Tools and Techniques for Linux and Unix Administration

3rd Edition
Revised & Updated

Essential

System Administration

O'REILLY®

Æleen Frisch

O'REILLY®

Essential System Administration

Since its first printing in October 1991, *Essential System Administration* has been the definitive practical guide for Unix and Linux system administrators. The book talks about all the usual administrative tools that Unix and Linux provide—and also shows how to use those tools in smarter and more efficient ways.

Author Æleen Frisch expands coverage of networking, electronic mail, security, and kernel configuration—topics of increasing importance to administrators. It also includes coverage of services such as LDAP, PAM, DHCP, and DNS, and discussions of many important open source tools, including SSH, Cfengine, Amanda, RRDTool, and Cricket. The latest versions of all major Unix platforms, including Red Hat Linux 7.3 and SuSE Linux 8, Solaris 8 and 9, FreeBSD 4.6, AIX 5, HP-UX 11 and 11i, and Tru64 5.1, have been thoroughly reviewed and tested.

You will find this book indispensable whether you are responsible for a large, shared computer system or a network of workstations, or you use a standalone Unix or Linux system and have found that the fine line between a user and an administrator has vanished. And even if you aren't directly or solely responsible for system administration, you'll find that understanding important administrative functions will greatly increase your ability to use Unix effectively.

"Æleen is a master at teaching system administration. The third edition of Essential System Administration *covers the bases of administering the many flavors of Unix and Linux. If your site is one of the many non-vanilla shops, you really need this book."*

—Tina Darmohray, Information Warehouse, Inc.

"This book is 'essential' in more ways than one. For system administrators looking to bring Linux into a Unix environment (or to migrate from Unix to Linux), the details are all there. It is equally a great book for those thinking about becoming system administrators. But perhaps most impressively, it is an accessible book for people who use Unix or Linux and want to learn how to get the most out of their system. A true example of what great technical writing can be, and the standard that O'Reilly sets as a publisher."

—Michael Tiemann, Chief Information Officer, Red Hat, Inc.

"This update of a classic is a great introduction to the Unix attitude, the system administrator's attitude, and the practical details of managing Unix systems of a wide variety of types. It's a great introduction if you're new to Unix or to system administration, and a great reference if you've been around a while and just want to look up a few details."

—Elizabeth Zwicky, Consultant and former president of SAGE

www.oreilly.com

US $54.95 **CAN $85.95**

ISBN-10: 0-596-00343-9

ISBN-13: 978-0-596-00343-2

9 780596 003432 55495

Essential System Administration

THIRD EDITION

Essential System Administration

Æleen Frisch

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Essential System Administration, Third Edition
by Æleen Frisch

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media, Inc. books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor:	Michael Loukides
Production Editor:	Leanne Clarke Soylemez
Cover Designer:	Edie Freedman
Interior Designer:	David Futato

Printing History:

August 2002:	Third Edition.
September 1995:	Second Edition.
October 1991:	First Edition.

Library of Congress Cataloging-in-Publication Data

Frisch, AEleen
Essential System Administration/by AEleen Frisch.--3rd ed.
p. cm.
Includes index.
ISBN 0-596-00343-9
ISBN13 978-0-596-00343-2
1. UNIX (Computer file) 2. Operating systems (Computers) I. Title.

QA76.76.063 F75 2002
005.4'32--dc21 2002023321

[M] [05/07]

For Frank Willison

"Part of the problem is passive-aggressive behavior, my pet peeve and bête noire, and I don't like it either. Everyone should get off their high horse, particularly if that horse is my bête noire. We all have pressures on us, and nobody's pressure is more important than anyone else's."

***

"Thanks also for not lending others your O'Reilly books. Let others buy them. Buyers respect their books. You seem to recognize that 'lend' and 'lose' are synonyms where books are concerned. If I had been prudent like you, I would still have Volume 3 (Cats–Dorc) of the Encyclopedia Britannica."

Table of Contents

Preface . . . . . **xi**

1. Introduction to System Administration **1**
Thinking About System Administration 3
Becoming Superuser 6
Communicating with Users 12
About Menus and GUIs 14
Where Does the Time Go? 31

2. The Unix Way **32**
Files 33
Processes 53
Devices 61

3. Essential Administrative Tools and Techniques **74**
Getting the Most from Common Commands 74
Essential Administrative Techniques 90

4. Startup and Shutdown **127**
About the Unix Boot Process 127
Initialization Files and Boot Scripts 151
Shutting Down a Unix System 169
Troubleshooting: Handling Crashes and Boot Failures 173

5. TCP/IP Networking **180**
Understanding TCP/IP Networking 180
Adding a New Network Host 202
Network Testing and Troubleshooting 219

6. Managing Users and Groups 222
Unix Users and Groups 222
Managing User Accounts 237
Administrative Tools for Managing User Accounts 256
Administering User Passwords 277
User Authentication with PAM 302
LDAP: Using a Directory Service for User Authentication 313

7. Security 330
Prelude: What's Wrong with This Picture? 331
Thinking About Security 332
User Authentication Revisited 339
Protecting Files and the Filesystem 348
Role-Based Access Control 366
Network Security 373
Hardening Unix Systems 387
Detecting Problems 391

8. Managing Network Services 414
Managing DNS Servers 414
Routing Daemons 452
Configuring a DHCP Server 457
Time Synchronization with NTP 469
Managing Network Daemons under AIX 475
Monitoring the Network 475

9. Electronic Mail 521
About Electronic Mail 521
Configuring User Mail Programs 532
Configuring Access Agents 537
Configuring the Transport Agent 542
Retrieving Mail Messages 596
Mail Filtering with procmail 599
A Few Final Tools 614

10. Filesystems and Disks 616
Filesystem Types 617
Managing Filesystems 621

From Disks to Filesystems 634
Sharing Filesystems 694

11. Backup and Restore 707
Planning for Disasters and Everyday Needs 707
Backup Media 717
Backing Up Files and Filesystems 726
Restoring Files from Backups 736
Making Table of Contents Files 742
Network Backup Systems 744
Backing Up and Restoring the System Filesystems 759

12. Serial Lines and Devices 766
About Serial Lines 766
Specifying Terminal Characteristics 769
Adding a New Serial Device 776
Troubleshooting Terminal Problems 794
Controlling Access to Serial Lines 796
HP-UX and Tru64 Terminal Line Attributes 797
The HylaFAX Fax Service 799
USB Devices 807

13. Printers and the Spooling Subsystem 814
The BSD Spooling Facility 818
System V Printing 829
The AIX Spooling Facility 848
Troubleshooting Printers 858
Sharing Printers with Windows Systems 860
LPRng 864
CUPS 874
Font Management Under X 878

14. Automating Administrative Tasks 885
Creating Effective Shell Scripts 886
Perl: An Alternate Administrative Language 899
Expect: Automating Interactive Programs 911
When Only C Will Do 919
Automating Complex Configuration Tasks with Cfengine 921

Stem: Simplified Creation of Client-Server Applications 932
Adding Local man Pages 942

15. Managing System Resources 945
Thinking About System Performance 945
Monitoring and Controlling Processes 951
Managing CPU Resources 963
Managing Memory 978
Disk I/O Performance Issues 1001
Monitoring and Managing Disk Space Usage 1007
Network Performance 1017

16. Configuring and Building Kernels 1024
FreeBSD and Tru64 1026
HP-UX 1031
Linux 1033
Solaris 1046
AIX System Parameters 1047

17. Accounting 1049
Standard Accounting Files 1051
BSD-Style Accounting: FreeBSD, Linux, and AIX 1052
System V–Style Accounting: AIX, HP-UX, and Solaris 1058
Printing Accounting 1066

Afterword: The Profession of System Administration 1069
SAGE: The System Administrators Guild 1069
Administrative Virtues 1070

Appendix: Administrative Shell Programming 1073

Index 1097

Preface

This book is an agglomeration of lean-tos and annexes and there is no knowing how big the next addition will be, or where it will be put. At any point, I can call the book finished or unfinished.
—Alexander Solzhenitsyn

A poem is never finished, only abandoned.
—Paul Valery

This book covers the fundamental and essential tasks of Unix system administration. Although it includes information designed for people new to system administration, its contents extend well beyond the basics. The primary goal of this book is to make system administration on Unix systems straightforward; it does so by providing you with exactly the information you need. As I see it, this means finding a middle ground between a general overview that is too simple to be of much use to anyone but a complete novice, and a slog through all the obscurities and eccentricities that only a fanatic could love (some books actually suffer from both these conditions at the same time). In other words, I won't leave you hanging when the first complication arrives, and I also won't make you wade through a lot of extraneous information to find what actually matters.

This book approaches system administration from a task-oriented perspective, so it is organized around various facets of the system administrator's job, rather than around the features of the Unix operating system, or the workings of the hardware subsystems in a typical system, or some designated group of administrative commands. These are the raw materials and tools of system administration, but an effective administrator has to know when and how to apply and deploy them. You need to have the ability, for example, to move from a user's complaint ("This job only needs 10 minutes of CPU time, but it takes it three hours to get it!") through a diagnosis of the problem ("The system is thrashing because there isn't enough swap space"), to the particular command that will solve it (`swap` or `swapon`). Accordingly, this book covers all facets of Unix system administration: the general concepts,

underlying structure, and guiding assumptions that define the Unix environment, as well as the commands, procedures, strategies, and policies essential to success as a system administrator. It will talk about all the usual administrative tools that Unix provides and also how to use them more smartly and efficiently.

Naturally, some of this information will constitute advice about system administration; I won't be shy about letting you know what my opinion is. But I'm actually much more interested in giving you the information you need to make informed decisions for your own situation than in providing a single, univocal view of the "right way" to administer a Unix system. It's more important that you know what the issues are concerning, say, system backups, than that you adopt anyone's specific philosophy or scheme. When you are familiar with the problem and the potential approaches to it, you'll be in a position to decide for yourself what's right for your system.

Although this book will be useful to anyone who takes care of a Unix system, I have also included some material designed especially for system administration professionals. Another way that this book covers essential system administration is that it tries to convey the essence of what system administration is, as well as a way of approaching it when it is your job or a significant part thereof. This encompasses intangibles such as system administration as a profession, professionalism (not the same thing), human and humane factors inherent in system administration, and its relationship to the world at large. When such issues are directly relevant to the primary, technical content of the book, I mention them. In addition, I've included other information of this sort in special sidebars (the first one comes later in this Preface). They are designed to be informative and thought-provoking and are, on occasion, deliberately provocative.

The Unix Universe

More and more, people find themselves taking care of multiple computers, often from more than one manufacturer; it's quite rare to find a system administrator who is responsible for only one system (unless he has other, unrelated duties as well). While Unix is widely lauded in marketing brochures as the "standard" operating system "from microcomputers to supercomputers"—and I must confess to having written a few of those brochures myself—this is not at all the same as there being a "standard" Unix.At this point, Unix is hopelessly plural, and nowhere is this plurality more evident than in system administration. Before going on to discuss how this book addresses that fact, let's take a brief look at how things got to be the way they are now.

Figure P-1 attempts to capture the main flow of Unix development. It illustrates a simplified Unix genealogy, with an emphasis on influences and family relationships (albeit Faulknerian ones) rather than on strict chronology and historical accuracy. It

traces the major lines of descent from an arbitrary point in time: Unix Version 6 in 1975 (note that the dates in the diagram refer to the earliest manifestation of each version). Over time, two distinct flavors (strains) of Unix emerged from its beginnings at AT&T Bell Laboratories—which I'll refer to as System V and BSD—but there was also considerable cross-influence between them (in fact, a more detailed diagram would indicate this even more clearly).

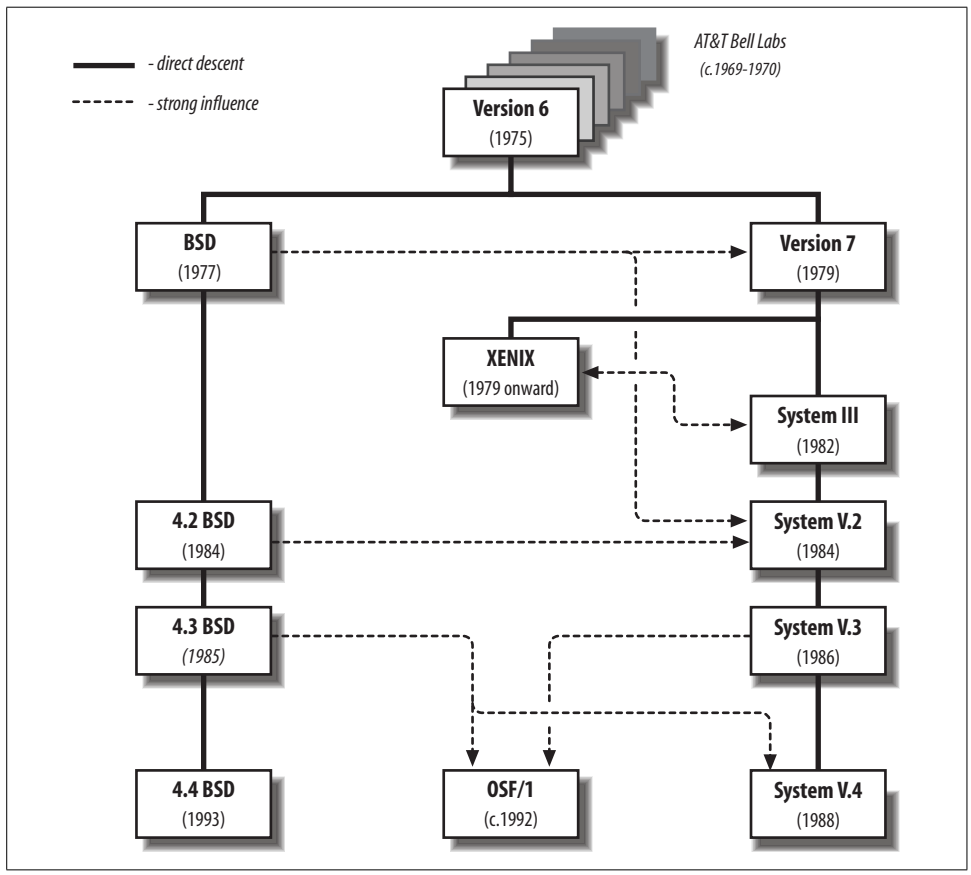


Figure P-1. Unix genealogy (simplified)

For a Unix family tree at the other extreme of detail, see *http://perso.wanadoo.fr/levenez/unix/*. Also, the opening chapters of *Life with UNIX*, by Don Libes and Sandy Ressler (PTR Prentice Hall), give a very entertaining overview of the history of Unix. For a more detailed written history, see *A Quarter Century of UNIX* by Peter Salus (Addison-Wesley).

The split we see today between System V and BSD occurred after Version 6.* developers at the University of California, Berkeley, extended Unix in many ways, adding virtual memory support, the C shell, job control, and TCP/IP networking, to name just a few. Some of these contributions were merged into the AT&T code lines at various points.

System V Release 4 was often described as a merger of the System V and BSD lines, but this is not quite accurate. It incorporated the most important features of BSD (and SunOS) into System V. The union was a marriage and not a merger, however, with some but not all characteristics from each parent dominant in the offspring (as well as a few whose origins no one is quite sure of).

The diagram also includes OSF/1.

In 1988, Sun and AT&T agreed to jointly develop future versions of System V. In response, IBM, DEC, Hewlett-Packard, and other computer and computer-related companies and organizations formed the Open Software Foundation (OSF), designing it with the explicit goal of producing an alternative, compatible, non-AT&T-dependent, Unix-like operating system. OSF/1 is the result of this effort (although its importance is more as a standards definition than as an actual operating system implementation).

The proliferation of new computer companies throughout the 1980s brought dozens of new Unix systems to market—Unix was usually chosen as much for its low cost and lack of serious alternatives as for its technical characteristics—and also as many variants. These vendors tended to start with some version of System V or BSD and then make small to extensive modifications and customizations. Extant operating systems mostly spring from System V Release 3 (usually Release 3.2), System V Release 4, and occasionally 4.2 or 4.3 BSD (SunOS is the major exception, derived from an earlier BSD version). As a further complication, many vendors freely intermixed System V and BSD features within a single operating system.

