won't complain.

Many units have no installation procedure, so they can't truly be said to be enabled or disabled; they're just available. Such units' status is listed as `static`. They only become active if activated by hand (**systemctl start**) or named as a dependency of other active units.

Unit files that are `linked` were created with **systemctl link**. This command creates a symbolic link from one of **systemd**'s **system** directories to a unit file that lives elsewhere in the filesystem. Such unit files can be addressed by commands or named as dependencies, but they are not full citizens of the ecosystem and have some notable quirks. For example, running **systemctl disable** on a `linked` unit file deletes the link and all references to it.

Unfortunately, the exact behavior of linked unit files is not well documented. Although the idea of keeping local unit files in a separate repository and linking them into **systemd** has a certain appeal, it's probably not the best approach at this point. Just make copies.

The `masked` status means "administratively blocked." **systemd** knows about the unit, but has been forbidden from activating it or acting on any of its configuration directives by **systemctl mask**. As a rule of thumb, turn off units whose status is `enabled` or `linked` with **systemctl disable** and reserve **systemctl mask** for `static` units.

Returning to our investigation of the **cups** service, we could use the following commands to reenable and start it:

```
$ sudo systemctl enable cups
Synchronizing state of cups.service with SysV init with /lib/systemd/
    systemd-sysv-install...
Executing /lib/systemd/systemd-sysv-install enable cups
insserv: warning: current start runlevel(s) (empty) of script 'cups'
    overrides LSB defaults (2 3 4 5).
insserv: warning: current stop runlevel(s) (1 2 3 4 5) of script 'cups'
    overrides LSB defaults (1).
Created symlink from /etc/systemd/system/sockets.target.wants/cups.socket
    to /lib/systemd/system/cups.socket.
Created symlink from /etc/systemd/system/multi-user.target.wants/cups.
    path to /lib/systemd/system/cups.path.
$ sudo systemctl start cups
```


Targets

Unit files can declare their relationships to other units in a variety of ways. In the example on page 45, for example, the `WantedBy` clause says that if the system has a **multi-user.target** unit, that unit should depend on this one (**rsync.service**) when this unit is enabled.

Because units directly support dependency management, no additional machinery is needed to implement the equivalent of **init**'s run levels. For clarity, **systemd** does define a distinct class of units (of type **.target**) to act as well-known markers for common operating modes. However, targets have no real superpowers beyond the dependency management that's available to any other unit.

Traditional **init** defines at least seven numeric run levels, but many of those aren't actually in common use. In a perhaps-ill-advised gesture toward historical continuity, **systemd** defines targets that are intended as direct analogs of the **init** run levels (**runlevel0.target**, etc.). It also defines mnemonic targets for day-to-day use such as **poweroff.target** and **graphical.target**. Table 2.6 on the next page shows the mapping between **init** run levels and **systemd** targets.

The only targets to really be aware of are **multi-user.target** and **graphical.target** for day-to-day use, and **rescue.target** for accessing single-user mode. To change the system's current operating mode, use the **systemctl isolate** command:

```
$ sudo systemctl isolate multi-user.target
```

The **isolate** subcommand is so-named because it activates the stated target and its dependencies but deactivates all other units.

Under traditional **init**, you use the **telinit** command to change run levels once the system is booted. Some distributions now define **telinit** as a symlink to the **systemctl** command, which recognizes how it's being invoked and behaves appropriately.

Table 2.6 Mapping between init run levels and systemd targets

Run level	Target	Description
0	**poweroff.target**	System halt
emergency	**emergency.target**	Bare-bones shell for system recovery
1, s, single	**rescue.target**	Single-user mode
2	**multi-user.target** [a]	Multiuser mode (command line)
3	**multi-user.target** [a]	Multiuser mode with networking
4	**multi-user.target** [a]	Not normally used by **init**
5	**graphical.target**	Multiuser mode with networking and GUI
6	**reboot.target**	System reboot

a. By default, **multi-user.target** maps to **runlevel3.target**, multiuser mode with networking.

To see the target the system boots into by default, run the **get-default** subcommand:

```
$ systemctl get-default
graphical.target
```

Most Linux distributions boot to **graphical.target** by default, which isn't appropriate for servers that don't need a GUI. But that's easily changed:

```
$ sudo systemctl set-default multi-user.target
```

To see all the system's available targets, run **systemctl list-units**:

```
$ systemctl list-units --type=target
```

Dependencies among units

Linux software packages generally come with their own unit files, so administrators don't need a detailed knowledge of the entire configuration language. However, they do need a working knowledge of **systemd**'s dependency system to diagnose and fix dependency problems.

To begin with, not all dependencies are explicit. **systemd** takes over the functions of the old **inetd** and also extends this idea into the domain of the D-Bus interprocess communication system. In other words, **systemd** knows which network ports or IPC connection points a given service will be hosting, and it can listen for requests on those channels without actually starting the service. If a client does materialize, **systemd** simply starts the actual service and passes off the connection. The service runs if it's actually used and remains dormant otherwise.

Second, **systemd** makes some assumptions about the normal behavior of most kinds of units. The exact assumptions vary by unit type. For example, **systemd** assumes that the average service is an add-on that shouldn't be running during the early phases of system initialization. Individual units can turn off these assumptions with the line

```
DefaultDependencies=false
```

in the `[Unit]` section of their unit file; the default is `true`. See the man page for **systemd.***unit-type* to see the exact assumptions that apply to each type of unit (e.g., **man systemd.service**).

A third class of dependencies are those explicitly declared in the `[Unit]` sections of unit files. Table 2.7 shows the available options.

Table 2.7 Explicit dependencies in the [Unit] section of unit files

Option	Meaning
`Wants`	Units that should be coactivated if possible, but are not required
`Requires`	Strict dependencies; failure of any prerequisite terminates this service
`Requisite`	Like `Requires`, but must already be active
`BindsTo`	Similar to `Requires`, but even more tightly coupled
`PartOf`	Similar to `Requires`, but affects only starting and stopping
`Conflicts`	Negative dependencies; cannot be coactive with these units

With the exception of `Conflicts`, all the options in Table 2.7 express the basic idea that the unit being configured depends on some set of other units. The exact distinctions among these options are subtle and primarily of interest to service developers. The least restrictive variant, `Wants`, is preferred when possible.

You can extend a unit's `Wants` or `Requires` cohorts by creating a *unit-file***.wants** or *unit-file***.requires** directory in **/etc/systemd/system** and adding symlinks there to other unit files. Better yet, just let **systemctl** do it for you. For example, the command

```
$ sudo systemctl add-wants multi-user.target my.local.service
```

adds a dependency on **my.local.service** to the standard multiuser target, ensuring that the service will be started whenever the system enters multiuser mode.

In most cases, such ad hoc dependencies are automatically taken care of for you, courtesy of the `[Install]` sections of unit files. This section includes `WantedBy` and `RequiredBy` options that are read only when a unit is enabled with **systemctl enable** or disabled with **systemctl disable**. On enablement, they make **systemctl** perform the equivalent of an **add-wants** for every `WantedBy` or an **add-requires** for every `RequiredBy`.