Recent years have seen a number of efforts at standardizing Unix. Competition has shifted from acrimonious lawsuits and countersuits to surface-level cooperation in unifying the various versions. However, existing standards simply don't address system administration at anything beyond the most superficial level. Since vendors are free to do as they please in the absence of a standard, there is no guarantee that

* The movement from Version 7 to System III in the System V line is a simplification of strict chronology and descent. System III was derived from an intermediate release between Version 6 and Version 7 (CB Unix), and not every Version 7 feature was included in System III. A word about nomenclature: The successive releases of Unix from the research group at Bell Labs were originally known as "editions"—the Sixth Edition, for example—although these versions are now generally referred to as "Versions." After Version 6, there are two distinct sets of releases from Bell Labs: Versions 7 and following (constituting the original research line), and System III through System V (commercial implementations started from this line). Later versions of System V are called "Releases," as in System V Release 3 and System V Release 4.

system administrative commands and procedures will even be similar under different operating systems that uphold the same set of standards.

Unix Versions Discussed in This Book

How do you make sense out of the myriad of Unix variations? One approach is to use computer systems only from a single vendor. However, since that often has other disadvantages, most of us end up having to deal with more than one kind of Unix system. Fortunately, taking care of *n* different kinds of systems doesn't mean that you have to learn as many different administrative command sets and approaches. Ultimately, we get back to the fact that there are really just two distinct Unix varieties; it's just that the features of any specific Unix implementation can be an arbitrary mixture of System V and BSD features (regardless of its history and origins). This doesn't always ensure that there are only two different commands to perform the same administrative function—there are cases where practically every vendor uses a different one—but it does mean that there are generally just two different approaches to the area or issue. And once you understand the underlying structure, philosophy, and assumptions, learning the specific commands for any given system is simple.

When you recognize and take advantage of this fact, juggling several Unix versions becomes straightforward rather than impossibly difficult. In reality, lots of people do it every day, and this book is designed to reflect that and to support them. It will also make administering heterogeneous environments even easier by systematically providing information about different systems all in one place.

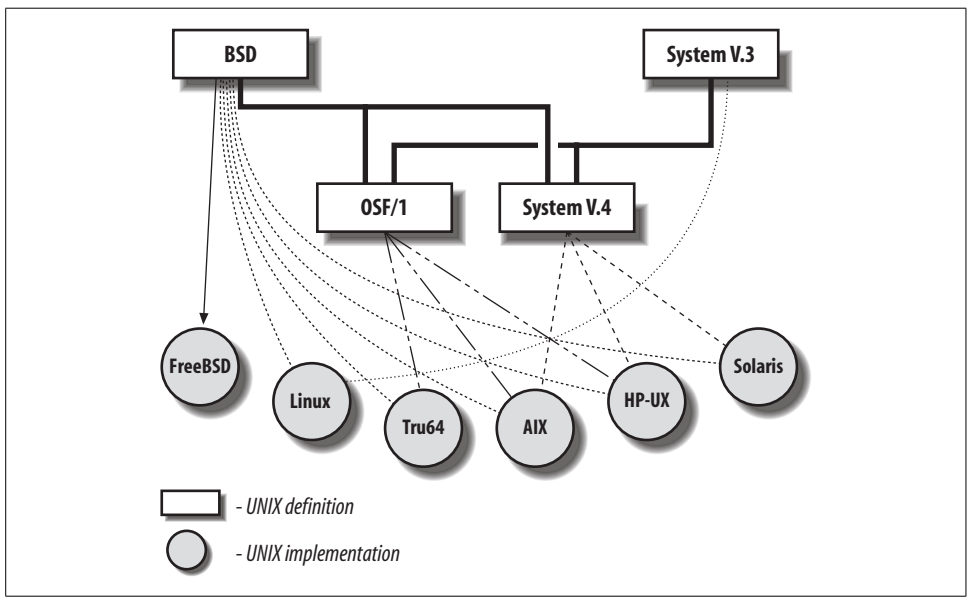


Figure P-2. Unix versions discussed in this book

The Unix versions covered by this book appear in Figure P-2, which illustrates the influences on the various operating systems, rather than their actual origins. If the version on your system isn't one of them, don't despair. Read on anyway, and you'll find that the general information given here applies to your system as well in most cases.

The specific operating system levels covered in this book are:

- AIX Version 5.1
- FreeBSD Version 4.6 (with a few glances at the upcoming Version 5)
- HP-UX Version 11 (including many Version 11i features)
- Linux: Red Hat Version 7.3 and SuSE Version 8
- Solaris Versions 8 and 9
- Tru64 Version 5.1

This list represents some changes from the second edition of this book. We've dropped SCO Unix and IRIX and added FreeBSD. I decided to retain Tru64 despite the recent merger of Compaq and Hewlett-Packard, because it's likely that some Tru64 features will eventually make their way into future HP-UX versions.

When there are significant differences between versions, I've made extensive use of headers and other devices to indicate which version is being considered. You'll find it easy to keep track of where we are at any given point and even easier to find out the specific information you need for whatever version you're interested in. In addition, the book will continue to be useful to you when you get your next, different Unix system—and sooner or later, you will.

The book also covers a fair amount of free software that is not an official part of any version of Unix. In general, the packages discussed can be built for any of the discussed operating systems.

Audience

This book will be of interest to:

- Full or part-time administrators of Unix computer systems. The book includes help both for Unix users who are new to system administration and for experienced system administrators who are new to Unix.
- Workstation and microcomputer users. For small, standalone systems, there is often no distinction between the user and the system administrator. And even if your workstation is part of a larger network with a designated administrator, in practice, many system management tasks for your workstation will be left to you.
- Users of Unix systems who are not full-time system managers but who perform administrative tasks periodically.

Why Vendors Like Standards

Standards are supposed to help computer users by minimizing the differences between products from different vendors and ensuring that such products will successfully work together. However, standards have become a weapon in the competitive arsenal of computer-related companies, and vendor product literature and presentations are often a cacophony of acronyms. Warfare imagery dominates discussions comparing standards compliance rates for different products.

For vendors of computer-related products, upholding standards is in large part motivated by the desire to create a competitive advantage. There is nothing wrong with that, but it's important not to mistake it for the altruism that it is often purported to be. "Proprietary" is a dirty word these days, and "open systems" are all the rage, but that doesn't mean that what's going on is anything other than business as usual.

Proprietary features are now called "extensions" and "enhancements," and defining new standards has become a site of competition. New standards are frequently created by starting from one of the existing alternatives, vendors are always ready to argue for the one they developed, and successful attempts are then touted as further evidence of their product's superiority (and occasionally they really are).

Given all of this, though, we have to at least suspect that it is not really in most vendors' interest for the standards definition process to ever stop.

This book assumes that you are familiar with Unix user commands: that you know how to change the current directory, get directory listings, search files for strings, edit files, use I/O redirection and pipes, set environment variables, and so on. It also assumes a very basic knowledge of shell scripts: you should know what a shell script is, how to execute one, and be able to recognize commonly used features like if statements and comment characters. If you need help at this level, consult *Learning the UNIX Operating System*, by Grace Todino-Gonguet, John Strang, and Jerry Peek, and the relevant editions of *UNIX in a Nutshell* (both published by O'Reilly & Associates).

If you have previous Unix experience but no administrative experience, several sections in Chapter 1 will show you how to make the transition from user to system manager. If you have some system administration experience but are new to Unix, Chapter 2 will explain the Unix approach to major system management tasks; it will also be helpful to current Unix users who are unfamiliar with Unix file, process, or device concepts.

This book is not designed for people who are already Unix wizards. Accordingly, it stays away from topics like writing device drivers.

Organization

This book is the foundation volume for O'Reilly & Associates' system administration series. As such, it provides you with the fundamental information needed by everyone who takes care of Unix systems. At the same time, it consciously avoids trying to be all things to all people; the other books in the series treat individual topics in complete detail. Thus, you can expect this book to provide you with the essentials for all major administrative tasks by discussing both the underlying high-level concepts and the details of the procedures needed to carry them out. It will also tell you where to get additional information as your needs become more highly specialized.

These are the major changes in content with respect to the second edition (in addition to updating all material to the most recent versions of the various operating systems):

- Greatly expanded networking coverage, especially of network server administration, including DHCP, DNS (BIND 8 and 9), NTP, network monitoring with SNMP, and network performance tuning.
- Comprehensive coverage of email administration, including discussions of sendmail, Postfix, procmail, and setting up POP3 and IMAP.
- Additional security topics and techniques, including the secure shell (`ssh`), one-time passwords, role-based access control (RBAC), `chroot` jails and sandboxing, and techniques for hardening Unix systems.
- Discussions of important new facilities that have emerged in the time since the second edition. The most important of these are LDAP, PAM, and advanced file-system features such as logical volume managers and fault tolerance features.
- Overviews and examples of some new scripting and automation tools, specifically Cfengine and Stem.
- Information about device types that have become available or common on Unix systems relatively recently, including USB devices and DVD drives.
- Important open source packages are covered, including the following additions: Samba (for file and printer sharing with Windows systems), the Amanda enterprise backup system, modern printing subsystems (LPRng and CUPS), font management, file and electronic mail encryption and digital signing (PGP and GnuPG), the HylaFAX fax service, network monitoring tools (including RRDTool, Cricket and NetSaint), and the GRUB boot loader.

Chapter Descriptions

The first three chapters of the book provide some essential background material required by different types of readers. The remaining chapters generally focus on a single administrative area of concern and discuss various aspects of everyday system operation and configuration issues.

Chapter 1, *Introduction to System Administration*, describes some general principles of system administration and the *root* account. By the end of this chapter, you'll be thinking like a system administrator.

Chapter 2, *The Unix Way*, considers the ways that Unix structure and philosophy affect system administration. It opens with a description of the man online help facility and then goes on to discuss how Unix approaches various operating system functions, including file ownership, privilege, and protection; process creation and control; and device handling. This chapter closes with an overview of the Unix system directory structure and important configuration files.

Chapter 3, *Essential Administrative Tools and Techniques*, discusses the administrative uses of Unix commands and capabilities. It also provides approaches to several common administrative tasks. It concludes with a discussion of the cron and syslog facilities and package management systems.

Chapter 4, *Startup and Shutdown*, describes how to boot up and shut down Unix systems. It also considers Unix boot scripts in detail, including how to modify them for the needs of your system. It closes with information about how to troubleshoot booting problems.

Chapter 5, *TCP/IP Networking*, provides an overview of TCP/IP networking on Unix systems. It focuses on fundamental concepts and configuring TCP/IP client systems, including interface configuration, name resolution, routing, and automatic IP address assignment with DHCP. The chapter concludes with a discussion of network troubleshooting.

Chapter 6, *Managing Users and Groups*, details how to add new users to a Unix system. It also discusses Unix login initialization files and groups. It covers user authentication in detail, including both traditional passwords and newer authentication facilities like PAM. The chapter also contains information about using LDAP for user account data.

Chapter 7, *Security*, provides an overview of Unix security issues and solutions to common problems, including how to use Unix groups to allow users to share files and other system resources while maintaining a secure environment. It also discusses optional security-related facilities such as dialup passwords and secondary authentication programs. The chapter also covers the more advanced security configuration available by using access control lists (ACLs) and role-based access control (RBAC). It also discusses the process of hardening Unix systems. In reality, though, security is something that is integral to every aspect of system administration, and a good administrator consciously considers the security implications of every action and decision. Thus, expecting to be able to isolate and abstract security into a separate chapter is unrealistic, and so you will find discussion of security-related issues and topics in every chapter of the book.

Chapter 8, *Managing Network Services*, returns to the topic of networking. It discusses configuring and managing various networking daemons, including those for

DNS, DHCP, routing, and NTP. It also contains a discussion of network monitoring and management tools, including the SNMP protocol and tools, Netsaint, RRDTool, and Cricket.

Chapter 9, *Electronic Mail*, covers all aspects of managing the email subsystem. It covers user mail programs, configuring the POP3 and IMAP protocols, the sendmail and Postfix mail transport agents, and the procmail and fetchmail facilities.

Chapter 10, *Filesystems and Disks*, discusses how discrete disk partitions become part of a Unix filesystem. It begins by describing the disk mounting commands and filesystem configuration files. It also considers Unix disk partitioning schemes and describes how to add a new disk to a Unix system. In addition, advanced features such as logical volume managers and software striping and RAID are covered. It also discusses sharing files with remote Unix and Windows systems using NFS and Samba.

Chapter 11, *Backup and Restore*, begins by considering several possible backup strategies before going on to discuss the various backup and restore services that Unix provides. It also covers the open source Amanda backup facility.

Chapter 12, *Serial Lines and Devices*, discusses Unix handling of serial lines, including how to add and configure new serial devices. It covers both traditional serial lines and USB devices. It also includes a discussion of the HylaFAX fax service.

Chapter 13, *Printers and the Spooling Subsystem*, covers printing on Unix systems, including both day-to-day operations and configuration issues. Remote printing via a local area network is also discussed. Printing using open source spooling systems is also covered, via Samba, LPRng, and CUPS.

Chapter 14, *Automating Administrative Tasks*, considers Unix shell scripts, scripts, and programs in other languages and environments such as Perl, C, Expect, and Stem. It provides advice about script design and discusses techniques for testing and debugging them. It also covers the Cfengine facility, which provides high level automation features to system administrators.

Chapter 15, *Managing System Resources*, provides an introduction to performance issues on Unix systems. It discusses monitoring and managing use of major system resources: CPU, memory, and disk. It covers controlling process execution, optimizing memory performance and managing system paging space, and tracking and apportioning disk usage. It concludes with a discussion of network performance monitoring and tuning.

Chapter 16, *Configuring and Building Kernels*, discusses when and how to create a customized kernel, as well as related system configuration issues. It also discusses how to view and modify tunable kernel parameters.

Chapter 17, *Accounting*, describes the various Unix accounting services, including printer accounting.

The Appendix covers the most important Bourne shell and `bash` features.

The Afterword contains some final thoughts on system administration and information about the System Administrator's Guild (SAGE).

Conventions Used in This Book

The following typographic and usage conventions are used in this book:

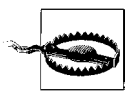
italic
: Used for filenames, directory names, hostnames, and URLs. Also used liberally for annotations in configuration file examples.

`constant width`
: Used for names of commands, utilities, daemons, and other options. Also used in code and configuration file examples.

`constant width italic`
: Used to indicate variables in code.

`constant width bold`
: Used to indicate user input on a command line.

`constant width bold italic`
: Used to indicate variables in command-line user input.


Indicates a warning.


Indicates a note.


Indicates a tip.

he, she
: This book is meant to be straightforward and to the point. There are times when using a third-person pronoun is just the best way to say something: "This setting will force the user to change his password the next time he logs in." Personally, I don't like always using "he" in such situations, and I abhor "he or she" and "s/he," so I use "he" some of the time and "she" some of the time, alternating semi-randomly. However, when the text refers to one of the example users who appear from time to time throughout the book, the appropriate pronoun is always used.

Comments and Questions

Please address comments and questions concerning this book to the publisher:

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

There is a web page for this book, which lists errata, examples, or any additional information. You can access this page at:

http://www.oreilly.com/catalog/esa3/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about books, conferences, Resource Centers, and the O'Reilly Network, see the O'Reilly web site at:

http://www.oreilly.com

Acknowledgments

Many people have helped this book at various points in its successive incarnations. In writing this third edition, I'm afraid I fell at times into the omnipresent trap of writing a different book rather than revising the one at hand; although this made the book take longer to finish, I hope that readers will benefit from my rethinking many topics and issues.

I am certain that few writers have been as fortunate as I have in the truly first-rate set of technical reviewers who read and critiqued the manuscript of the third edition. They were, without doubt, the most meticulous group I have ever encountered:

- Jon Forrest
- Peter Jeremy
- Jay Kreibich
- David Malone
- Eric Melander
- Jay Migliaccio
- Jay Nelson
- Christian Pruett
- Eric Stahl

Luke Boyett, Peter Norton and Nate Williams also commented on significant amounts of the present edition.

My thanks go also to the technical reviews of the first two editions. The second edition reviewers were Nora Chuang, Clem Cole, Walt Daniels, Drew Eckhardt, Zenon Fortuna, Russell Heise, Tanya Herlick, Karen Kerschen, Tom Madell, Hanna Nelson, Barry Saad, Pamela Sogard, Jaime Vazquez, and Dave Williams; first edition reviewers were Jim Binkley, Tan Bronson, Clem Cole, Dick Dunn, Laura Hook, Mike Loukides, and Tim O'Reilly. This book still benefits from their comments.

Many other people helped this edition along by pointing out bugs and providing important information at key points: Jeff Andersen, John Andrea, Jay Ashworth, Christoph Badura, Jiten Bardwaj, Clive Blackledge, Mark Burgess, Trevor Chandler, Douglas Clark, Joseph C. Davidson, Jim Davis, Steven Dick, Matt Eakle, Doug Edwards, Ed Flinn, Patrice Fournier, Rich Fuchs, Brian Gallagher, Michael Gerth, Adam Goodman, Charles Gordon, Uri Guttman, Enhua He, Matthias Heidbrink, Matthew A. Hennessy, Derek Hilliker, John Hobson, Lee Howard, Colin Douglas Howell, Hugh Kennedy, Jonathan C. Knowles, Ki Hwan Lee, Tom Madell, Sean Maguire, Steven Matheson, Jim McKinstry, Barnabus Misanik, John Montgomery, Robert L. Montgomery, Dervi Morgan, John Mulshine, John Mulshine, Darren Nickerson, Jeff Okimoto, Guilio Orsero, Jerry Peek, Chad Pelander, David B. Perry, Tim Rice, Mark Ritchie, Michael Saunby, Carl Schelin, Mark Summerfield, Tetsuji Tanigawa, Chuck Toporek, Gary Trucks, Sean Wang, Brian Whitehead, Bill Wisniewski, Simon Wright, and Michael Zehe.

Any errors that remain are mine alone.

I am also grateful to companies who loaned me or provided access to hardware and/or software:

- Gaussian, Inc. gave me access to several computer systems. Thanks to Mike Frisch, Jim Cheeseman, Jim Hess, John Montgomery, Thom Vreven and Gary Trucks.
- Christopher Mahmood and Jay Migliaccio of SuSE, Inc. gave me advance access to SuSE 8.
- Lorien Golarski of Red Hat gave me access to their beta program.
- Chris Molnar provided me with an advance copy of KDE version 3.
- Angela Loh of Compaq arranged for an equipment loan of an Alpha Linux system.
- Steve Behling, Tony Perraglia and Carlos Sosa of IBM expedited AIX releases for me and also provided useful information.
- Adam Goodman and the staff of *Linux Magazine* provided feedback on early versions of some sections of this book. Thanks also for their long suffering patience with my habitual lateness.

I'd also like to thank my stellar assistant Cat Dubail for all of her help on this third edition. Felicia Bear also provided important editorial help. Thanks also to Laura Lasala, my copy editor for the second edition.

At O'Reilly & Associates, my deepest gratitude goes to my amazing editor Mike Loukides, whose support and guidance brought this edition to completion. Bob Woodbury and Besty Waliszewski provided advice and help at key points. Darren Kelly helped with some technical issues regarding the index. Finally, my enthusiastic thanks go to the excellent production group at O'Reilly & Associates for putting the finishing touches on all three editions of this book.

Finally, no one finishes a task of this size without a lot of support and encouragement from their friends. I'd like to especially thank Mike and Mo for being there for me throughout this project. Thanks also to the furry Frischs: Daphne, Susan, Lyta, and Talia.

—ÆF; Day 200 of 2002; North Haven, CT, USA

CHAPTER 1

Introduction to System Administration

The traditional way to begin a book like this is to provide a list of system administration tasks—I've done it several times myself at this point. Nevertheless, it's important to remember that you have to take such lists with a grain of salt. Inevitably, they leave out many intangibles, the sorts of things that require lots of time, energy, or knowledge, but never make it into job descriptions. Such lists also tend to suggest that system management has some kind of coherence across the vastly different environments in which people find themselves responsible for computers. There are similarities, of course, but what is important on one system won't necessarily be important on another system at another site or on the same system at a different time. Similarly, systems that are very different may have similar system management needs, while nearly identical systems in different environments might have very different needs.

But now to the list. In lieu of an idealized list, I offer the following table showing how I spent most of my time in my first job as full-time system administrator (I managed several central systems driving numerous CAD/CAM workstations at a Fortune 500 company) and how these activities have morphed in the intervening two decades.

Table 1-1. Typical system administration tasks

Then: early 1980s	Now: early 2000s
Adding new users.	I still do it, but it's automated, and I only have to add a user once for the entire network. Converting to LDAP did take a lot of time, though.
Adding toner to electrostatic plotters.	Printers need a lot less attention—just clearing the occasional paper jam—but I still get my hands dirty changing those inkjet tanks.
Doing backups to tape.	Backups are still high priority, but the process is more centralized, and it uses CDs and occasionally spare disks as well as tape.
Restoring files from backups that users accidentally deleted or trashed.	This will never change.
Answering user questions ("How do I send mail?"), usually not for the first or last time.	Users will always have questions. Mine also whine more: "Why can't I have an Internet connection on my desk?" or "Why won't IRC work through the firewall?"

Table 1-1. Typical system administration tasks (continued)

Then: early 1980s	Now: early 2000s
Monitoring system activity and trying to tune system parameters to give these overloaded systems the response time of an idle system.	Installing and upgrading hardware to keep up with monotonically increasing resource appetites.
Moving jobs up in the print queue, after more or less user whining, pleading, or begging, contrary to stated policy (about moving jobs, not about whining).	This is one problem that is no longer an issue for me. Printers are cheap, so they are no longer a scare resource that has to be managed.
Worrying about system security, and plugging the most noxious security holes I inherited.	Security is always a worry, and keeping up with security notices and patches takes a lot of time.
Installing programs and operating system updates.	Same.
Trying to free up disk space (and especially contiguous disk space).	The emphasis is more on high performance disk I/O (disk space is cheap): RAID and so on.
Rebooting the system after a crash (always at late and inconvenient times).	Systems crash a lot less than they used to (thankfully).
Straightening out network glitches ("Why isn't *hamlet* talking to *ophelia*?"). Occasionally, this involved physically tracing the Ethernet cable around the building, checking it at each node.	Last year, I replaced my last Thinnet network with twisted-pair cabling. I hope never to see the former again. However, I now occasionally have to replace cable segments that have malfunctioned.
Rearranging furniture to accommodate new equipment; installing said equipment.	Machines still come and go on a regular basis and have to be accommodated.
Figuring out why a program/command/account suddenly and mysteriously stopped working yesterday, even though the user swore he changed nothing.	Users will still be users.
Fixing—or rather, trying to fix—corrupted CAD/CAM binary data files.	The current analog of this is dealing with email attachments that users don't know how to access. Protecting users from potentially harmful attachments is another concern.
Going to meetings.	No meetings, but lots of casual conversations.
Adding new systems to the network.	This goes without saying: systems are virtually always added to the network.
Writing scripts to automate as many of the above activities as possible.	Automation is still the administrator's salvation.

As this list indicates, system management is truly a hodgepodge of activities and involves at least as many people skills as computer skills. While I'll offer some advice about the latter in a moment, interacting with people is best learned by watching others, emulating their successes, and avoiding their mistakes.

Currently, I look after a potpourri of workstations from many different vendors, as well as a couple of larger systems (in terms of physical size but not necessarily CPU power), with some PCs and Macs thrown in to keep things interesting. Despite these significant hardware changes, it's surprising how many of the activities from the early 1980s I still have to do. Adding toner now means changing a toner cartridge in a laser printer or the ink tanks in an inkjet printer; backups go to 4 mm tape and CDs rather than 9-track tape; user problems and questions are in different areas but

are still very much on the list. And while there are (thankfully) no more meetings, there's probably even more furniture-moving and cable-pulling.

Some of these topics—moving furniture and going to or avoiding meetings, most obviously—are beyond the scope of this book. Space won't allow other topics to be treated exhaustively; in these cases, I'll point you in the direction of another book that takes up where I leave off. This book will cover most of the ordinary tasks that fall under the category of "system administration." The discussion will be relevant whether you've got a single PC (running Unix), a room full of mainframes, a building full of networked workstations, or a combination of several types of computers. Not all topics will apply to everyone, but I've learned not to rule out any of them *a priori* for a given class of user. For example, it's often thought that only big systems need process-accounting facilities, but it's now very common for small businesses to address their computing needs with a moderately-sized Unix system. Because they need to be able to bill their customers individually, they have to keep track of the CPU and other resources expended on behalf of each customer. The moral is this: take what you need and leave the rest; you're the best judge of what's relevant and what isn't.

Thinking About System Administration

I've touched briefly on some of the nontechnical aspects of system administration. These dynamics will probably not be an issue if it really is just you and your PC, but if you interact with other people at all, you'll encounter these issues. It's a cliché that system administration is a thankless job—one widely-reprinted cartoon has a user saying "I'd thank you but system administration is a thankless job"—but things are actually more complicated than that. As another cliché puts it, system administration is like keeping the trains on time; no one notices except when they're late.

System management often seems to involve a tension between authority and responsibility on the one hand and service and cooperation on the other. The extremes seem easier to maintain than any middle ground; fascistic dictators who rule "their system" with an iron hand, unhindered by the needs of users, find their opposite in the harried system managers who jump from one user request to the next, in continual interrupt mode. The trick is to find a balance between being accessible to users and their needs—and sometimes even to their mere wants—while still maintaining your authority and sticking to the policies you've put in place for the overall system welfare. For me, the goal of effective system administration is to provide an environment where users can get done what they need to, in as easy and efficient a manner as possible, given the demands of security, other users' needs, the inherent capabilities of the system, and the realities and constraints of the human community in which they all are located.

To put it more concretely, the key to successful, productive system administration is knowing when to solve a CPU-overuse problem with a command like:*

```
# kill -9 `ps aux | awk '$1=="chavez" {print $2}'
```

(This command blows away all of user *chavez*'s processes.) It's also knowing when to use:

```
$ write chavez
You've got a lot of identical processes running on dalton.
Any problem I can help with?
^D
```

and when to walk over to her desk and talk with her face-to-face. The first approach displays Unix finesse as well as administrative brute force, and both tactics are certainly appropriate—even vital—at times. At other times, a simpler, less aggressive approach will work better to resolve your system's performance problems in addition to the user's confusion. It's also important to remember that there are some problems no Unix command can address.

To a great extent, successful system administration is a combination of careful planning and habit, however much it may seem like crisis intervention at times. The key to handling a crisis well lies in having had the foresight and taken the time to anticipate and plan for the type of emergency that has just come up. As long as it only happens once in a great while, snatching victory from the jaws of defeat can be very satisfying and even exhilarating.

On the other hand, many crises can be prevented altogether by a determined devotion to carrying out all the careful procedures you've designed: changing the root password regularly, faithfully making backups (no matter how tedious), closely monitoring system logs, logging out and clearing the terminal screen as a ritual, testing every change several times before letting it loose, sticking to policies you've set for users' benefit—whatever you need to do for your system. (Emerson said, "A foolish consistency is the hobgoblin of little minds," but not a wise one.)

My philosophy of system administration boils down to a few basic strategies that can be applied to virtually any of its component tasks:

- Know how things work. In these days, when operating systems are marketed as requiring little or no system administration, and the omnipresent simple-to-use tools attempt to make system administration simple for an uninformed novice, someone has to understand the nuances and details of how things really work. It should be you.
- Plan it before you do it.
- Make it reversible (backups help a lot with this one).

* On HP-UX systems, the command is `ps -ef`. Solaris systems can run either form depending on which version of `ps` comes first in the search path. AIX and Linux can emulate both versions, depending on whether a hyphen is used with options (System V style) or not (BSD style).

- Make changes incrementally.
- Test, test, test, before you unleash it on the world.

I learned about the importance of reversibility from a friend who worked in a museum putting together ancient pottery fragments. The museum followed this practice so that if better reconstructive techniques were developed in the future, they could undo the current work and use the better method. As far as possible, I've tried to do the same with computers, adding changes gradually and preserving a path by which to back out of them.

A simple example of this sort of attitude in action concerns editing system configuration files. Unix systems rely on many configuration files, and every major subsystem has its own files (all of which we'll get to). Many of these will need to be modified from time to time.

I never modify the original copy of the configuration file, either as delivered with the system or as I found it when I took over the system. Rather, I always make a copy of these files the first time I change them, appending the suffix *.dist* to the filename; for example:

```
# cd /etc
# cp inittab inittab.dist
# chmod a-w inittab.dist
```

I write-protect the *.dist* file so I'll always have it to refer to. On systems that support it, use the `cp` command's `-p` option to replicate the file's current modification time in the copy.

I also make a copy of the current configuration file before changing it in any way so undesirable changes can be easily undone. I add a suffix like *.old* or *.sav* to the filename for these copies. At the same time, I formulate a plan (at least in my head) about how I would recover from the worst consequence I can envision of an unsuccessful change (e.g., I'll boot to single-user mode and copy the old version back).

Once I've made the necessary changes (or the first major change, when several are needed), I test the new version of the file, in a safe (nonproduction) environment if possible. Of course, testing doesn't always find every bug or prevent every problem, but it eliminates the most obvious ones. Making only one major change at a time also makes testing easier.

Some administrators use the a revision control system to track the changes to important system configuration files (e.g., CVS or RCS). Such packages are designed to track and manage changes to application source code by multiple programmers, but they can also be used to record changes to configuration files. Using a revision control system allows you to record the author and reason for any particular change, as well as reconstruct any previous version of a file at any time.

The remaining sections of this chapter discuss some important administrative tools. The first describes how to become the superuser (the Unix privileged account). Because I believe a good system manager needs to have both technical expertise and an awareness of and sensitivity to the user community of which he's a part, this first chapter includes a section on Unix communication commands. The goal of these discussions—as well as of this book as a whole—is to highlight how a system manager thinks about system tasks and problems, rather than merely to provide literal, cookbook solutions for common scenarios.

Important administrative tools of other kinds are covered in later chapters of this book.

Becoming Superuser

On a Unix system, the superuser refers to a privileged account with unrestricted access to all files and commands. The username of this account is *root*. Many administrative tasks and their associated commands require superuser status.

There are two ways to become the superuser. The first is to log in as *root* directly. The second way is to execute the command `su` while logged in to another user account. The `su` command may be used to change one's current account to that of a different user after entering the proper password. It takes the username corresponding to the desired account as its argument; *root* is the default when no argument is provided.

After you enter the `su` command (without arguments), the system prompts you for the *root* password. If you type the password correctly, you'll get the normal root account prompt (by default, a number sign: #), indicating that you have successfully become superuser and that the rules normally restricting file access and command execution do not apply. For example:

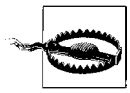
```
$ su
Password:    Not echoed
#
```

If you type the password incorrectly, you get an error message and return to the normal command prompt.

You may exit from the superuser account with `exit` or Ctrl-D. You may suspend the shell and place it in the background with the `suspend` command; you can return to it later using `fg`.

When you run `su`, the new shell inherits the environment from your current shell environment rather than creating the environment that *root* would get after logging in. However, you can simulate an actual *root* login session with the following command form:

```
$ su -
```



Unlike some other operating systems, the Unix superuser has all privileges all the time: access to all files, commands, etc. Therefore, it is entirely too easy for a superuser to crash the system, destroy important files, and create havoc inadvertently. For this reason, people who know the superuser password (including the system administrator) should *not* do their routine work as superuser. *Only use superuser status when it is needed.*

The *root* account should always have a password, and this password should be changed periodically. Only experienced Unix users with special requirements should know the superuser password, and the number of people who know it should be kept to an absolute minimum.

To set or change the superuser password, become superuser and execute one of the following commands:

```
# passwd          Works most of the time.
# passwd root     Solaris and FreeBSD systems when su'd to root.
```

Generally, you'll be asked to type the old superuser password and then the new password twice. The *root* password should also be changed whenever someone who knows it stops using the system for any reason (e.g., transfer, new job, etc.), or if there is any suspicion that an unauthorized user has learned it. Passwords are discussed in detail in Chapter 6.

I try to avoid logging in directly as *root*. Instead, I `su` to *root* only as necessary, exiting from or suspending the superuser shell when possible. Alternatively, in a windowing environment, you can create a separate window in which you `su` to *root*, again executing commands there only as necessary.