The `[Install]` clauses themselves have no effect in normal operation, so if a unit doesn't seem to be started when it should be, make sure that it has been properly enabled and symlinked.

Execution order

You might reasonably guess that if unit A `Requires` unit B, then unit B will be started or configured before unit A. But in fact that is not the case. In **systemd**, the order

in which units are activated (or deactivated) is an *entirely separate* question from that of which units to activate.

When the system transitions to a new state, **systemd** first traces the various sources of dependency information outlined in the previous section to identify the units that will be affected. It then uses `Before` and `After` clauses from the unit files to sort the work list appropriately. To the extent that units have no `Before` or `After` constraints, they are free to be adjusted in parallel.

Although potentially surprising, this is actually a praiseworthy design feature. One of the major design goals of **systemd** was to facilitate parallelism, so it makes sense that units do not acquire serialization dependencies unless they explicitly ask for them.

In practice, `After` clauses are typically used more frequently than `Wants` or `Requires`. Target definitions (and in particular, the reverse dependencies encoded in `WantedBy` and `RequiredBy` clauses) establish the general outlines of the services running in each operating mode, and individual packages worry only about their immediate and direct dependencies.

A more complex unit file example

Now for a closer look at a few of the directives used in unit files. Here's a unit file for the NGINX web server, **nginx.service**:

```
[Unit]
Description=The nginx HTTP and reverse proxy server
After=network.target remote-fs.target nss-lookup.target

[Service]
Type=forking
PIDFile=/run/nginx.pid
ExecStartPre=/usr/bin/rm -f /run/nginx.pid
ExecStartPre=/usr/sbin/nginx -t
ExecStart=/usr/sbin/nginx
ExecReload=/bin/kill -s HUP $MAINPID
KillMode=process
KillSignal=SIGQUIT
TimeoutStopSec=5
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

This service is of type `forking`, which means that the startup command is expected to terminate even though the actual daemon continues running in the background. Since **systemd** won't have directly started the daemon, the daemon records its PID (process ID) in the stated `PIDFile` so that **systemd** can determine which process is the daemon's primary instance.

The `Exec` lines are commands to be run in various circumstances. `ExecStartPre` commands are run before the actual service is started; the ones shown here validate

the syntax of NGINX's configuration file and ensure that any preexisting PID file is removed. `ExecStart` is the command that actually starts the service. `ExecReload` tells **systemd** how to make the service reread its configuration file. (**systemd** automatically sets the environment variable MAINPID to the appropriate value.)

See page 94 for more information about signals.

Termination for this service is handled through `KillMode` and `KillSignal`, which tell **systemd** that the service daemon interprets a QUIT signal as an instruction to clean up and exit. The line

```
ExecStop=/bin/kill -s HUP $MAINPID
```

would have essentially the same effect. If the daemon doesn't terminate within `TimeoutStopSec` seconds, **systemd** will force the issue by pelting it with a TERM signal and then an uncatchable KILL signal.

The `PrivateTmp` setting is an attempt at increasing security. It puts the service's **/tmp** directory somewhere other than the actual **/tmp**, which is shared by all the system's processes and users.

Local services and customizations

As you can see from the previous examples, it's relatively trivial to create a unit file for a home-grown service. Browse the examples in **/usr/lib/systemd/system** and adapt one that's close to what you want. See the man page for **systemd.service** for a complete list of configuration options for services. For options common to all types of units, see the page for **systemd.unit**.

Put your new unit file in **/etc/systemd/system**. You can then run

```
$ sudo systemctl enable custom.service
```

to activate the dependencies listed in the service file's `[Install]` section.

As a general rule, you should never edit a unit file you didn't write. Instead, create a configuration directory in **/etc/systemd/system/***unit-file***.d** and add one or more configuration files there called *xxx***.conf**. The *xxx* part doesn't matter; just make sure the file has a **.conf** suffix and is in the right location. **override.conf** is the standard name.

.conf files have the same format as unit files, and in fact **systemd** smooshes them all together with the original unit file. However, override files have priority over the original unit file should both sources try to set the value of a particular option.

One point to keep in mind is that many **systemd** options are allowed to appear more than once in a unit file. In these cases, the multiple values form a list and are all active simultaneously. If you assign a value in your **override.conf** file, that value joins the list but does not replace the existing entries. This may or may not be what you want. To remove the existing entries from a list, just assign the option an empty value before adding your own.

Let's look at an example. Suppose that your site keeps its NGINX configuration file in a nonstandard place, say, **/usr/local/www/nginx.conf**. You need to run the **nginx** daemon with a **-c /usr/local/www/nginx.conf** option so that it can find the proper configuration file.

You can't just add this option to **/usr/lib/systemd/system/nginx.service** because that file will be replaced whenever the NGINX package is updated or refreshed. Instead, you can use the following command sequence:

```
$ sudo mkdir /etc/systemd/system/nginx.service.d
$ sudo cat > !$/override.conf[11]
[Service]
ExecStart=
ExecStart=/usr/sbin/nginx -c /usr/local/www/nginx.conf
<Control-D>
$ sudo systemctl daemon-reload
$ sudo systemctl restart nginx.service
```

The first `ExecStart=` removes the current entry, and the second sets an alternative start command. **systemctl daemon-reload** makes **systemd** re-parse unit files. However, it does not restart daemons automatically, so you'll also need an explicit **systemctl restart** to make the change take effect immediately.

This command sequence is such a common idiom that **systemctl** now implements it directly:

```
$ sudo systemctl edit nginx.service
<edit the override file in the editor>
$ sudo systemctl restart nginx.service
```

As shown, you must still do the **restart** by hand.

One last thing to know about override files is that they can't modify the `[Install]` section of a unit file. Any changes you make are silently ignored. Just add dependencies directly with **systemctl add-wants** or **systemctl add-requires**.

Service and startup control caveats

systemd has many architectural implications, and adopting it is not a simple task for the teams that build Linux distributions. Current releases are mostly Frankenstein systems that adopt much of **systemd** but also retain a few links to the past. Sometimes the holdovers just haven't yet been fully converted. In other cases, various forms of glue have been left behind to facilitate compatibility.

11. The > and **!$** are shell metacharacters. The > redirects output to a file, and the **!$** expands to the last component of the previous command line so that you don't have to retype it. All shells understand this notation. See *Shell basics* starting on page 189 for some other handy features.

Though **systemctl** can and should be used for managing services and daemons, don't be surprised when you run into traditional **init** scripts or their associated helper commands. If you attempt to use **systemctl** to disable the network on a CentOS or Red Hat system, for example, you'll receive the following output:

```
$ sudo systemctl disable network
network.service is not a native service, redirecting to /sbin/chkconfig.
Executing /sbin/chkconfig network off
```

See page 696 for more information about Apache.

Traditional **init** scripts often continue to function on a **systemd** system. For example, an **init** script **/etc/rc.d/init.d/my-old-service** might be automatically mapped to a unit file such as **my-old-service.service** during system initialization or when **systemctl daemon-reload** is run. Apache 2 on Ubuntu 17.04 is a case in point: attempting to disable the **apache2.service** results in the following output:

```
$ sudo systemctl disable apache2
Synchronizing state of apache2.service with SysV service script with
    /lib/systemd/systemd-sysv-install.
Executing: /lib/systemd/systemd-sysv-install disable apache2
```

The end result is what you wanted, but it goes through a rather circuitous route.

Red Hat, and by extension CentOS, still run the **/etc/rc.d/rc.local** script at boot time if you configure it to be executable.[12] In theory, you can use this script to hack in site-specific tweaks or post-boot tasks if desired. (At this point, though, you should really skip the hacks and do things **systemd**'s way by creating an appropriate set of unit files.)

Some Red Hat and CentOS boot chores continue to use config files found in the **/etc/sysconfig** directory. Table 2.8 summarizes these.

Table 2.8 Files and subdirectories of Red Hat's /etc/sysconfig directory

File or directory	Contents
console/	A directory that historically allowed for custom keymapping
crond	Arguments to pass to the **crond** daemon
init	Configuration for handling messages from startup scripts
iptables-config	Loads additional **iptables** modules such as NAT helpers
network-scripts/	Accessory scripts and network config files
nfs	Optional RPC and NFS arguments
ntpd	Command-line options for **ntpd**
selinux	Symlink to **/etc/selinux/config** [a]

a. Sets arguments for SELinux or allows you to disable it altogether; see page 85.

12. A quick **sudo chmod +x /etc/rc.d/rc.local** will ensure that the file is executable.

A couple of the items in Table 2.8 merit additional comment:

- The **network-scripts** directory contains additional material related to network configuration. The only things you might need to change here are the files named **ifcfg-***interface*. For example, **network-scripts/ifcfg-eth0** contains the configuration parameters for the interface eth0. It sets the interface's IP address and networking options. See page 419 for more information about configuring network interfaces.
- The **iptables-config** file doesn't actually allow you to modify the **iptables** (firewall) rules themselves. It just provides a way to load additional modules such as those for network address translation (NAT) if you're going to be forwarding packets or using the system as a router. See page 440 for more information about configuring **iptables**.

systemd logging

Capturing the log messages produced by the kernel has always been something of a challenge. It became even more important with the advent of virtual and cloud-based systems, since it isn't possible to simply stand in front of these systems' consoles and watch what happens. Frequently, crucial diagnostic information was lost to the ether.

systemd alleviates this problem with a universal logging framework that includes all kernel and service messages from early boot to final shutdown. This facility, called the journal, is managed by the **journald** daemon.

System messages captured by **journald** are stored in the **/run** directory. **rsyslog** can process these messages and store them in traditional log files or forward them to a remote syslog server. You can also access the logs directly with the **journalctl** command.

Without arguments, **journalctl** displays all log entries (oldest first):

```
$ journalctl
-- Logs begin at Fri 2016-02-26 15:01:25 UTC, end at Fri 2016-02-26
    15:05:16 UTC. --
Feb 26 15:01:25 ubuntu systemd-journal[285]: Runtime journal is using
    4.6M (max allowed 37.0M, t
Feb 26 15:01:25 ubuntu systemd-journal[285]: Runtime journal is using
    4.6M (max allowed 37.0M, t
Feb 26 15:01:25 ubuntu kernel: Initializing cgroup subsys cpuset
Feb 26 15:01:25 ubuntu kernel: Initializing cgroup subsys cpu
Feb 26 15:01:25 ubuntu kernel: Linux version 3.19.0-43-generic (buildd@
    lcy01-02) (gcc version 4.
Feb 26 15:01:25 ubuntu kernel: Command line: BOOT_IMAGE=/boot/vmlinuz-
    3.19.0-43-generic root=UUI
Feb 26 15:01:25 ubuntu kernel: KERNEL supported cpus:
Feb 26 15:01:25 ubuntu kernel:   Intel GenuineIntel
...
```

You can configure **journald** to retain messages from prior boots. To do this, edit **/etc/systemd/journald.conf** and configure the `Storage` attribute:

```
[Journal]
Storage=persistent
```

Once you've configured **journald**, you can obtain a list of prior boots with

```
$ journalctl --list-boots
-1 a73415fade0e4e7f4bea60913883d180dc Fri 2016-02-26 15:01:25 UTC
    Fri 2016-02-26 15:05:16 UTC
0 0c563fa3830047ecaa2d2b053d4e661d Fri 2016-02-26 15:11:03 UTC  Fri
    2016-02-26 15:12:28 UTC
```

You can then access messages from a prior boot by referring to its index or by naming its long-form ID:

```
$ journalctl -b -1
$ journalctl -b a73415fade0e4e7f4bea60913883d180dc
```

To restrict the logs to those associated with a specific unit, use the **-u** flag:

```
$ journalctl -u ntp
-- Logs begin at Fri 2016-02-26 15:11:03 UTC, end at Fri 2016-02-26
    15:26:07 UTC. --
Feb 26 15:11:07 ub-test-1 systemd[1]: Stopped LSB: Start NTP daemon.
Feb 26 15:11:08 ub-test-1 systemd[1]: Starting LSB: Start NTP daemon...
Feb 26 15:11:08 ub-test-1 ntp[761]: * Starting NTP server ntpd
...
```

System logging is covered in more detail in Chapter 10, *Logging*.

2.8 FreeBSD init and startup scripts

See Chapter 7 for more information about shell scripting.

FreeBSD uses a BSD-style **init**, which does not support the concept of run levels. To bring the system to its fully booted state, FreeBSD's **init** just runs **/etc/rc**. This program is a shell script, but it should not be directly modified. Instead, the **rc** system implements a couple of standardized ways for administrators and software packages to extend the startup system and make configuration changes.

/etc/rc is primarily a wrapper that runs other startup scripts, most of which live in **/usr/local/etc/rc.d**. and **/etc/rc.d**. Before it runs any of those scripts, however, **rc** executes three files that hold configuration information for the system:

- **/etc/defaults/config**
- **/etc/rc.conf**
- **/etc/rc.conf.local**

These files are themselves scripts, but they typically contain only definitions for the values of shell variables. The startup scripts then check these variables to determine

how to behave. (**/etc/rc** uses some shell magic to ensure that the variables defined in these files are visible everywhere.)

/etc/defaults/rc.conf lists all the configuration parameters and their default settings. Never edit this file, lest the startup script bogeyman hunt you down and overwrite your changes the next time the system is updated. Instead, just override the default values by setting them again in **/etc/rc.conf** or **/etc/rc.conf.local**. The **rc.conf** man page has an extensive list of the variables you can specify.

In theory, the **rc.conf** files can also specify other directories in which to look for startup scripts by your setting the value of the `local_startup` variable. The default value is **/usr/local/etc/rc.d**, and we recommend leaving it that way.[13]

As you can see from peeking at **/etc/rc.d**, there are many different startup scripts, more than 150 on a standard installation. **/etc/rc** runs these scripts in the order calculated by the **rcorder** command, which reads the scripts and looks for dependency information that's been encoded in a standard way.

FreeBSD's startup scripts for garden-variety services are fairly straightforward. For example, the top of the **sshd** startup script is as follows:

```
#!/bin/sh
# PROVIDE: sshd
# REQUIRE: LOGIN FILESYSTEMS
# KEYWORD: shutdown
. /etc/rc.subr
name="sshd"
rcvar="sshd_enable"
command="/usr/sbin/${name}"
...
```

The `rcvar` variable contains the name of a variable that's expected to be defined in one of the **rc.conf** scripts, in this case, `sshd_enable`. If you want **sshd** (the real daemon, not the startup script; both are named **sshd**) to run automatically at boot time, put the line

```
sshd_enable="YES"
```

into **/etc/rc.conf**. If this variable is set to `"NO"` or commented out, the **sshd** script will not start the daemon or check to see whether it should be stopped when the system is shut down.

The **service** command provides a real-time interface into FreeBSD's **rc.d** system.[14] To stop the **sshd** service manually, for example, you could run the command

```
$ sudo service sshd stop
```

13. For local customizations, you have the option of either creating standard **rc.d**-style scripts that go in **/usr/local/etc/rc.d** or editing the system-wide **/etc/rc.local** script. The former is preferred.

14. The version of **service** that FreeBSD uses derives from the Linux **service** command, which manipulates traditional **init** services.

Note that this technique works only if the service is enabled in the **/etc/rc.conf** files. If it is not, use the subcommand **onestop**, **onestart**, or **onerestart**, depending on what you want to do. (**service** is generally forgiving and will remind you if need be, however.)

2.9 Reboot and shutdown procedures

Historically, UNIX and Linux machines were touchy about how they were shut down. Modern systems have become less sensitive, especially when a robust filesystem is used, but it's always a good idea to shut down a machine nicely when possible.

Consumer operating systems of yesteryear trained many sysadmins to reboot the system as the first step in debugging any problem. It was an adaptive habit back then, but these days it more commonly wastes time and interrupts service. Focus on identifying the root cause of problems, and you'll probably find yourself rebooting less often.

That said, it's a good idea to reboot after modifying a startup script or making significant configuration changes. This check ensures that the system can boot successfully. If you've introduced a problem but don't discover it until several weeks later, you're unlikely to remember the details of your most recent changes.

Shutting down physical systems

The **halt** command performs the essential duties required for shutting down the system. **halt** logs the shutdown, kills nonessential processes, flushes cached filesystem blocks to disk, and then halts the kernel. On most systems, **halt -p** powers down the system as a final flourish.

reboot is essentially identical to **halt**, but it causes the machine to reboot instead of halting.

The **shutdown** command is a layer over **halt** and **reboot** that provides for scheduled shutdowns and ominous warnings to logged-in users. It dates back to the days of time-sharing systems and is now largely obsolete. **shutdown** does nothing of technical value beyond **halt** or **reboot**, so feel free to ignore it if you don't have multiuser systems.

Shutting down cloud systems

You can halt or restart a cloud system either from within the server (with **halt** or **reboot**, as described in the previous section) or from the cloud provider's web console (or its equivalent API).

Generally speaking, powering down from the cloud console is akin to turning off the power. It's better if the virtual server manages its own shutdown, but feel free to kill a virtual server from the console if it becomes unresponsive. What else can you do?

Either way, make sure you understand what a shutdown means from the perspective of the cloud provider. It would be a shame to destroy your system when all you meant to do was reboot it.

In the AWS universe, the Stop and Reboot operations do what you'd expect. "Terminate" decommissions the instance and removes it from your inventory. If the underlying storage device is set to "delete on termination," not only will your instance be destroyed, but the data on the root disk will also be lost. That's perfectly fine, as long as it's what you expect. You can enable "termination protection" if you consider this a bad thing.

2.10 Stratagems for a nonbooting system

A variety of problems can prevent a system from booting, ranging from faulty devices to kernel upgrades gone wrong. There are three basic approaches to this situation, listed here in rough order of desirability:

- Don't debug; just restore the system to a known-good state.
- Bring the system up just enough to run a shell, and debug interactively.
- Boot a separate system image, mount the sick system's filesystems, and investigate from there.

The first option is the one most commonly used in the cloud, but it can be helpful on physical servers, too, as long as you have access to a recent image of the entire boot disk. If your site does backups by filesystem, a whole-system restore may be more trouble than it's worth. We discuss the whole-system restore option in *Recovery of cloud systems*, which starts on page 62.

The remaining two approaches focus on giving you a way to access the system, identify the underlying issue, and make whatever fix is needed. Booting the ailing system to a shell is by far the preferable option, but problems that occur very early in the boot sequence may stymie this approach.

The "boot to a shell" mode is known generically as single-user mode or rescue mode. Systems that use **systemd** have an even more primitive option available in the form of emergency mode; it's conceptually similar to single-user mode, but does an absolute minimum of preparation before starting a shell.

Because single-user, rescue, and emergency modes don't configure the network or start network-related services, you'll generally need physical access to the console to make use of them. As a result, single-user mode normally isn't available for cloud-hosted systems. We review some options for reviving broken cloud images starting on page 62.

Single-user mode

In single-user mode, also known as **rescue.target** on systems that use **systemd**, only a minimal set of processes, daemons, and services are started. The root filesystem is mounted (as is **/usr**, in most cases), but the network remains uninitialized.

At boot time, you request single-user mode by passing an argument to the kernel, usually **single** or **-s**. You can do this through the boot loader's command-line interface. In some cases, it may be set up for you automatically as a boot menu option.

If the system is already running, you can bring it down to single-user mode with a **shutdown** (FreeBSD), **telinit** (traditional **init**), or **systemctl** (**systemd**) command.

See Chapter 3, for more information about the root account.

Sane systems prompt for the root password before starting the single-user root shell. Unfortunately, this means that it's virtually impossible to reset a forgotten root password through single-user mode. If you need to reset the password, you'll have to access the disk by way of separate boot media.

See Chapter 5 for more information about filesystems and mounting.

From the single-user shell, you can execute commands in much the same way as when logged in on a fully booted system. However, sometimes only the root partition is mounted; you must mount other filesystems manually to use programs that don't live in **/bin**, **/sbin**, or **/etc**.

You can often find pointers to the available filesystems by looking in **/etc/fstab**. Under Linux, you can run **fdisk -l** (lowercase L option) to see a list of the local system's disk partitions. The analogous procedure on FreeBSD is to run **camcontrol devlist** to identify disk devices and then run **fdisk -s** *device* for each disk.

In many single-user environments, the filesystem root directory starts off being mounted read-only. If **/etc** is part of the root filesystem (the usual case), it will be impossible to edit many important configuration files. To fix this problem, you'll have to begin your single-user session by remounting / in read/write mode. Under Linux, the command

```
# mount -o rw,remount /
```

usually does the trick. On FreeBSD systems, the remount option is implicit when you repeat an existing mount, but you'll need to explicitly specify the source device. For example,