For security reasons, it's a bad idea to leave any logged-in session unattended; naturally, that goes double for a *root* session. Whenever I leave a workstation where I am logged in as *root*, I log out or lock the screen to prevent anyone from sneaking onto the system. The `xlock` command will lock an X session; the password of the user who ran `xlock` must be entered to unlock the session (on some systems, the *root* password can also unlock sessions locked by other users).* While screen locking programs may have security pitfalls of their own, they do prevent opportunistic breaches of system security that would otherwise be caused by a momentary lapse into laziness.

If you are logged in as *root* on a serial console, you should also use a locking utility provided by the operating system. In some cases, if you are using multiple virtual consoles, you will need to lock each one individually.

* For some unknown reason, FreeBSD does not provide `xlock`. However, the `xlockmore` (see *http://www.tux.org/~bagleyd/xlockmore.html*) utility provides the same functionality (it's actually a follow-on to `xlock`).

Controlling Access to the Superuser Account

On many systems, any user who knows the root password may become superuser at any time by running su. This is true for HP-UX, Linux, and Solaris systems in general.* Solaris allows you to configure some aspects of how the command works via settings in the */etc/default/su* configuration file.

Traditionally, BSD systems limited access to su to members of group 0 (usually named *wheel*); under FreeBSD, if the *wheel* group has a null user list in the group file (*/etc/group*), any user may su to root; otherwise, only members of the *wheel* group can use it. The default configuration is a *wheel* group consisting of just *root*.

AIX allows the system administrator to specify who can use su on an account-by-account basis (no restrictions are imposed by default). The following commands display the current groups that are allowed to su to *root* and then limit that same access to the *system* and *admins* groups:

```
# lsuser -a sugroups root
root sugroups=ALL
# chuser sugroups="system,admins" root
```

Most Unix versions also allow you to restrict direct *root* logins to certain terminals. This topic is discussed in Chapter 12.

An Armadillo?

The armadillo typifies one attribute that a successful system administrator needs: a thick skin. Armadillos thrive under difficult environmental conditions through strength and perseverance, which is also what system administrators have to do a lot of the time (see the colophon at the back of the book for more information about the armadillo). System managers will find other qualities valuable as well, including the quickness and cleverness of the mongoose (Unix is the snake), the sense of adventure and playfulness of puppies and kittens, and at times, the chameleon's ability to blend in with the surroundings, becoming invisible even though you're right in front of everyone's eyes.

Finally, however, as more than one reader has noted, the armadillo also provides a cautionary warning to system administrators not to become so single-mindedly or narrowly focused on what they are doing that they miss the big picture. Armadillos who fail to heed this advice end up as roadkill.

* When the PAM authentication facility is in use, it controls access to su (see "User Authentication with PAM" in Chapter 6).

Running a Single Command as root

su also has a mode whereby a single command can be run as *root*. This mode is not a very convenient way to interactively execute superuser commands, and I tend to see it as a pretty unimportant feature of su. Using su -c can be very useful in scripts, however, keeping in mind that the target user need not be *root*.

Nevertheless, I have found that it does have one important use for a system administrator: it allows you to fix something quickly when you are at a user's workstation (or otherwise not at your own system) without having to worry about remembering to exit from an su session.* There are users who will absolutely take advantage of such lapses, so I've learned to be cautious.

You can run a single command as *root* by using a command of this form:

```
$ su root -c "command"
```

where *command* is replaced by the command you want to run. The command should be enclosed in quotation marks if it contains any spaces or special shell characters. When you execute a command of this form, su prompts for the *root* password. If you enter the correct password, the specified command runs as *root*, and subsequent commands are run normally from the original shell. If the command produces an error or is terminated (e.g. with CTRL-C), control again returns to the unprivileged user shell.

The following example illustrates this use of su to unmount and eject the CD-ROM mounted in the */cdrom* directory:

```
$ su root -c "eject /cdrom"
Password:    root password entered
```

Commands and output would be slightly different on other systems.

You can start a background command as *root* by including a final ampersand within the specified command (*inside* the quotation marks), but you'll want to consider the security implications of a user bringing it to the foreground before you do this at a user's workstation.

sudo: Selective Access to Superuser Commands

Standard Unix takes an all-or-nothing approach to granting *root* access, but often what you actually want is something in between. The freely available sudo facility allows specified users to run specific commands as *root* without having to know the *root* password (available at *http://www.courtesan.com/sudo/*).†

* Another approach is always to open a new window when you need to do something at a user's workstation. It's easy to get into the habit of always closing it down as you leave.

† Administrative roles are another, more sophisticated way of partitioning *root* access. They are discussed in detail in "Role-Based Access Control" in Chapter 7.

For example, a non-*root* user could use this sudo command to shut down the system:

```
$ sudo  /sbin/shutdown ...
Password:
```

sudo requires only the user's own password to run the command, not the *root* password. Once a user has successfully given a password to sudo, she may use it to run additional commands for a limited period of time without having to enter a password again; this period defaults to five minutes. A user can extend the time period by an equal amount by running sudo -v before it expires. She can also terminate the grace period by running sudo -K.

sudo uses a configuration file, usually */etc/sudoers*, to determine which users may use the sudo command and the other commands available to each of them after they've started a sudo session. The configuration file must be set up by the system administrator. Here is the beginning of a sample version:

```
# Host alias specifications: names for host lists
Host_Alias    PHYSICS = hamlet, ophelia, laertes
Host_Alias    CHEM = duncan, puck, brutus

# User alias specifications: named groups of users
User_Alias    BACKUPOPS = chavez, vargas, smith

# Command alias specifications: names for command groups
Cmnd_Alias    MOUNT = /sbin/mount, /sbin/umount
Cmnd_Alias    SHUTDOWN = /sbin/shutdown
Cmnd_Alias    BACKUP = /usr/bin/tar, /usr/bin/mt
Cmnd_Alias    CDROM = /sbin/mount /cdrom, /bin/eject
```

These three configuration file sections define sudo aliases—uppercase symbolic names—for groups of computers, users and commands, respectively. This example file defines two sets of hosts (PHYSICS and CHEM), one set of users (BACKUPOPS), and four command aliases. For example, the MOUNT command alias is defined as the mount and umount commands. Following good security practice, all commands include the full pathname for the executable.

The final command alias illustrates the use of arguments within a command list. This alias consists of a command to mount a CD at */cdrom* and to eject the media from the drive. Note, however, that it does not grant general use of the mount command.

The final section of the file (see below) specifies which users may use the sudo command, as well as what commands they can run with it and which computers they may run them on. Each line in this section consists of a username or alias, followed by one or more items of the form:

```
host = command(s) [: host = command(s) ...]
```

where *host* is a hostname or a host alias, and *command(s)* are one or more commands or command aliases, with multiple commands or hosts separated by commas. Multiple access specifications may be included for a single user, separated by colons. The alias ALL stands for all hosts or commands, depending on its context.

Here is the remainder of our example configuration file:

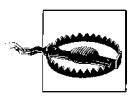
```
# User specifications: who can do what where
root          ALL = ALL
%chem         CHEM = SHUTDOWN, MOUNT
chavez        PHYSICS = MOUNT : achilles = /sbin/swapon
harvey        ALL = NOPASSWD: SHUTDOWN
BACKUPOPS     ALL, !CHEM = BACKUP, /usr/local/bin
```

The first entry after the comment grants *root* access to all commands on all hosts. The second entry applies to members of the *chem* group (indicated by the initial percent sign), who may run system shutdown and mounting commands on any computer in the CHEM list.

The third entry specifies that user *chavez* may run the mounting commands on the hosts in the PHYSICS list and may also run the `swapon` command on host *achilles*. The next entry allows user *harvey* to run the `shutdown` command on any system, and `sudo` will not require him to enter his password (via the NOPASSWD: preceding the command list).

The final entry applies to the users specified for the BACKOPS alias. On any system except those in the CHEM list (the preceding exclamation point indicates exclusion), they may run the command listed in the BACKUP alias as well as any command in the */usr/local/bin directory*.

Users can use the `sudo -l` command form to list the commands available to them via this facility.



Commands should be selected for use with `sudo` with some care. In particular, shell scripts should not be used, nor should any utility which provides *shell escapes*—the ability to execute a shell command from within a running interactive program (editors, games, and even output display utilities like `more` and `less` are common examples). Here is the reason: when a user runs a command with `sudo`, that command runs as *root*, so if the command lets the user execute other commands via a shell escape, any command he runs from within the utility will also be run as *root*, and the whole purpose of `sudo`—to grant *selective* access to superuser command—will be subverted. Following similar reasoning, because most text editors provide shell escapes, any command that allows the user to invoke an editor should also be avoided. Some administrative utilities (e.g., AIX's SMIT) also provide shell escapes.

The `sudo` package provides the `visudo` command for editing */etc/sudoers*. It locks the file, preventing two users from modifying the file simultaneously, and it performs syntax checking when editing is complete (if there are errors, the editor is restarted, but no explicit error messages are given).

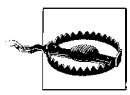
There are other ways you might want to customize `sudo`. For example, I want to use a somewhat longer interval for password-free use. Changes of this sort must be made by rebuilding `sudo` from source code. This requires rerunning the `configure` script

with options. Here is the command I used, which specifies a log file for all sudo operations, sets the password-free period to ten minutes, and tells visudo to use the text editor specified in the *EDITOR* environment variable:

```
# cd sudo-source-directory
# ./configure --with-logpath=/var/log/sudo.log \
            --with-timeout=10 --with-env-editor
```

Once the command completes, use the make command to rebuild sudo.*

sudo's logging facility is important and useful in that it enables you to keep track of privileged commands that are run. For this reason, using sudo can sometimes be preferable to using su even when limiting *root*-level command access is not an issue.



The one disadvantage of sudo is that it provides no integrated remote-access password protection. Thus, when you run sudo from an insecure remote session, passwords are transmitted over the network for any eavesdropper to see. Of course, using SSH can overcome this limitation.

Communicating with Users

The commands discussed in this section are simple and familiar to most Unix users. For this reason, they're often overlooked in system administration discussions. However, I believe you'll find them to be an indispensable part of your repertoire. One other important communications mechanism is electronic mail (see Chapter 9).

Sending a Message

A system administrator frequently needs to send a message to a user's screen (or window). write is one way to do so:

```
$ write username [tty]
```

where *username* indicates the user to whom you wish to send the message. If you want to write to a user who is logged in more than once, the *tty* argument may be used to select the appropriate terminal or window. You can find out where a user is logged in using the who command.

Once the write command is executed, communication is established between your terminal and the user's terminal: lines that you type on your terminal will be transmitted to him. End your message with a CTRL-D. Thus, to send a message to user *harvey* for which no reply is needed, execute a command like this:

* A couple more configuration notes: sudo can also be integrated into the PAM authentication system (see "User Authentication with PAM" in Chapter 6). Use the --use-pam option to configure. On the other hand, if your system does not use a shadow password file, you must use the --disable-shadow option.

```
$ write harvey
The file you needed has been restored.
Additional lines of message text
^D
```

In some implementations (e.g., AIX, HP-UX and Tru64), `write` may also be used over a network by appending a hostname to the username. For example, the command below initiates a message to user *chavez* on the host named *hamlet*:

```
$ write chavez@hamlet
```

When available, the `rwho` command may be used to list all users on the local subnet (it requires a remote who daemon be running on the remote system).

The `talk` command is a more sophisticated version of `write`. It formats the messages between two users in two separate spaces on the screen. The recipient is notified that someone is calling her, and she must issue her own `talk` command to begin communication. Figure 1-1 illustrates the use of `talk`.

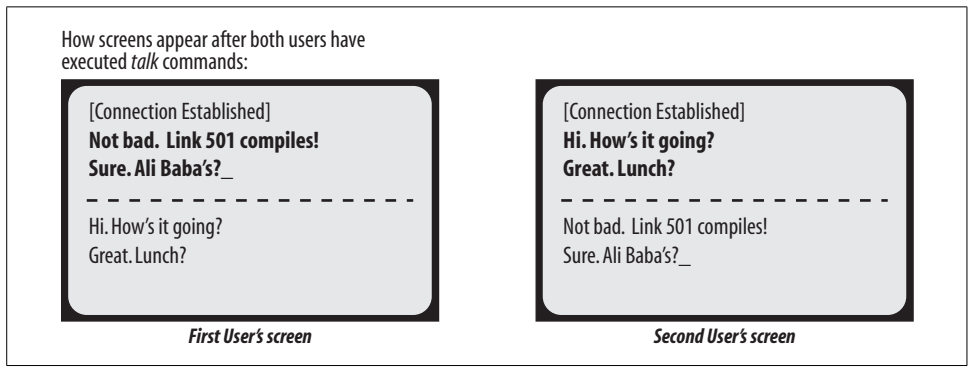


Figure 1-1. Two-way communication with talk

Users may disable messages from both `write` and `talk` by using the command `mesg n` (they can include it in their *.login* or *.profile* initialization file). Sending messages as the superuser overrides this command. Be aware, however, that sometimes users have good reasons for turning off messages.

In general, the effectiveness of system messages is inversely proportional to their frequency.

Sending a Message to All Users

If you need to send a message to every user on the system, you can use the `wall` command. `wall` stands for "write all" and allows the administrator to send a message to all users simultaneously.

To send a message to all users, execute the command:

```
$ wall
Followed by the message you want to send, terminated with CTRL-D on a separate line
^D
```

Unix then displays a phrase like:

```
Broadcast Message from root on console ...
```

to every user, followed by the text of your message. Similarly, the `rwall` command sends a message to every user on the local subnet.

Anyone can use this facility; it does not require superuser status. However, as with `write` and `talk`, only messages from the superuser override users' `mesg n` commands. A good example of such a message would be to give advance warning of an imminent but unscheduled system shutdown.

The Message of the Day

Login time is a good time to communicate certain types of information to users. It's one of the few times that you can be reasonably sure of having a user's attention (sending a message to the screen won't do much good if the user isn't at the workstation). The file */etc/motd* is the system's message of the day. Whenever anyone logs in, the system displays the contents of this file. You can use it to display system-wide information such as maintenance schedules, news about new software, an announcement about someone's birthday, or anything else considered important and appropriate on your system. This file should be short enough so that it will fit entirely on a typical screen or window. If it isn't, users won't be able to read the entire message as they log in.

On many systems, a user can disable the message of the day by creating a file named *.hushlogin* in her home directory.

Specifying the Pre-Login Message

On Solaris, HP-UX, Linux and Tru64 systems, the contents of the file */etc/issue* is displayed immediately before the login prompt on unused terminals. You can customize this message by editing this file.

On other systems, login prompts are specified as part of the terminal-related configuration files; these are discussed in Chapter 12.

About Menus and GUIs

For several years now, vendors and independent programmers have been developing elaborate system administration applications. The first of these were menu-driven, containing many levels of nested menus organized by subsystem or administrative

task. Now, the trend is toward independent GUI-based tools, each designed to manage some particular system area and perform the associated tasks.

Whatever their design, all of them are designed to allow even relative novices to perform routine administrative tasks. The scope and aesthetic complexity of these tools vary considerably, ranging from shell scripts employing simple selections lists and prompts to form-based utilities running under X. A few even offer a mouse-based interface with which you perform operations by dragging icons around (e.g., dropping a user icon on top of a group icon adds that user to that group, dragging a disk icon into the trash unmounts a filesystem, and the like).

In this section, we'll take a look at such tools, beginning with general concepts and then going on to a few practical notes about the tools available on the systems we are considering (usually things I wish I had known about earlier). The tools are very easy to use, so I won't be including detailed instructions for using them (consult the appropriate documentation for that).

Ups and Downs

Graphical and menu-based system administration tools have some definite good points:

- They can provide a quick start to system administration, allowing you to get things done while you learn about the operating system and how things work. The best tools include aids to help you learn the underlying standard administrative commands.

  Similarly, these tools can be helpful in figuring out how to perform some task for the first time; when you don't know how to begin, it can be hard to find a solution with just the manual pages.
- They can help you get the syntax right for complex commands with lots of options.
- They make certain kinds of operations more convenient by combining several steps into a single menu screen (e.g., adding a user or installing an operating system upgrade).

On the other hand, they have their down side as well:

- Typing the equivalent command is usually significantly faster than running it from an administrative tool.
- Not all commands are always available through the menu system, and sometimes only part of the functionality is implemented for commands that are included. Often only the most frequently used commands and/or options are available. Thus, you'll still need to execute some versions of commands by hand.
- Using an administrative tool can slow down the learning process and sometimes stop it altogether. I've met inexperienced administrators who had become

convinced that certain operations just weren't possible simply because the menu system didn't happen to include them.

- The GUI provides unique functionality accessible only through its interface, so creating scripts to automate frequent tasks becomes much more difficult or impossible, especially when you want to do things in a way that the original author did not think of.

In my view, an ideal administrative tool has all of these characteristics:

- The tool must run normal operating system commands, not opaque, undocumented programs stored in some obscure, out-of-the-way directory. The tool thus makes system administration easier, leaving the thinking to the human using it.
- You should be able to display the commands being run, ideally before they are executed.
- The tool should log of all its activities (at least optionally).
- As much as possible, the tool should validate the values the user enters. In fact, novice administrators frequently assume that the tools do make sure their selections are reasonable, falsely thinking that they are protected from anything harmful.
- All of the options for commands included in the tool should be available for use, except when doing so would violate the next item.
- The tool should not include every administrative command. More specifically, it should deliberately omit commands that could cause catastrophic consequences if they are used incorrectly. Which items to omit depends on the sort of administrators the tool is designed for; the scope of the tool should be directly proportional to the amount of knowledge its user is assumed to have. In the extreme case, dragging a disk icon into a trash can icon should never do anything other than dismount it, and there should not be any way to, say, reformat an existing filesystem. Given that such a tool is consciously designed for minimally-competent administrators, including such capabilities is just asking for trouble.

In addition, these features make using an administrative tool much more efficient, but they are not absolutely essential:

- A way of specifying the desired starting location within a deep menu tree when you invoke the tool.
- A one-keystroke exit command that works at every point within menu system.
- Context-sensitive help.
- The ability to limit access to subsections of the tool by user.
- Customization features.

If one uses these criteria, AIX's SMIT comes closest to an ideal administrative tool, a finding that many have found ironic.

As usual, using menu interfaces in moderation is probably the best approach. These applications are great when they save you time and effort, but relying on them to lead you through every situation will inevitably lead to frustration and disappointment somewhere down the line.

The Unix versions we are considering offer various system administration facilities. They are summarized and compared in Table 1-2. The table columns hold the Unix version, tool command or name, tool type, whether or not the command to be run can be previewed before execution, whether or not the facility can log its actions and whether or not the tool can be used to administer remote systems.

Table 1-2. Some system administration facilities

Unix Version	Command/tool	Type	Command preview?	Creates logs?[a]	Remote admin?
AIX	`smit`	menu	yes	yes	no
	WSM	GUI	no	no	yes
FreeBSD	`sysinstall`	menu	no	no	no
HP-UX	`sam`	both	no	yes	yes
Linux	`linuxconf`	both	no	no	no
Red Hat Linux	`redhat-config-*`	GUI	no	no	no
SuSE Linux	`yast`	menu	no	no	no
	`yast2`	GUI	no	no	no
Solaris	`admintool`	menu	no	no	no
	CDE admin tools	GUI	no	no	no
	AdminSuite/SMC	menu	no	yes	yes
Tru64	`sysman`	menu	no	no	no
	`sysman -station`	menu	no	no	yes

[a] Some tools do some rather half-hearted logging to the syslog facility, but it's not very useful.

There are also some other tools on some of these systems that will be mentioned in this book when appropriate, but they are ignored here.

AIX: SMIT and WSM

AIX offers two main system administration facilities: the System Management Interface Tool (SMIT) and the Workspace System Manager (WSM) facility. Both of them run in both graphical and text mode.

SMIT consists of a many-leveled series of nested menus. Its main menu is illustrated in Figure 1-2.

One of SMIT's most helpful features is command preview: if you click on the Command button or press F6, SMIT displays the command to be executed by the current dialog. This feature is illustrated in the window on the right in Figure 1-2.

You can also go directly to any screen by including the corresponding *fast path* keyword on the `smit` command line. Many SMIT fast paths are the same as the command

Why Menus and Icons Aren't Enough

Every site needs at least one experienced system administrator who can perform those tasks that are beyond the abilities of the administrative tool. Not only does every current tool leave significant amounts of uncovered territory, but they also all suffer from limitations inherent in programs designed for routine operations under normal system conditions. When the system is in trouble, and these assumptions no longer hold, the tools don't work.

For example, I've been in a situation where the administrative tool couldn't configure a replacement because the old disk hadn't been unconfigured properly before being removed. One part of the tool thought the old disk was still on the system and wouldn't replace it, while another part wouldn't delete the old configuration data because it couldn't access the corresponding physical disk.

I was able to solve this problem because I understood enough about the device database on that system to fix things manually. Not only will such things happen to every system from time to time, they will happen to everyone, sooner or later. It's a lot easier to coax a system back to life from single user mode after a power failure when you understand, for example, what the Check Filesystem Integrity menu item actually does. In the end, *you* need to know how things really work.

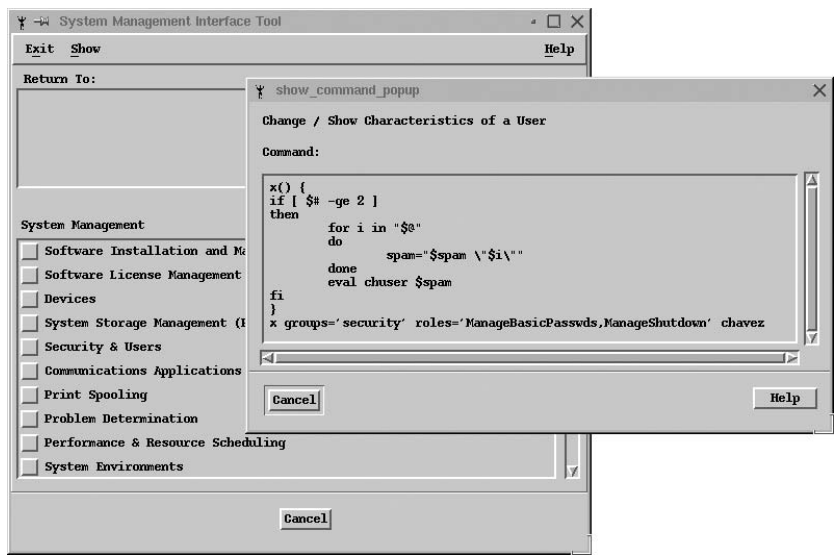


Figure 1-2. The AIX SMIT facility

executed from a particular screen. Many other fast paths fall into a predictable pattern, beginning with one of the prefixes `mk` (make or start), `ch` (change or reconfigure), `ls` (list), or `rm` (remove or stop), to which an object code is appended: `mkuser`, `chuser`,

`lsuser`, `rmuser` for working with user accounts; `mkprt`, `chprt`, `lsprt`, `rmprt` for working with printers, and so on. Thus, it's often easy to guess the fast path you want.

You can display the fast path for any SMIT screen by pressing F8 in the ASCII version of the tool:

```
Current fast path:
     "mkuser"
```

If the screen doesn't have a fast path, the second line will be blank. Other useful fast paths that are harder to guess include the following:

chgsys
: View/change AIX parameters.

configtcp
: Reconfigure TCP/IP.

crfs
: Create a new filesystem.

lvm
: Main Logical Volume Manager menu.

_nfs
: Main NFS menu.

spooler
: Manipulate print jobs.

Here are a few additional SMIT notes:

- The `smitty` command may be used to start the ASCII version of SMIT from within an X session (where the graphical version is invoked by default).
- Although I like them, many people are annoyed by the SMIT log files. You can use a command like this one to eliminate the SMIT log files:

  ```
  $ smit -s /dev/null -l /dev/null ...
  ```

  You can define an alias in your shell initialization file to get rid of these files permanently (C shell users would omit the equals sign):

  ```
  alias smit="/usr/sbin/smitty -s /dev/null -l /dev/null"
  ```

- `smit -x` provides a command preview mode. The commands that would be run are written to the log file but not executed.
- Newer versions of `smit` have the following annoying feature: when a command has successfully completed, and you click Done to close the output window, you are taken back to the command setup window. At this point, to exit, you must click Cancel, not OK. Doing the latter will cause the command to run again, which is not what you want and is occasionally quite troublesome!

The WSM facility contains a variety of GUI-based tools for managing various aspects of the system. Its functionality is a superset of SMIT's, and it has the advantage of being able to administer remote systems (it requires that remote systems be running

a web server). You can access WSM via the Common Desktop Environment's Applications area: click on the file cabinet icon (the one with the calculator peeking out of it); the system administration tools are then accessible under the System_Admin icon. You can also run a command-line version of WSM via the `wsm` command.

The WSM tools are run on a remote system via a Java-enabled web browser. You can connect to the tools by pointing the browser at *http://*hostname*/wsm.html*, where *hostname* corresponds to the desired remote system. Of course, you can also run the text version by entering the `wsm` command into a remote terminal session.

HP-UX: SAM

HP-UX provides the System Administration Manager, also known as SAM. SAM is easy to use and can perform a variety of system management tasks. SAM operates in both menu-based and GUI mode, although the latter requires support for Motif.

The items on SAM's menus invoke a combination of regular HP-UX commands and special scripts and programs, so it's not always obvious what they do. One way to find out more is to use SAM's built-in logging feature. SAM allows you to specify the level of detail in log file displays, and you can optionally keep the log open as you are working in order to monitor what is actually happening. The SAM main window and log display are illustrated in Figure 1-3.

If you really want to know what SAM is doing, you'll need to consult its configuration files, stored in the subdirectories of */usr/sam/lib*. Most subdirectories have two-character names, closely related to a top-level icon or menu item. For example, the *ug* subdirectory contains files for the Users and Groups module, and the *pm* subdirectory contains those for Process Management. If you examine the *.tm* file there, you can figure out what some of the menu items do. This example illustrates the kinds of items to look for in these files:

```
#egrep '^task [a-z]|^ *execute' pm.tm
task pm_get_ps {
   execute "/usr/sam/lbin/pm_parse_ps"
task pm_add_cron {
   execute "/usr/sam/lbin/cron_change ADD /var/sam/pm_tmpfile"
task pm_add_cron_check {
   execute "/usr/sam/lbin/cron_change CHECK /var/sam/pm_tmpfile"
task pm_mod_nice {
   execute "unset UNIX95;/usr/sbin/renice -n %$INT_ID% %$STRING_ID%"
task pm_rm_cron {
   execute "/usr/sam/lbin/cron_change REMOVE /var/sam/pm_tmpfile"
```

The items come in pairs, relating a menu item or icon and an actual HP-UX command. For example, the fourth pair in the previous output allows you to figure out what the Modify Nice Priority menu item does (runs the `renice` command). The second pair indicates that the item related to adding `cron` entries executes the listed shell script; you can examine that file directly to get further details.

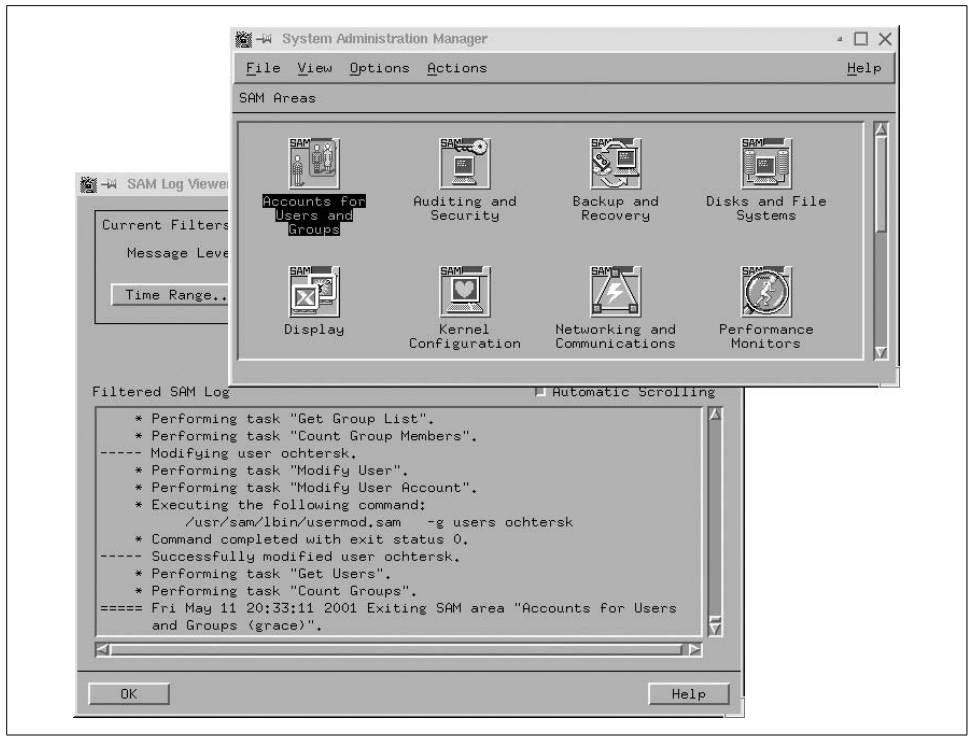


Figure 1-3. The HP-UX SAM facility

There is another configuration file for each main menu item in the */usr/sam/lib/C* subdirectory, named *pm.ui* in this case. Examining the lines containing "action" and "do" provides similar information. Note that "do" entries that end with parentheses (e.g., *do pm_forcekill_xmit()*) indicate a call to a routine in one of SAM's component shared libraries, which will mean the end of the trail for your detective work.

SAM allows you to selectively grant access to its functional areas on a per-user basis. Invoke it via `sam -r` to set up user privileges and restrictions. In this mode, you select the user or group for which you want to define allowed access, and then you navigate through the various icons and menus, enabling or disabling items as appropriate. When you are finished, you can save these settings and also save groups of settings as named permission templates that can subsequently be applied to other users and groups.

In this mode, the SAM display changes, and the icons are colored indicating the allowed access: red for prohibited, green for allowed, and yellow when some features are allowed and others are prohibited.

You can use SAM for remote administration by selecting the Run SAM on Remote System icon from the main window. The first time you connect to a specific remote system, SAM automatically sets up the environment.

Solaris: admintool and Sun Management Console

From a certain point of view, current versions of Solaris actually offer three distinct tool options:

- `admintool`, the menu-based system administration package available under Solaris for many years. You must be a member of the *sysadmin* group to run this program.
- A set of GUI-based tools found under the System_Admin icon of the Applications Manager window under the Common Desktop Environment (CDE), which is illustrated on the left in Figure 1-4. Select the Applications → Application Manager menu path from the CDE's menu to open this window. Most of these tools are very simple, one-task utilities related to media management, although there is also an icon there for `admintool`.
- The Solaris AdminSuite, whose components are controlled by the Sun Management Console (SMC). The facility's main window is illustrated on the right in Figure 1-4.

  In some cases, this package is included with the Solaris operating system. It is also available for (free) download (from *http://www.sun.com/bigadmin/content/adminpack/*). In fact, it is well worth the overnight download required if you have only a slow modem (two nights if you want the documentation as well).

  This tool can be used to perform administrative tasks on remote systems. You specify the system on which you want to operate when you log in to the facility.

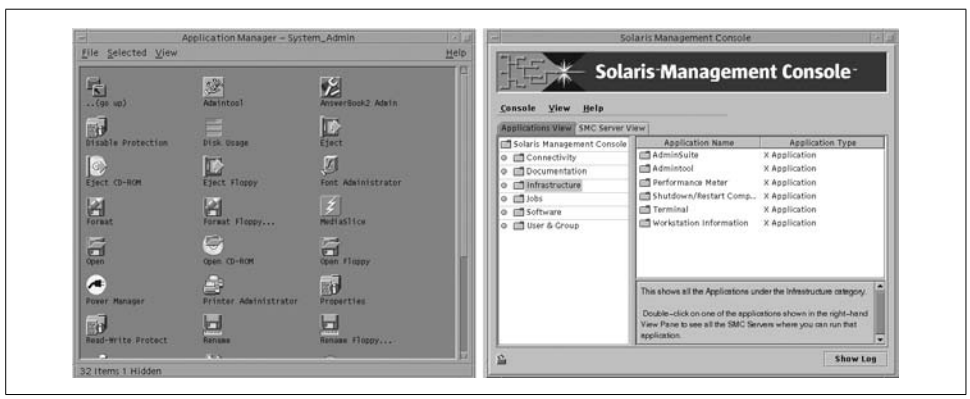


Figure 1-4. Solaris system administration tools

Linux: Linuxconf

Many Linux systems, including some Red Hat versions, offer the Linuxconf graphical administrative tool written by Jacques Gélinas. This tool can also be used with other Linux distributions (see *http://www.solucorp.qc.ca/linuxconf/*). It is illustrated in Figure 1-5.