```
# mount -o rw /dev/gpt/rootfs /
```

Single-user mode in Red Hat and CentOS is a bit more aggressive than normal. By the time you reach the shell prompt, these systems have tried to mount all local filesystems. Although this default is usually helpful, it can be problematic if you have a sick filesystem. In that case, you can boot to emergency mode by adding **systemd.unit=emergency.target** to the kernel arguments from within the boot loader (usually GRUB). In this mode, no local filesystems are mounted and only a few essential services are started.

The **fsck** command is run during a normal boot to check and repair filesystems. Depending on what filesystem you're using for the root, you may need to run **fsck** manually when you bring the system up in single-user or emergency mode. See page 766 for more details about **fsck**.

Single-user mode is just a waypoint on the normal booting path, so you can terminate the single-user shell with **exit** or <Control-D> to continue with booting. You can also type <Control-D> at the password prompt to bypass single-user mode entirely.

Single-user mode on FreeBSD

FreeBSD includes a single-user option in its boot menu:

```
1. Boot Multi User [Enter]
2. Boot Single User
3. Escape to loader prompt
4. Reboot

Options
5. Kernel: default/kernel (1 of 2)
6. Configure Boot Options...
```

One nice feature of FreeBSD's single-user mode is that it asks you what program to use as the shell. Just press <Enter> for **/bin/sh**.

If you choose option 3, "Escape to loader prompt," you'll drop into a boot-level command-line environment implemented by FreeBSD's final-common-stage boot loader, **loader**.

Single-user mode with GRUB

On systems that use **systemd**, you can boot into rescue mode by appending **systemd.unit=rescue.target** to the end of the existing Linux kernel line. At the GRUB splash screen, highlight your desired kernel and press the "e" key to edit its boot options. Similarly, for emergency mode, use **systemd.unit=emergency.target**.

Here's an example of a typical configuration:

```
linux16 /vmlinuz-3.10.0-229.el7.x86_64 root=/dev/mapper/rhel_rhel-root
    ro crashkernel=auto rd.lvm.lv=rhel_rhel/swap rd.lvm.lv=rhel_rhel/root
    rhgb quiet LANG=en_US.UTF-8 systemd.unit=rescue.target
```

Type <Control-X> to start the system after you've made your changes.

Recovery of cloud systems

See Chapter 9 for a broader introduction to cloud computing.

It's inherent in the nature of cloud systems that you can't hook up a monitor or USB stick when boot problems occur. Cloud providers do what they can to facilitate problem solving, but basic limitations remain.

Backups are important for all systems, but cloud servers are particularly easy to snapshot. Providers charge extra for backups, but they're inexpensive. Be liberal with your snapshots and you'll always have a reasonable system image to fall back on at short notice.

From a philosophical perspective, you're probably doing something wrong if your cloud servers require boot-time debugging. Pets and physical servers receive veterinary care when they're sick, but cattle get euthanized. Your cloud servers are cattle; replace them with known-good copies when they misbehave. Embracing this approach helps you not only avoid critical failures but also facilitates scaling and system migration.

That said, you will inevitably need to attempt to recover cloud systems or drives, so we briefly discuss that process below.

Within AWS, single-user and emergency modes are unavailable. However, EC2 filesystems can be attached to other virtual servers if they're backed by Elastic Block Storage (EBS) devices. This is the default for most EC2 instances, so it's likely that you can use this method if you need to. Conceptually, it's similar to booting from a USB drive so that you can poke around on a physical system's boot disk.

Here's what to do:

1. Launch a new instance in the same availability zone as the instance you're having issues with. Ideally, this recovery instance should be launched from the same base image and should use the same instance type as the sick system.
2. Stop the problem instance. (But be careful not to "terminate" it; that operation deletes the boot disk image.)
3. With the AWS web console or CLI, detach the volume from the problem system and attach the volume to the recovery instance.
4. Log in to the recovery system. Create a mount point and mount the volume, then do whatever's necessary to fix the issue. Then unmount the volume. (Won't unmount? Make sure you're not **cd**'ed there.)
5. In the AWS console, detach the volume from the recovery instance and reattach it to the problem instance. Start the problem instance and hope for the best.

DigitalOcean droplets offer a VNC-enabled console that you can access through the web, although the web app's behavior is a bit wonky on some browsers. DigitalOcean does not afford a way to detach storage devices and migrate them to a

recovery system the way Amazon does. Instead, most system images let you boot from an alternate recovery kernel.[15]

To access the recovery kernel, first power off the droplet and then mount the recovery kernel and reboot. If all went well, the virtual terminal will give you access to a single-user-like mode. More detailed instructions for this process are available at digitalocean.com.

Boot issues within a Google Compute Engine instance should first be investigated by examination of the instance's serial port information:

```
$ gcloud compute instances get-serial-port-output instance
```

The same information is available through GCP web console.

A disk-shuffling process similar to that described above for the Amazon cloud is also available on Google Compute Engine. You use the CLI to remove the disk from the defunct instance and boot a new instance that mounts the disk as an add-on filesystem. You can then run filesystem checks, modify boot parameters, and select a new kernel if necessary. This process is nicely detailed in Google's documentation at cloud.google.com/compute/docs/troubleshooting.

15. The recovery kernel is not available on all modern distributions. If you're running a recent release and the recovery tab tells you that "The kernel for this Droplet is managed internally and cannot be changed from the control panel" you'll need to open a support ticket with DigitalOcean to have them associate your instance with a recovery ISO, allowing you to continue your recovery efforts.

3 Access Control and Rootly Powers

This chapter is about "access control," as opposed to "security," by which we mean that it focuses on the mechanical details of how the kernel and its delegates make security-related decisions. Chapter 27, *Security*, addresses the more general question of how to set up a system or network to minimize the chance of unwelcome access by intruders.

Access control is an area of active research, and it has long been one of the major challenges of operating system design. Over the last decade, UNIX and Linux have seen a Cambrian explosion of new options in this domain. A primary driver of this surge has been the advent of kernel APIs that allow third party modules to augment or replace the traditional UNIX access control system. This modular approach creates a variety of new frontiers; access control is now just as open to change and experimentation as any other aspect of UNIX.

Nevertheless, the traditional system remains the UNIX and Linux standard, and it's adequate for the majority of installations. Even for administrators who want to venture into the new frontier, a thorough grounding in the basics is essential.

3.1 Standard UNIX access control

The standard UNIX access control model has remained largely unchanged for decades. With a few enhancements, it continues to be the default for general-purpose OS distributions. The scheme follows a few basic rules:

- Access control decisions depend on which user is attempting to perform an operation, or in some cases, on that user's membership in a UNIX group.
- Objects (e.g., files and processes) have owners. Owners have broad (but not necessarily unrestricted) control over their objects.
- You own the objects you create.
- The special user account called "root" can act as the owner of any object.
- Only root can perform certain sensitive administrative operations.[1]

Certain system calls (e.g., **settimeofday**) are restricted to root; the implementation simply checks the identity of the current user and rejects the operation if the user is not root. Other system calls (e.g., **kill**) implement different calculations that involve both ownership matching and special provisions for root. Finally, filesystems have their own access control systems, which they implement in cooperation with the kernel's VFS layer. These are generally more elaborate than the access controls found elsewhere in the kernel. For example, filesystems are much more likely to make use of UNIX groups for access control.

See page 130 for more information about device files.

Complicating this picture is that the kernel and the filesystem are intimately intertwined. For example, you control and communicate with most devices through files that represent them in **/dev**. Since device files are filesystem objects, they are subject to filesystem access control semantics. The kernel uses that fact as its primary form of access control for devices.

Filesystem access control

In the standard model, every file has both an owner and a group, sometimes referred to as the "group owner." The owner can set the permissions of the file. In particular, the owner can set them so restrictively that no one else can access it. We talk more about file permissions in Chapter 5, *The Filesystem* (see page 132).

See page 254 for more information about groups.

Although the owner of a file is always a single person, many people can be group owners of the file, as long as they are all part of a single group. Groups are traditionally defined in the **/etc/group** file, but these days group information is often stored in a network database system such as LDAP; see Chapter 17, *Single Sign-On*, for details.

1. Keep in mind that we are here describing the original design of the access control system. These days, not all of these statements remain literally true. For example, a Linux process that bears appropriate capabilities (see page 82) can now perform some operations that were previously restricted to root.

The owner of a file gets to specify what the group owners can do with it. This scheme allows files to be shared among members of the same project.

You can determine the ownerships of a file with **ls -l**:

```
$ ls -l ~garth/todo
-rw-r----- 1 garth  staff   1259 May 29 19:55 /Users/garth/todo
```

This file is owned by user garth and group staff. The letters and dashes in the first column symbolize the permissions on the file; see page 134 for details on how to decode this information. In this case, the codes mean that garth can read or write the file and that members of the staff group can read it.

*See Chapter 8 for more information about the **passwd** and **group** files.*

Both the kernel and the filesystem track owners and groups as numbers rather than as text names. In the most basic case, user identification numbers (UIDs for short) are mapped to usernames in the **/etc/passwd** file, and group identification numbers (GIDs) are mapped to group names in **/etc/group**. (See Chapter 17, *Single Sign-On*, for information about the more sophisticated options.)

The text names that correspond to UIDs and GIDs are defined only for the convenience of the system's human users. When commands such as **ls** should display ownership information in a human-readable format, they must look up each name in the appropriate file or database.

Process ownership

The owner of a process can send the process signals (see page 94) and can also reduce (degrade) the process's scheduling priority. Processes actually have multiple identities associated with them: a real, effective, and saved UID; a real, effective, and saved GID; and under Linux, a "filesystem UID" that is used only to determine file access permissions. Broadly speaking, the real numbers are used for accounting (now largely vestigial), and the effective numbers are used for the determination of access permissions. The real and effective numbers are normally the same.

The saved UID and GID are parking spots for IDs that are not currently in use but that remain available for the process to invoke. The saved IDs allow a program to repeatedly enter and leave a privileged mode of operation; this precaution reduces the risk of unintended misbehavior.

See Chapter 21 for more about NFS.

The filesystem UID is generally explained as an implementation detail of NFS, the Network File System. It is usually the same as the effective UID.

The root account

The root account is UNIX's omnipotent administrative user. It's also known as the superuser account, although the actual username is "root".

The defining characteristic of the root account is its UID of 0. Nothing prevents you from changing the username on this account or from creating additional accounts

whose UIDs are 0; however, these are both bad ideas.[2] Such changes have a tendency to create inadvertent breaches of system security. They also create confusion when other people have to deal with the strange way you've configured your system.

Traditional UNIX allows the superuser (that is, any process for which the effective UID is 0) to perform any valid operation on any file or process.[3]

Some examples of restricted operations are

- Creating device files
- Setting the system clock
- Raising resource usage limits and process priorities
- Setting the system's hostname
- Configuring network interfaces
- Opening privileged network ports (those numbered below 1,024)
- Shutting down the system

An example of superuser powers is the ability of a process owned by root to change its UID and GID. The **login** program and its GUI equivalents are a case in point; the process that prompts you for your password when you log in to the system initially runs as root. If the password and username that you enter are legitimate, the login program changes its UID and GID to your UID and GID and starts up your shell or GUI environment. Once a root process has changed its ownerships to become a normal user process, it can't recover its former privileged state.

Setuid and setgid execution

Traditional UNIX access control is complemented by an identity substitution system that's implemented by the kernel and the filesystem in collaboration. This scheme allows specially marked executable files to run with elevated permissions, usually those of root. It lets developers and administrators set up structured ways for unprivileged users to perform privileged operations.

When the kernel runs an executable file that has its "setuid" or "setgid" permission bits set, it changes the effective UID or GID of the resulting process to the UID or GID of the file containing the program image rather than the UID and GID of the user that ran the command. The user's privileges are thus promoted for the execution of that specific command only.

For example, users must be able to change their passwords. But since passwords are (traditionally) stored in the protected **/etc/master.passwd** or **/etc/shadow** file, users need a setuid **passwd** command to mediate their access. The **passwd** command

2. Jennine Townsend, one of our stalwart technical reviewers, commented, "Such bad ideas that I fear even mentioning them might encourage someone!"
3. "Valid" is the operative word here. Certain operations (such as executing a file on which the execute permission bit is not set) are forbidden even to the superuser.

checks to see who's running it and customizes its behavior accordingly: users can change only their own passwords, but root can change any password.

Programs that run setuid, especially ones that run setuid to root, are prone to security problems. The setuid commands distributed with the system are theoretically secure; however, security holes have been discovered in the past and will undoubtedly be discovered in the future.

The surest way to minimize the number of setuid *problems* is to minimize the number of setuid *programs*. Think twice before installing software that needs to run setuid, and avoid using the setuid facility in your own home-grown software. Never use setuid execution on programs that were not explicitly written with setuid execution in mind.

See page 767 for more information about filesystem mount options.

You can disable setuid and setgid execution on individual filesystems by specifying the **nosuid** option to **mount**. It's a good idea to use this option on filesystems that contain users' home directories or that are mounted from less trustworthy administrative domains.

3.2 Management of the root account

Root access is required for system administration, and it's also a pivot point for system security. Proper husbandry of the root account is a crucial skill.

Root account login

Since root is just another user, most systems let you log in directly to the root account. However, this turns out to be a bad idea, which is why Ubuntu forbids it by default.

To begin with, root logins leave no record of what operations were performed as root. That's bad enough when you realize that you broke something last night at 3:00 a.m. and can't remember what you changed; it's even worse when an access was unauthorized and you are trying to figure out what an intruder has done to your system. Another disadvantage is that the log-in-as-root scenario leaves no record of who was actually doing the work. If several people have access to the root account, you won't be able to tell who used it and when.

For these reasons, most systems allow root logins to be disabled on terminals, through window systems, and across the network—everywhere but on the system console. We suggest that you use these features. See *PAM: cooking spray or authentication wonder?* starting on page 590 to see how to implement this policy on your particular system.

If root does have a password (that is, the root account is not disabled; see page 78), that password must be of high quality. See page 992 for some additional comments regarding password selection.

su: substitute user identity

A marginally better way to access the root account is to use the **su** command. If invoked without arguments, **su** prompts for the root password and then starts up a root shell. Root privileges remain in effect until you terminate the shell by typing <Control-D> or the **exit** command. **su** doesn't record the commands executed as root, but it does create a log entry that states who became root and when.

The **su** command can also substitute identities other than root. Sometimes, the only way to reproduce or debug a user's problem is to **su** to their account so that you reproduce the environment in which the problem occurs.

If you know someone's password, you can access that person's account directly by executing **su** - *username*. As with an **su** to root, you are prompted for the password for *username*. The - (dash) option makes **su** spawn the shell in login mode.

The exact implications of login mode vary by shell, but login mode normally changes the number or identity of the files that the shell reads when it starts up. For example, **bash** reads **~/.bash_profile** in login mode and **~/.bashrc** in nonlogin mode. When diagnosing other users' problems, it helps to reproduce their login environments as closely as possible by running in login mode.

On some systems, the root password allows an **su** or **login** to any account. On others, you must first **su** explicitly to root before **su**ing to another account; root can **su** to any account without entering a password.

Get in the habit of typing the full pathname to **su** (e.g., **/bin/su** or **/usr/bin/su**) rather than relying on the shell to find the command for you. This precaution gives you some protection against arbitrary programs called **su** that might have been sneaked into your search path with the intention of harvesting passwords.[4]

On most systems, you must be a member of the group "wheel" to use **su**.

We consider **su** to have been largely superseded by **sudo**, described in the next section. **su** is best reserved for emergencies. It's also helpful for fixing situations in which **sudo** has been broken or misconfigured.

sudo: limited su

Without one of the advanced access control systems outlined starting on page 83, it's hard to enable someone to do one task (backups, for example) without giving that person free run of the system. And if the root account is used by several administrators, you really have only a vague idea of who's using it or what they've done.

The most widely used solution to these problems is a program called **sudo** that is currently maintained by Todd Miller. It runs on all our example systems and is

4. For the same reason, do not include "." (the current directory) in your shell's search path (which you can see by typing **echo $PATH**). Although convenient, including "." makes it easy to inadvertently run "special" versions of system commands that an intruder has left lying around as a trap. Naturally, this advice goes double for root.

also available in source code form from sudo.ws. We recommend it as the primary method of access to the root account.

sudo takes as its argument a command line to be executed as root (or as another restricted user). **sudo** consults the file **/etc/sudoers** (**/usr/local/etc/sudoers** on FreeBSD), which lists the people who are authorized to use **sudo** and the commands they are allowed to run on each host. If the proposed command is permitted, **sudo** prompts for the *user's own* password and executes the command.

Additional **sudo** commands can be executed without the "doer" having to type a password until a five-minute period (configurable) has elapsed with no further **sudo** activity. This timeout serves as a modest protection against users with **sudo** privileges who leave terminals unattended.

See Chapter 10 for more information about syslog.

sudo keeps a log of the command lines that were executed, the hosts on which they were run, the people who ran them, the directories from which they were run, and the times at which they were invoked. This information can be logged by syslog or placed in the file of your choice. We recommend using syslog to forward the log entries to a secure central host.

A log entry for randy's executing **sudo /bin/cat /etc/sudoers** might look like this:

```
Dec 7 10:57:19 tigger sudo: randy: TTY=ttyp0 ; PWD=/tigger/users/randy;
    USER=root ; COMMAND=/bin/cat /etc/sudoers
```

Example configuration

The **sudoers** file is designed so that a single version can be used on many different hosts at once. Here's a typical example:

```
# Define aliases for machines in CS & Physics departments
Host_Alias  CS = tigger, anchor, piper, moet, sigi
Host_Alias  PHYSICS = eprince, pprince, icarus

# Define collections of commands
Cmnd_Alias  DUMP = /sbin/dump, /sbin/restore
Cmnd_Alias  WATCHDOG = /usr/local/bin/watchdog
Cmnd_Alias  SHELLS = /bin/sh, /bin/dash, /bin/bash

# Permissions
mark, ed    PHYSICS = ALL
herb        CS = /usr/sbin/tcpdump : PHYSICS = (operator) DUMP
lynda       ALL = (ALL) ALL, !SHELLS
%wheel      ALL, !PHYSICS = NOPASSWD: WATCHDOG
```

The first two sets of lines define groups of hosts and commands that are referred to in the permission specifications later in the file. The lists could be included literally in the specifications, but aliases make the **sudoers** file easier to read and understand; they also make the file easier to update in the future. It's also possible to define aliases for sets of users and for sets of users as whom commands may be run.

Each permission specification line includes information about

- The users to whom the line applies
- The hosts on which the line should be heeded
- The commands that the specified users can run
- The users as whom the commands can be executed

The first permission line applies to the users mark and ed on the machines in the `PHYSICS` group (eprince, pprince, and icarus). The built-in command alias `ALL` allows them to run any command. Since no list of users is specified in parentheses, **sudo** will run commands as root.

The second permission line allows herb to run **tcpdump** on `CS` machines and dump-related commands on `PHYSICS` machines. However, the dump commands can be run only as operator, not as root. The actual command line that herb would type would be something like

```
ubuntu$ sudo -u operator /usr/sbin/dump 0u /dev/sda1
```

The user lynda can run commands as any user on any machine, except that she can't run several common shells. Does this mean that lynda really can't get a root shell? Of course not:

```
ubuntu$ cp -p /bin/sh /tmp/sh
ubuntu$ sudo /tmp/sh
```

Generally speaking, any attempt to allow "all commands except …" is doomed to failure, at least in a technical sense. However, it might still be worthwhile to set up the **sudoers** file this way as a reminder that root shells are strongly discouraged.

The final line allows users in group wheel to run the local **watchdog** command as root on all machines except eprince, pprince, and icarus. Furthermore, no password is required to run the command.

Note that commands in the **sudoers** file are specified with full pathnames to prevent people from executing their own programs and scripts as root. Though no examples are shown above, it is possible to specify the arguments that are permissible for each command as well.

To manually modify the **sudoers** file, use the **visudo** command, which checks to be sure no one else is editing the file, invokes an editor on it (**vi**, or whichever editor you specify in your EDITOR environment variable), and then verifies the syntax of the edited file before installing it. This last step is particularly important because an invalid **sudoers** file might prevent you from **sudo**ing again to fix it.

sudo *pros and cons*

The use of **sudo** has the following advantages:

- Accountability is much improved because of command logging.
- Users can do specific chores without having unlimited root privileges.

- The real root password can be known to only one or two people.[5]
- Using **sudo** is faster than using **su** or logging in as root.
- Privileges can be revoked without the need to change the root password.
- A canonical list of all users with root privileges is maintained.
- The chance of a root shell being left unattended is lessened.
- A single file can control access for an entire network.

See page 1000 for more information about password cracking.

sudo has a couple of disadvantages as well. The worst of these is that any breach in the security of a sudoer's personal account can be equivalent to breaching the root account itself. You can't do much to counter this threat other than caution your sudoers to protect their own accounts as they would the root account. You can also run a password cracker regularly on sudoers' passwords to ensure that they are making good password selections. All the comments on password selection from page 992 apply here as well.

sudo's command logging can easily be subverted by tricks such as shell escapes from within an allowed program, or by **sudo sh** and **sudo su**. (Such commands do show up in the logs, so you'll at least know they've been run.)

sudo *vs. advanced access control*

If you think of **sudo** as a way of subdividing the privileges of the root account, it is superior in some ways to many of the drop-in access control systems outlined starting on page 83:

- You decide exactly how privileges will be subdivided. Your division can be coarser or finer than the privileges defined for you by an off-the-shelf system.
- Simple configurations—the most common—are simple to set up, maintain, and understand.
- **sudo** runs on all UNIX and Linux systems. You do need not worry about managing different solutions on different platforms.
- You can share a single configuration file throughout your site.
- You get consistent, high-quality logging for free.

Because the system is vulnerable to catastrophic compromise if the root account is penetrated, a major drawback of **sudo**-based access control is that the potential attack surface expands to include the accounts of all administrators.

sudo works well as a tool for well-intentioned administrators who need general access to root privileges. It's also great for allowing non-administrators to perform a few specific operations. Despite a configuration syntax that suggests otherwise, it is unfortunately not a safe way to define limited domains of autonomy or to place certain operations out of bounds.

5. Or even zero people, if you have the right kind of password vault system in place.

Don't even attempt these configurations. If you need this functionality, you are much better off enabling one of the drop-in access control systems described starting on page 83.

Typical setup

sudo's configuration system has accumulated a lot of features over the years. It has also expanded to accommodate a variety of unusual situations and edge cases. As a result, the current documentation conveys an impression of complexity that isn't necessarily warranted.

Since it's important that **sudo** be reliable and secure, it's natural to wonder if you might be exposing your systems to additional risk if you don't make use of **sudo**'s advanced features and set exactly the right values for all options. The answer is no. 90% of **sudoers** files look something like this:

```
User_Alias  ADMINS = alice, bob, charles
ADMINS      ALL = (ALL) ALL
```

This is a perfectly respectable configuration, and in many cases there's no need to complicate it further. We've mentioned a few extras you can play with in the sections below, but they're all problem-solving tools that are helpful for specific situations. Nothing more is required for general robustness.

Environment management

Many commands consult the values of environment variables and modify their behavior depending on what they find. In the case of commands run as root, this mechanism can be both a useful convenience and a potential route of attack.

For example, several commands run the program specified in your EDITOR environment variable to spawn a text editor. If this variable points to a hacker's malicious program instead of an editor, it's likely that you'll eventually end up running that program as root.[6]

To minimize this risk, **sudo**'s default behavior is to pass only a minimal, sanitized environment to the commands that it runs. If your site needs additional environment variables to be passed, you can whitelist them by adding them to the **sudoers** file's `env_keep` list. For example, the lines

```
Defaults    env_keep += "SSH_AUTH_SOCK"
Defaults    env_keep += "DISPLAY XAUTHORIZATION XAUTHORITY"
```

preserve several environment variables used by X Windows and by SSH key forwarding.

6. Just to be clear, the scenario in this case is that your account has been compromised, but the attacker does not know your actual password and so cannot run **sudo** directly. Unfortunately, this is a common situation—all it takes is a terminal window left momentarily unattended.

It's possible to set different env_keep lists for different users or groups, but the configuration rapidly becomes complicated. We suggest sticking to a single, universal list and being relatively conservative with the exceptions you enshrine in the **sudoers** file.

If you need to preserve an environment variable that isn't listed in the **sudoers** file, you can set it explicitly on the **sudo** command line. For example, the command

```
$ sudo EDITOR=emacs vipw
```

edits the system password file with **emacs**. This feature has some potential restrictions, but they're waived for users who can run ALL commands.

sudo without passwords

It's distressingly common to see **sudo** set up to allow command execution as root without the need to enter a password. Just for reference, that configuration is achieved with the NOPASSWD keyword in the **sudoers** file. For example:

```
ansible    ALL = (ALL) NOPASSWD: ALL        # Don't do this
```

See Chapter 23 for more information about Ansible.

Sometimes this is done out of laziness, but more typically, the underlying need is to allow some type of unattended **sudo** execution. The most common cases are when performing remote configuration through a system such as Ansible, or when running commands out of **cron**.

Needless to say, this configuration is dangerous, so avoid it if you can. At the very least, restrict passwordless execution to a specific set of commands if you can.

See page 591 for more information about PAM configuration.

Another option that works well in the context of remote execution is to replace manually entered passwords with authentication through **ssh-agent** and forwarded SSH keys. You can configure this method of authentication through PAM on the server where **sudo** will actually run.

Most systems don't include the PAM module that implements SSH-based authentication by default, but it is readily available. Look for a pam_ssh_agent_auth package.

SSH key forwarding has its own set of security concerns, but it's certainly an improvement over no authentication at all.

Precedence

A given invocation of **sudo** might potentially be addressed by several entries in the **sudoers** file. For example, consider the following configuration:

```
User_Alias      ADMINS = alice, bob, charles
User_Alias      MYSQL_ADMINS = alice, bob

%wheel          ALL = (ALL) ALL
MYSQL_ADMINS    ALL = (mysql) NOPASSWD: ALL
ADMINS          ALL = (ALL) NOPASSWD: /usr/sbin/logrotate
```

Here, administrators can run the **logrotate** command as any user without supplying a password. MySQL administrators can run any command as mysql without a password. Anyone in the wheel group can run any command under any UID, but must authenticate with a password first.

If user alice is in the wheel group, she is potentially covered by each of the last three lines. How do you know which one will determine **sudo**'s behavior?

The rule is that **sudo** always obeys the *last* matching line, with matching being determined by the entire 4-tuple of user, host, target user, and command. Each of those elements must match the configuration line, or the line is simply ignored.

Therefore, `NOPASSWD` exceptions must follow their more general counterparts, as shown above. If the order of the last three lines were reversed, poor alice would have to type a password no matter what **sudo** command she attempted to run.

sudo *without a control terminal*

In addition to raising the issue of passwordless authentication, unattended execution of **sudo** (e.g., from **cron**) often occurs without a normal control terminal. There's nothing inherently wrong with that, but it's an odd situation that **sudo** can check for and reject if the `requiretty` option is turned on in the **sudoers** file.

This option is not the default from **sudo**'s perspective, but some OS distributions include it in their default **sudoers** files, so it's worth checking for and removing. Look for a line of the form

```
Defaults     requiretty
```

and invert its value:

```
Defaults     !requiretty
```

The `requiretty` option does offer a small amount of symbolic protection against certain attack scenarios. However, it's easy to work around and so offers little real security benefit. In our opinion, `requiretty` should be disabled as a matter of course because it is a common source of problems.

Site-wide ***sudo*** *configuration*

Because the **sudoers** file includes the current host as a matching criterion for configuration lines, you can use a single **sudoers** file throughout an administrative domain (that is, a region of your site in which hostnames and user accounts are guaranteed to be name-equivalent). This approach makes the initial **sudoers** setup a bit more complicated, but it's a great idea, for multiple reasons. You should do it.

The main advantage of this approach is that there's no mystery about who has what permissions on what hosts. Everything is recorded in one authoritative file. When an administrator leaves your organization, for example, there's no need to track