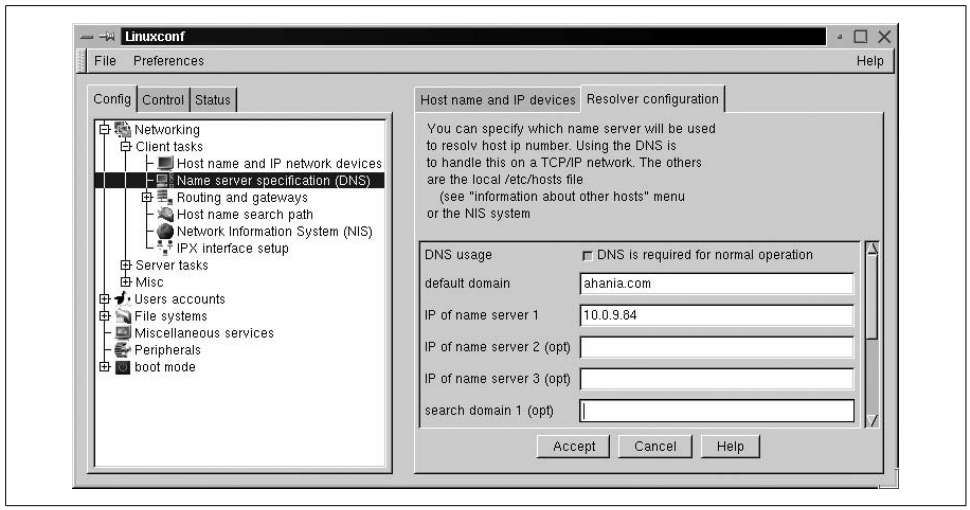


Figure 1-5. The Linuxconf facility

The tool's menu system is located in the area on the left, and forms related to the current selection are displayed on the right. Several of the program's subsections can be accessed directly via separate commands (which are in fact just links to the main `linuxconf` executable): `fsconf`, `mailconf`, `modemconf`, `netconf`, `userconf`, and `uucpconf`, which administer filesystems, electronic mail, modems, networking parameters, users and groups and UUCP, respectively.

Early versions of Linuxconf were dreadful: bug-rich and unbelievably slow. However, more recent versions have improved quite a bit, and the current version is pretty good. Linuxconf leans toward supporting all available options at the expense of novice's ease-of-use at times (a choice with which I won't quarrel). As a result, it is a tool that can make many kinds of configuration tasks easier for an experienced administrator; less expert users may find the number of settings in some dialogs to be somewhat daunting. You can also specify access to Linuxconf and its various subsections on a per-user basis (this is configured via the user account settings).

Red Hat Linux: redhat-config-*

Red Hat Linux provides several GUI-based administration tools, including these:

`redhat-config-bindconf`
: Configure the DNS server (`redhat-config-bind` under Version 7.2).

`redhat-config-network`
: Configure the networking on the local host (new with Red Hat Version 7.3).

`redhat-config-printer-gui`
: Configure and manage print queues and the print server.

`redhat-config-services`
: Select servers to be started at boot time.

`redhat-config-date` *and* `redhat-config-time`
: Set the date and/or time.

`redhat-config-users`
: Configure user accounts and groups.

There are often links to some of these utilities with different (shorter) names. They can also be accessed via icons from the System Settings icon under Start Here. Figure 1-6 illustrates the dialogs for creating a new user account (left) and specifying the local system's DNS server (right).

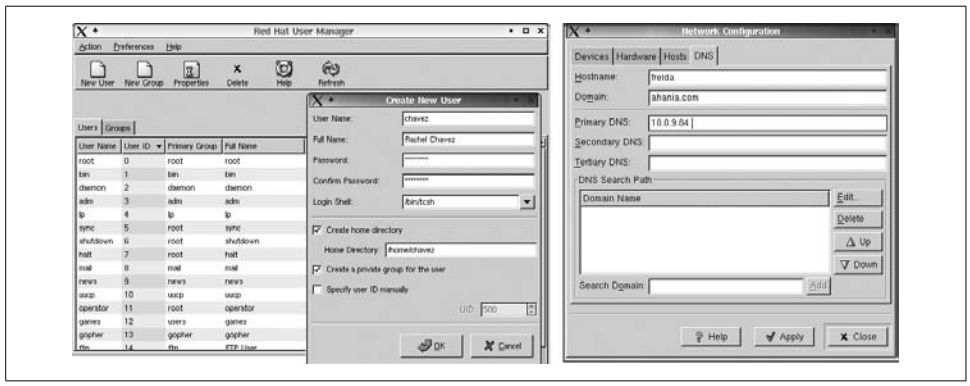


Figure 1-6. Red Hat Linux system configuration tools

SuSE Linux: YaST2

The "YaST" in YaST2 stands for "yet another setup tool." It is a follow-on to the original YaST, and like the previous program (which is also available), it is a somewhat prettied up menu-based administration facility. The program's main window is illustrated in Figure 1-7.

The `yast2` command is used to start the tool. Generally, the tool is easy to use and does its job pretty well. It does have one disadvantage, however. Whenever you add a new package or make other kinds of changes to the system configuration, the *SuSEconfig* script runs (actually, a series of scripts in */sbin/conf.d*). Before SuSE Version 8, this process was fiendishly slow.

SuSEconfig's actions are controlled by the settings in the */etc/rc.config* configuration file, as well as those in */etc/rc.config.d* (SuSE Version 7) or */etc/sysconfig* (SuSE Version 8). Its slowness stems from the fact that every action is performed every time anything changes on the system; in other words, it has no intelligence whatsoever that would allow it to operate only on items and areas that were modified.

Even worse, on SuSE 7 systems, *SuSEconfig*'s actions are occasionally just plain wrong. A particularly egregious example occurs with the Postfix electronic mail package. By default, the primary Postfix configuration file, *main.cf*, is overwritten

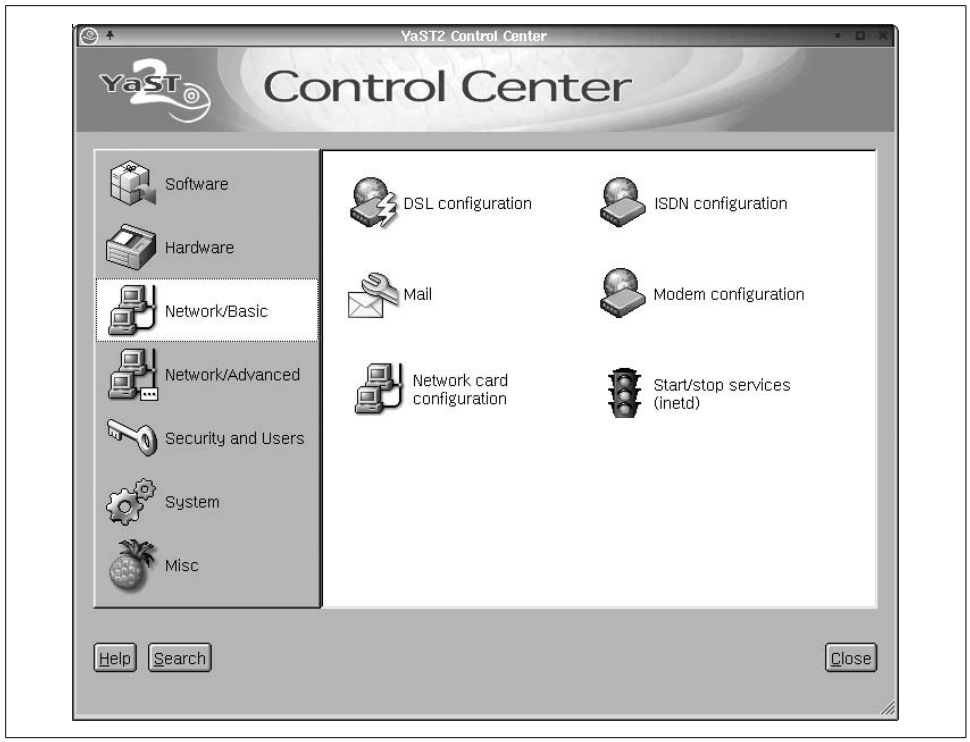


Figure 1-7. The SuSE Linux YaST2 facility

every time the Postfix *SuSEconfig* subscript is executed.* The latter happens every time *SuSEconfig* runs, which is practically every time you change anything on the system with YaST or YaST2 (regardless of its lack of relevance to Postfix). The net result is that any local customizations to *main.cf* get lost. Clearly, adding a new game package, for example, shouldn't clobber a key electronic-mail configuration file.

Fortunately, these problems have been cleared up in SuSE Version 8. I do also use YaST2 on SuSE 7 systems, but I've examined all of the component subscripts thoroughly and made changes to configuration files to disable actions I didn't want. You should do the same.

FreeBSD: sysinstall

FreeBSD offers only the `sysinstall` utility in terms of administrative tools, the same program that manages operating system installations and upgrades (its main menu is illustrated in Figure 1-8). Accordingly, the tasks that it can handle are limited to the ones that come up in the context of operating system installations: managing disks and partitions, basic networking configuration, and so on.

* You can prevent this by setting POSTFIX_CREATECF to no in */etc/rc.config.d/postfix.rc.config*.

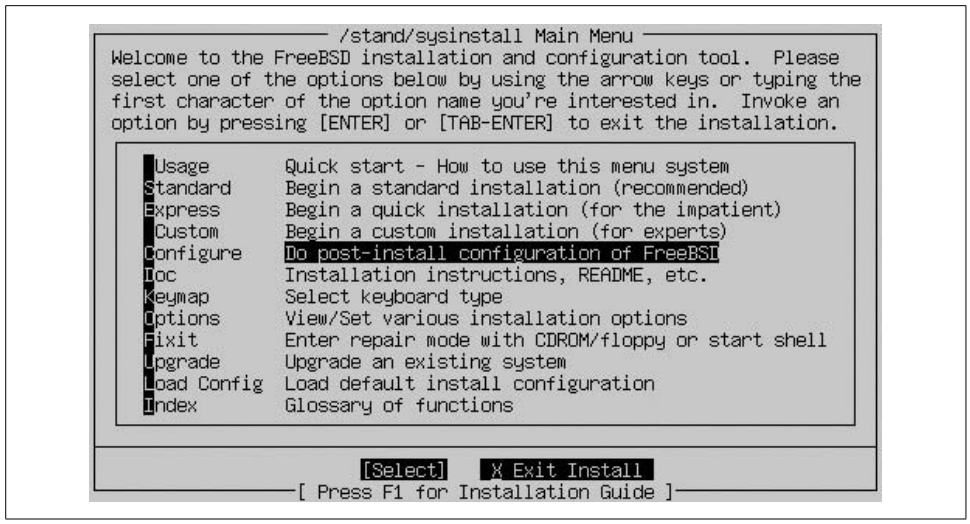


Figure 1-8. The FreeBSD sysinstall facility

Both the Configure and Index menu items are of interest for general system administration tasks. The latter is especially useful in that it lists individually all the available operations the tool can perform.

Tru64: SysMan

The Tru64 operating system offers the SysMan facility. This tool is essentially menu driven despite the fact that it can run in various graphical environments, including via a Java 1.1–enabled browser. SysMan can run in two different modes, as shown in Figure 1-9: as a system administration utility for the local system or as a monitoring and management station for the network. These two modes of operations are selected with the `sysman` command's `-menu` and `-station` options, respectively; `-menu` is the default.

This utility does not have any command preview or logging features, but it does have a variety of "accelerators": keywords that can be used to initiate a session at a particular menu point. For example, `sysman shutdown` takes you directly to the system shutdown dialog. Use the command `sysman -list` to obtain a complete list of all defined accelerators.

One final note: the `insightd` daemon must be running in order to be able to access the SysMan online help.

Other Freely Available Administration Tools

The freely available operating systems often provide some additional administrative tools as part of the various window manager packages that they include. For example, both the Gnome and KDE desktop environments include several administrative

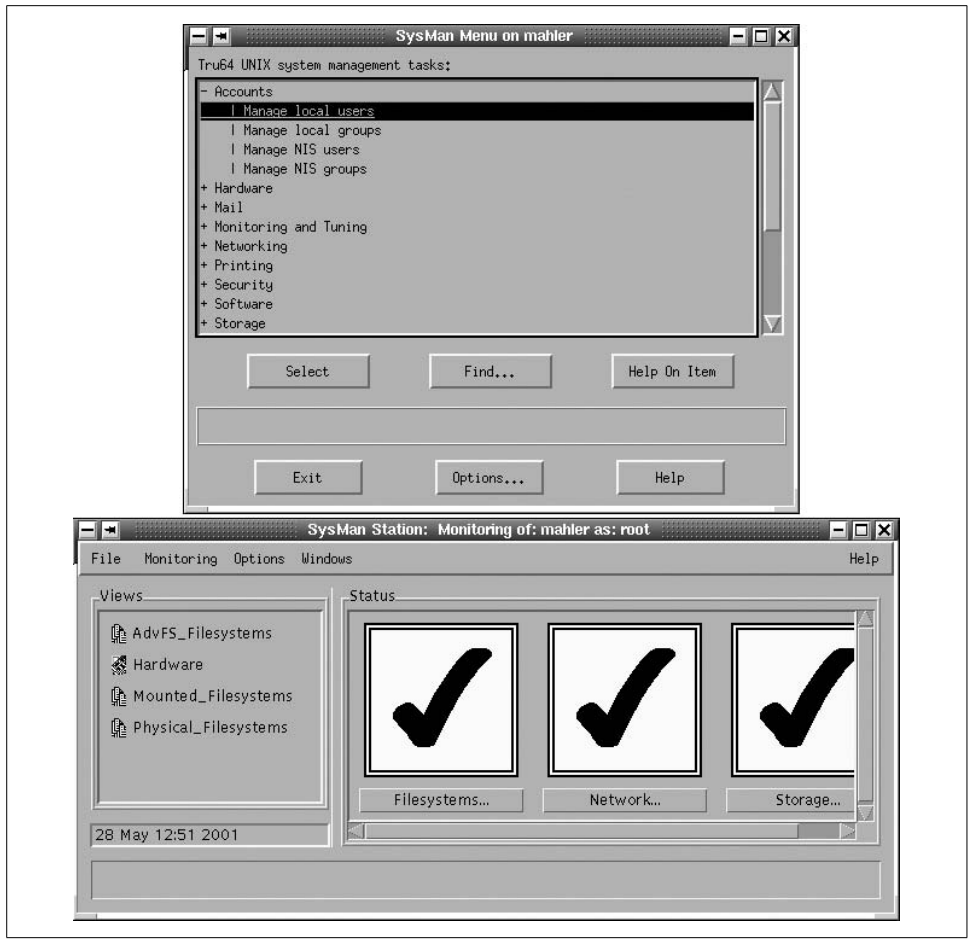


Figure 1-9. The SysMan facility

applets and utilities. Those available under KDE on a SuSE Linux system are illustrated in Figure 1-10.

We will consider some of the best of these tools from time to time in this book.

The Ximian Setup Tools

The Ximian project brings together the latest release of the Gnome desktop, the Red Carpet web-based system software update facility, and several other items into what is designed to be a commercial-quality desktop environment. As of this writing, it is available for several Linux distributions and for Solaris systems. Additional ports, including to BSD, are planned for the future.

The Ximian Setup Tools are a series of applets designed to facilitate system administration, ultimately in a multiplatform environment. Current modules allow you to

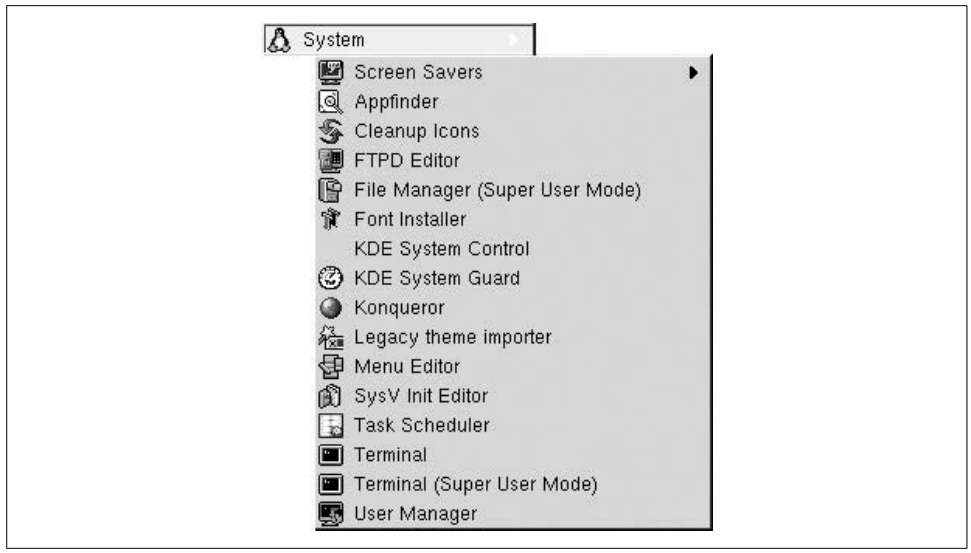


Figure 1-10. KDE administrative tools on a SuSE Linux system

administer boot setup (i.e., kernel selection), disks, swap space, users, basic networking, shared filesystems, printing, and the system time. The applet for the latter is illustrated in Figure 1-11.

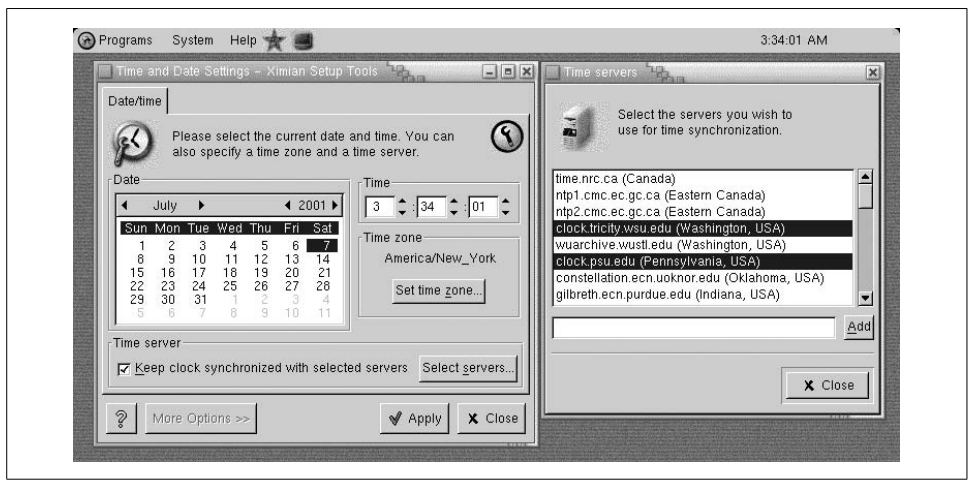


Figure 1-11. The Ximian Setup Tools

This applet, even in this early incarnation, goes well beyond a simple dialog allowing you to set the current date and time; it also allows you to specify time servers for Internet-based time synchronization. The other tools are of similar quality, and the package seems very promising for those who want GUI-based system administration tools.

VNC

I'll close this section by briefly looking at one additional administrative tool that can be of great use for remote administration, especially in a heterogeneous environment. It is called VNC, which stands for "virtual network computing." The package is available for a wide variety of Unix systems* at *http://www.uk.research.att.com/vnc/*. It is shown in Figure 1-12.

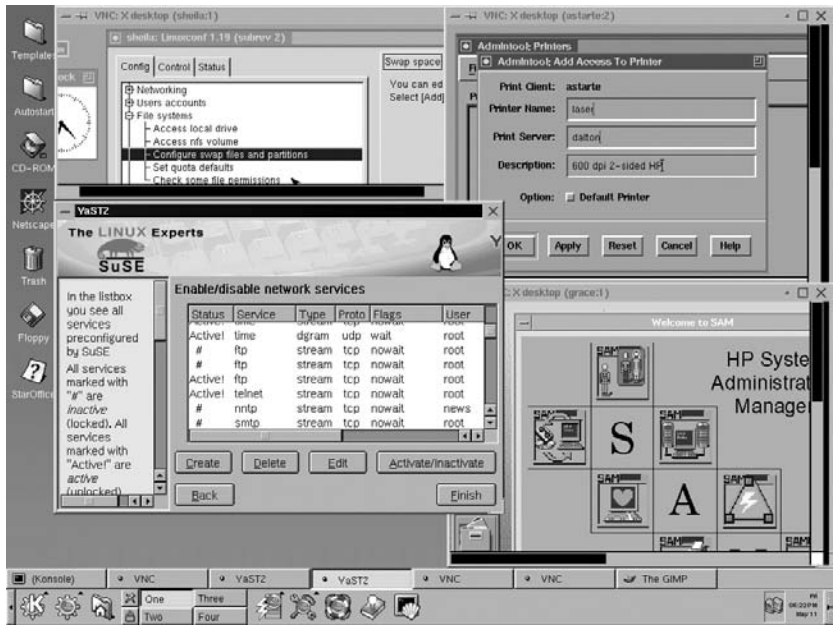


Figure 1-12. Using VNC for remote system administration

The illustration depicts the entire desktop on a SuSE Linux system. You can see several of its icons along the left edge, as well as the tool bar at the bottom of the screen (where you can determine that it is running the KDE window manager).

The four open windows are three individual VNC sessions to different remote computers, each running a different operating system and a local YaST session. Beginning at the upper left and moving clockwise, the remote sessions are a Red Hat Linux system (Linuxconf is open), a Solaris system (we can see `admintool`), and an HP-UX system (running SAM).

VNC has a couple of advantages over remote application sessions displayed via the X Windows system:

* Official binary versions of the various tools are available for a few systems on the main web page. In addition, consult the *contrib* area for ports to additional systems. It is also usually easy to build the tools from source code.

- With VNC you see the entire desktop, not just one application window. Thus, you can access applications via the remote system's own icons and menus (which may be much less convenient to initiate via commands).
- You eliminate missing font issues and many other display and resource problems, because you are using the X server on the remote system to generate the display images rather than the one on the local system.

In order to use VNC, you must download the software and build or install the five executables that comprise it (conventionally, they are placed in */usr/local/bin*). Then you must start a server process on systems that you want to administer remotely, using the `vncserver` command:

```
garden-$ vncserver
You will require a password to access your desktops.

Password:    Not echoed.
Verify:

New 'X' desktop is garden:1

Creating default startup script /home/chavez/.vnc/xstartup
Starting applications specified in /home/chavez/.vnc/xstartup
Log file is /home/chavez/.vnc/garden:1.log
```

This example starts a server on host *garden*. The first time you run the `vncserver` command, you will be asked for a password. This password, which is independent of your normal Unix password, will be required in order to connect to the server.

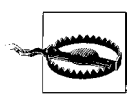
Once the server is running, you connect to it by running the `vncviewer` command. In this example, we connect to the `vncserver` on *garden*:

```
desert-$ vncviewer garden:1
```

The parameter given is the same as was indicated when the server was started. VNC allows multiple servers to be running simultaneously.

In order to shut down a VNC server, execute a command like this one on the remote system (i.e., the system where the server was started):

```
garden-$ vncserver -kill :1
```



Only the VNC server password is required for connection. Usernames are not checked, so an ordinary user can connect to a server started by *root* if she knows the proper password. Therefore, it is important to select strong passwords for the server password (see "Administering User Passwords" in Chapter 6) and to use a different password from the normal one if such cross-user connections are needed.

Additionally, VNC passwords are sent in plain text over the network. Thus, using VNC is problematic on an insecure network. In such circumstances, VNC traffic can be encrypted by tunneling it through a secure protocol, such as SSH.

Where Does the Time Go?

We'll close this chapter with a brief look at a nice utility that can be useful for keeping track of how you spend your time, information that system administrators will find comes in handy all too often. It is called `plod` and was written by Hal Pomeranz (see *http://bullwinkle.deer-run.com/~hal/plod/*). While there are similar utilities with a GUI interface (e.g., `gtt` and `karm`, from the Gnome and KDE window manager packages, respectively), I prefer this simpler one that doesn't require a graphical environment.

`plod` works by maintaining a log file containing time stamped entries that you provide; the files' default location is *~/.logdir/yyyymm*, where *yyyy* and *mm* indicate the current year and month, respectively. `plod` log files can optionally be encrypted.

The command has lots of options, but its simplest form is the following:

```
$ plod [text]
```

If some text is included on the command, it is written to the log file (tagged with the current date and time). Otherwise, you enter the command's interactive mode, in which you can type in the desired text. Input ends with a line containing a lone period.

Once you've accumulated some log entries, you can use the command's `-C`, `-P`, and `-E` options to display them, either as continuous output, piped through a paging command like `more` (although `less` is the default), or via an editor (`vi` is the default). You can specify a different paging program or editor with the *PAGER* and *EDITOR* environment variables (respectively).

You can also use the `-G` option to search `plod` log files; it differs from `grep` in that matching entries are displayed in their entirety. By default, searches are not case sensitive, but you can use `-g` to make them so.

Here is an example command that searches the current log file:

```
$ plod -g hp-ux
-----
05/11/2001, 22:56 --
Starting to configure the new HP-UX box.
-----
05/11/2001, 23:44 --
Finished configuring the new HP-UX box.
```

Given these features, `plod` can be used to record and categorize the various tasks that you perform. We will look at a script which can read and summarize `plod` data in Chapter 14.

CHAPTER 2
The Unix Way

It's easy to identify the most important issues and concerns system managers face, regardless of the type of computers they have. Almost every system manager has to deal with user accounts, system startup and shutdown, peripheral devices, system performance, security—the list could go on and on. While the commands and procedures you use in each of these areas vary widely across different computer systems, the general approach to such issues can be remarkably similar. For example, the process of adding users to a system has the same basic shape everywhere: add the user to the user account database, allocate some disk space for him, assign a password to the account, enable him to use major system facilities and applications, and so on. Only the commands to perform these tasks are different on different systems.

In other cases, however, even the *approach* to an administrative task or issue will change from one computer system to the next. For example, "mounting disks" doesn't mean the same thing on a Unix system that it does on a VMS or MVS system (where they're not always even called disks). No matter what operating system you're using—Unix, Windows 2000, MVS—you need to know something about what's happening inside, at least more than an ordinary user does.

Like it or not, a system administrator is generally called on to be the resident guru. If you're responsible for a multiuser system, you'll need to be able to answer user questions, come up with solutions to problems that are more than just band-aids, and more. Even if you're responsible only for your own workstation, you'll find yourself dealing with aspects of the computer's operation that most ordinary users can simply ignore. In either case, you need to know a fair amount about how Unix really works, both to manage your system and to navigate the eccentric and sometimes confusing byways of the often jargon-ridden technical documentation.

This chapter will explore the Unix approach to some basic computer entities: files, processes, and devices. In each case, I will discuss how the Unix approach affects system administration procedures and objectives. The chapter concludes with an overview of the standard Unix directory structure.

If you have managed non-Unix computer systems, this chapter will serve as a bridge between the administrative concepts you know and the specifics of Unix. If you have some familiarity with user-level Unix commands, this chapter will show you their place in the underlying operating system structure, enabling you to place them in an administrative context. If you're already familiar with things like file modes, inodes, special files, and fork-and-exec, you can probably skip this chapter.

Files

Files are central to Unix in ways that are not true for some other operating systems. Commands are executable files, usually stored in standard locations in the directory tree. System privileges and permissions are controlled in large part via access to files. Device I/O and file I/O are distinguished only at the lowest level. Even most interprocess communication occurs via file-like entities. Accordingly, the Unix view of files and its standard directory structure are among the first things a new administrator needs to know about.

Like all modern operating systems, Unix has a hierarchical (tree-structured) directory organization, know collectively as the *filesystem*.* The base of this tree is a directory called the *root directory*. The root directory has the special name / (the forward slash character). On Unix systems, all user-available disk space is transparently combined into a single directory tree under /, and the physical disk a file resides on is not part of a Unix file specification. We'll discuss this topic in more detail later in this chapter.

Access to files is organized around file ownership and protection. Security on a Unix system depends to a large extent on the interplay between the ownership and protection settings on its files and the system's user account and group† structure (as well as factors like physical access to the machine). The following sections discuss the basic principles of Unix file ownership and protection.

File Ownership

Unix file ownership is a bit more complex than it is under some other operating systems. You are undoubtedly familiar with the basic concept of a file having an owner: typically, the user who created it and has control over it. On Unix systems, files have two owners: a user owner and a group owner. What is unusual about Unix file ownership is that these two owners are decoupled. A file's group ownership is independent of the user who owns it. In other words, although a file's group owner is often,

* Or *file system*—the two forms refer to the same thing. To make things even more ambiguous, these terms are also used to refer to the collection of files on an individual formatted disk partition.

† On Unix systems, individual user accounts are organized into *groups*. Groups are simply collections of users, defined by the entries in */etc/passwd* and */etc/group*. The mechanics of defining groups and designating users as members of them are described in Chapter 6. Using groups effectively to enhance system security is discussed in Chapter 7.

perhaps even usually, the same as the group its user owner belongs to, this is not required. In fact, the user owner of a file does need not even need to be a member of the group that owns it. There is no necessary connection between them at all. In such a case, when file access is specified for a file's group owner, it applies to members of that group and not to other members of its user owner's group, who are treated simply as part of "other": the rest of the world.

The motivation behind this group ownership of files is to allow file protections and permissions to be organized according to your needs. The key point here is flexibility. Because Unix lets users be in more than one group, you are free to create groups as you need them. Files can be made accessible to almost completely arbitrary collections of the system's users. Group file ownership means that giving someone access to an entire set of files and commands is as simple as adding her to the group that owns them; similarly, taking access away from someone else involves removing her from the relevant group.

To consider a more concrete example, suppose user *chavez,* who is in the *chem* group, needs access to some files usually used by the *physics* group. There are several ways you can give her access:

- Make copies of the files for her. If they change, however, her copies will need to be updated. And if she needs to make changes too, it will be hard to avoid ending up with two versions that need to be merged together. (Because of inconveniences like these, this choice is seldom taken.)
- Make the files world-readable. The disadvantage of this approach is that it opens up the possibility that someone you don't want to look at the files will see them.
- Make *chavez* a member of the *physics* group. This is the best alternative and also the simplest. It involves changing only the group configuration file. The file permissions don't need to be modified at all, since they already allow access for *physics* group members.

Displaying file ownership

To display a file's user and group ownership, use the long form of the `ls` command by including the `-l` option (`-lg` under Solaris):

```
$ ls -l
-rwxr-xr-x  1 root     system      120   Mar 12 09:32  bronze
-r--r--r--  1 chavez   chem         84   Feb 28 21:43  gold
-rw-rw-r--  1 chavez   physics   12842   Oct 24 12:04  platinum
-rw-------  1 harvey   physics     512   Jan  2 16:10  silver
```

Columns three and four display the user and group owners for the listed files. For example, we can see that the file *bronze* is owned by user *root* and group *system*. The next two files are both owned by user *chavez,* but they have different group owners; *gold* is owned by group *chem*, while *platinum* is owned by group *physics.* The last file, *silver*, is owned by user *harvey* and group *physics*.

Who owns new files?

When a new file is created, its user owner is the user who creates it. On most Unix systems, the group owner is the current* group of the user who creates the file. However, on BSD-style systems, the group owner is the same as the group owner of the directory in which the file is created. Of the versions we are considering, FreeBSD and Tru64 Unix operate in the second manner by default.

Most current Unix versions, including all of those we are considering, allow a system to selectively use BSD-style group inheritance from the directory group ownership by setting the set group ID (setgid) attribute on the directory, which we discuss in more detail later in this chapter.

Changing file ownership

If you need to change the ownership of a file, use the `chown` and `chgrp` commands. The `chown` command changes the user owner of one or more files:

```
# chown new-owner files
```

where *new-owner* is the username (or user ID) of the new owner for the specified files. For example, to change the owner of the file *brass* to user *harvey*, execute this `chown` command:

```
# chown harvey brass
```

On most systems, only the superuser can run the `chown` command.

If you need to change the ownership of an entire directory tree, you can use the `-R` option (*R* for *recursive*). For example, the following command will change the user owner to *harvey* for the directory */home/iago/new/tgh* and all files and subdirectories contained underneath it:

```
# chown -R harvey /home/iago/new/tgh
```

You can also change both the user and group owner in a single operation, using this format:

```
# chown new-owner:new-group files
```

For example, to change the user owner to *chavez* and the group owner to *chem* for *chavez*'s home directory and all the files underneath it, use this command:

```
# chown -R chavez:chem /home/chavez
```

If you just want to change a file's group ownership, use the `chgrp` command:

```
$ chgrp new-group files
```

where *new-group* is the group name (or group ID) of the desired group owner for the specified files. `chgrp` also supports the `-R` option. Non-*root* users of `chgrp` must be

* See "Unix Users and Groups" in Chapter 6 for information about how the user's primary group is determined.

both the owner of the file and a member of the new group to change a file's group ownership (but need not be a member of its current group).

File Protection

Once ownership is set up properly, the next natural issue to consider is how to protect files from unwanted access (or the reverse: how to allow access to those people who need it). The protection on a file is referred to as its *file mode* on Unix systems. File modes are set with the `chmod` command; we'll look at `chmod` after discussing the file protection concepts it relies on.

Types of file and directory access

Unix supports three types of file access: read, write, and execute, designated by the letters *r*, *w*, and *x*, respectively. Table 2-1 shows the meanings of those access types.

Table 2-1. File access types

Access	Meaning for a file	Meaning for a directory
r	View file contents.	Search directory contents (e.g., use `ls`).
w	Alter file contents.	Alter directory contents (e.g., delete or rename files).
x	Run executable file.	Make it your current directory (`cd` to it).

The file access types are fairly straightforward. If you have read access to a file, you can see what's in it. If you have write access, you can change what's in it. If you have execute access and the file is a binary executable program, you can run it. To run a script, you need both read and execute access, since the shell has to read the commands to interpret them. When you run a compiled program, the operating system loads it into memory for you and begins execution, so you don't need read access yourself.

The corresponding meanings for directories may seem strange at first, but they do make sense. If you have execute access to a directory, you can `cd` to it (or include it in a path that you want to `cd` to). You can also access files in the directory by name. However, to list all the files in the directory (i.e., to run the `ls` command without any arguments), you also need read access to the directory. This is consistent because a directory is just a file whose contents are the names of the files it contains, along with information pointing to their disk locations. Thus, to `cd` to a directory, you need only execute access since you don't need to be able to read the directory file itself. In contrast, if you want to run any command lists or use files in the directory via an explicit or implicit wildcard—e.g., `ls` without arguments or `cat *.dat`—you do need read access to the directory file itself to expand the wildcards.

Table 2-2 illustrates the workings of these various access types by listing some sample commands and the minimum access you would need to successfully execute them.

Table 2-2. File protection examples

Command	Minimum access needed	
	On file itself	On directory file is in
`cd /home/chavez`	N/A	*x*
`ls /home/chavez/*.c`	(none) *r*	*r* *x*
`ls -l /home/chavez/*.c`	(none) *r*	*rx* *x*
`cat myfile`	*r*	*x*
`cat >>myfile`	*w*	*x*
runme (executable)	*x*	*x*
cleanup.sh (script)	*rx*	*x*
`rm myfile`	(none)	*wx*

Some items in this list are worth a second look. For example, when you don't have access to any of the component files, you still need only read access to a directory in order to do a simple `ls`; if you include `-l` (or any other option that lists file sizes), you also need execute access to the directory. This is because the file sizes must be determined from the disk information, an action which implicitly changes the directory in question. In general, any operation that involves more than simply reading the list of filenames from the directory file is going to require execute access if you don't have access to the relevant files themselves.

Note especially that write access on a file *is not required* to delete it; write access to the directory where the file resides is sufficient (although in this case, you'll be asked whether to override the protection on the file):

```
$ rm copper
rm: override protection 440 for copper? y
```

If you answer yes, the file will be deleted (the default response is no). Why does this work? Because deleting a file actually means removing its entry from the directory file (among other things), which is a form of altering the directory file, for which you need only write access to the directory. The moral is that write access to directories is very powerful and should be granted with care.

Given these considerations, we can summarize the different options for protecting directories as shown in Table 2-3.

Table 2-3. Directory protection summary

Access granted	Resulting availability
--- (no access)	Does not allow any activity of any kind within the directory or any of its subdirectories.
r-- (read access only)	Allows users to list the names of the files in the directory, but does not reveal any of their attributes (i.e., size, ownership, mode, and so on).

Table 2-3. Directory protection summary (continued)

Access granted	Resulting availability
--x (execute access only)	Lets users work with programs in the directory specified by full pathname, but hides all other files.
r-x (read and execute access)	Lets users work with programs in the directory and list the contents of the directory, but does not allow them to create or delete files in the directory.
-wx (write and execute access)	Used for a drop-box directory. Users can change to the directory and leave files there, but can't discover the names of files placed there by others. The sticky bit is also usually set on such directories (see below).
rwx (full access)	Lets users work with programs in the directory, look at the contents of the directory, and create or delete files in the directory.

Access classes

Unix defines three basic classes of file access for which protection may be specified separately:

User access (u)
: Access granted to the owner of the file.

Group access (g)
: Access granted to members of the same group as the group owner of the file (but does not apply to the owner himself, even if he is a member of this group).

Other access (o)
: Access granted to all other normal users.

Unix file protection specifies the access types available to members of each of the three access classes for the file or directory.

The long version of the `ls` command also displays file permissions in addition to user and group ownership:

```
$ ls -l
-rwxr-xr-x  1 root     system     120 Mar 12 09:32  bronze
-r--r--r--  1 chavez   chem        84 Feb 28 21:43  gold
-rw-rw-r--  1 chavez   physics  12842 Oct 24 12:04  platinum
```

The set of letters and hyphens at the beginning of each line represents the file's mode. The 10 characters are interpreted as indicated in Table 2-4.

Table 2-4. Interpreting mode strings

		User access			Group access			Other access		
File	type 1	read 2	write 3	exec 4	read 5	write 6	exec 7	read 8	write 9	exec 10
bronze	-	r	w	x	r	-	x	r	-	x
gold	-	r	-	-	r	-	-	r	-	-
platinum	-	r	w	-	r	w	-	r	-	-
/etc/passwd	-	r	w	-	r	-	-	r	-	-

Table 2-4. Interpreting mode strings (continued)

File	type 1	User access read 2	User access write 3	User access exec 4	Group access read 5	Group access write 6	Group access exec 7	Other access read 8	Other access write 9	Other access exec 10
/etc/shadow	-	r	-	-	-	-	-	-	-	-
/etc/inittab	-	r	w	-	r	w	-	r	-	-
/bin/sh	-	r	-	x	r	-	x	r	-	x
/tmp	d	r	w	x	r	w	x	r	w	t

The first character indicates the file type: a hyphen indicates a plain file, and a *d* indicates a directory (other possibilities are discussed later in this chapter). The remaining nine characters are arranged in three groups of three. Moving from left to right, the groups represent user, group, and other access. Within each group, the first character denotes read access, the second character write access, and the third character execute access. If a certain type of access is allowed, its code letter appears in the proper position within the triad; if it is not granted, a hyphen appears instead.

For example, in the previous listing, read access and no other is granted for all users on the file *gold.* On the file *bronze,* the owner—in this case, *root*—is allowed read, write, and execute access, while all other users are allowed only read and execute access. Finally, for the file *platinum,* the owner (*chavez*) and all members of the group *physics* are allowed read and write access, while everyone else is granted only read access.

The remaining entries in Table 2-4 (below the line) are additional examples illustrating the usual protections for various common system files.

Setting file protection

The `chmod` command is used to specify the access mode for files:

```
$ chmod access-string files
```

`chmod`'s second argument is an *access string*, which states the permissions you want to set (or remove) for the listed files. It has three parts: the code for one or more access classes, the operator, and the code for one or more access types.

Figure 2-1 illustrates the structure of an access string. To create an access string, you choose one or more codes from the access class column, one operator from the middle column, and one or more access types from the third column. Then you concatenate them into a single string (no spaces). For example, the access string *u+w* says to add write access for the user owner of the file. Thus, to add write access for yourself for a file you own (*lead,* for example), use:

```
$ chmod u+w lead
```

To add write access for everybody, use the *all* access class:

```
$ chmod a+w lead
```

To remove write access, use a minus sign instead of a plus sign:

```
$ chmod a-w lead
```

This command sets the permissions on the file *lead* to allow only read access for all users:

```
$ chmod a=r lead
```

If execute or write access had previously been set for any access class, executing this command removes it.

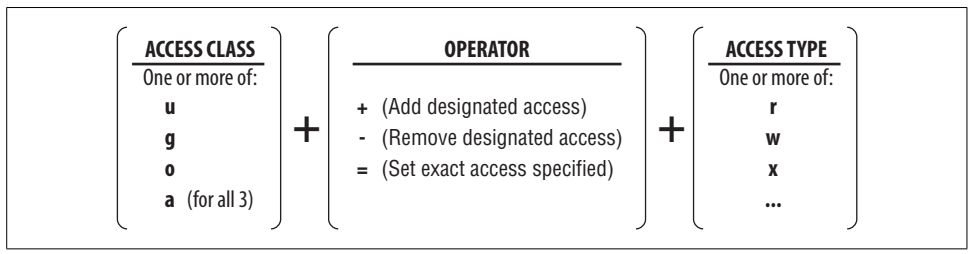


Figure 2-1. Constructing an access string for chmod

You can specify more than one access type and more than one access class. For example, the access string *g-rw* says to remove read and write access from the group access. The access string *go=r* says to set the group and other access to read-only (no execute access, no write access), changing the current setting as needed. And the access string *go+rx* says to add both read and execute access for both group and other users.

You can also include more than one set of operation–access type pairs for any given access class specification. For example, the access string *u+x-w* adds execute access and removes write access for the user owner. You can combine multiple access strings by separating them with commas (no spaces between them). Thus, the following command adds write access for the file owner and removes write access and adds read access for the group and other classes for the files *bronze* and *brass*:

```
$ chmod u+w,og+r-w bronze brass
```

The `chmod` command supports a *recursive* option (`-R`), to change the mode of a directory and all files under it. For example, if user *chavez* wants to protect all the files under her home directory from everyone else, she can use the command:

```
$ chmod -R go-rwx /home/chavez
```

Beyond the basics

So far, this discussion has undoubtedly made `chmod` seem more rigid than it actually is. In reality, it is a very flexible command. For example, both the access class and the access type may be omitted under some circumstances.

When the access class is omitted, it defaults to *a*. For example, the following command grants read access to all users for the current directory and every file under it:

```
$ chmod -R +r .
```

On some systems, this form operates slightly differently than a `chmod a+r` command. When the *a* access class is omitted, the specified permissions are compared against the default permissions currently in effect (i.e., as specified by the umask). When there is disagreement between them, the current default permissions take precedence. We'll look at this in more detail when we consider the umask a bit later.

The access string may be omitted altogether when using the = operator; this form has the effect of removing all access. For example, this command prevents any access to the file *lead* by anyone other than its owner:

```
$ chmod go= lead
```

Similarly, the form `chmod =` may be used to remove all access from a file (subject to constraints on some systems, to be discussed shortly).

The *X* access type grants execute access to the specified access classes only when execute access is already set for some access class. A typical use for this access type is to grant group or other read and execute access to all the directories and executable files within a subtree while granting only read access to all other types of files (the first group will all presumably have user execute access set). For example:

```
$ ls -lF
-rw-------   1 chavez chem609 Nov 29 14:31 data_file.txt
drwx------   2 chavez chem512 Nov 29 18:23 more_stuff/
-rwx------   1 chavez chem161 Nov 29 18:23 run_me*
$ chmod go+rX *
$ ls -lF
-rw-r--r--   1 chavez chem609 Nov 29 14:31 data_file.txt
drwxr-xr-x   2 chavez chem512 Nov 29 18:23 more_stuff/
-rwxr-xr-x   1 chavez chem161 Nov 29 18:23 run_me*
```

By specifying *X*, we avoid making *data_file.txt* executable, which would be a mistake.

`chmod` also supports the *u*, *g*, and *o* access *types*, which may be used as a shorthand form for the corresponding class's current settings (determined separately for each specified file). For example, this command makes the other access the same as the current group access for each file in the current directory:

```
$ chmod o=g *
```

If you like thinking in octal, or if you've been around Unix a long time, you may find numeric modes more convenient than incantations like *go+rX*. Numeric modes are described in the next section.

Specifying numeric file modes

The method just described for specifying file modes uses *symbolic* modes, since code letters are used to refer to each access class and type. The mode may also be set as an *absolute* mode by converting the symbolic representation used by `ls` to a numeric form. Each access triad (for a different user class) is converted to a single digit by setting each individual character in the triad to 1 or 0, depending on whether that type of access is permitted or not, and then taking the resulting three-digit binary number and converting it to an integer (which will be between 0 and 7). Here is a sample conversion:

	user			group			other		
Mode	r	w	x	r	-	x	r	-	-
Convert to binary	1	1	1	1	0	1	1	0	0
Convert to octal digit		7			5			4	
Corresponding absolute mode					754				

To set the protection on a file to match those above, you specify the numeric file mode 754 to `chmod` as the access string:

```
$ chmod 754 pewter
```

Specifying the default file mode

You can use the `umask` command to specify the default mode for newly created files. Its argument is a three-digit numeric mode that represents the access to be *inhibited*—masked out—when a file is created. Thus, the value is the octal complement of the desired numeric file mode.

If masks confuse, you can compute the umask value by subtracting the numeric access mode you want to assign from 777. For example, to obtain the mode 754 by default, compute 777 – 754 = 023; this is the value you give to `umask`:

```
$ umask 023
```

Note that leading zeros are included to make the mask three digits long.

Once this command is executed, all future files created are given this protection automatically. You usually put a `umask` command in the system-wide login initialization file and in the individual login initialization files you give to users when you create their accounts (see Chapter 6).

As we mentioned earlier, the `chmod` command's actions are affected by the default permissions when no explicit access class is specified, as in this example:

```
% chmod +rx *
```

In such cases, the current umask is taken into account before the file access mode is changed. More specifically, an individual access permission is not changed unless the umask allows it to be set.

It takes a concrete example to fully appreciate this aspect of chmod:

```
$ umask          Displays the current value.
23
$ ls -l gold silver
----------   1 chavez  chem      609 Oct 24 14:31  gold
-rwxrwxrwx   1 chavez  chem    12874 Oct 22 23:14  silver
$ chmod +rwx gold
$ chmod -rwx silver
$ ls -l gold silver
-rwxr-xr--   1 chavez  chem      609 Nov 12 09:04  gold
-----w--wx   1 chavez  chem    12874 Nov 12 09:04  silver
```

The current umask of 023 allows all access for the user, read and execute access for the group, and read-only access for other users. Thus, the first chmod command acts as one would expect, setting access in accordance with what is allowed by the umask. However, the interaction between the current umask and chmod's "–" operator may seem somewhat bizarre. The second chmod command clears only those access bits that are *permitted* by the umask; in this case, write access for group and write and execute access for other remain turned on.

Special-purpose access modes

The simple file access modes described previously do not exhaust the Unix possibilities. Table 2-5 lists the other defined file modes.

Table 2-5. Special-purpose access modes

Code	Name	Meaning
t	save text mode, sticky bit	Files: Keep executable in memory after exit. Directories: Restrict deletions to each user's own files.
s	setuid bit	*Files*: Set process user ID on execution.
s	setgid bit	Files: Set process group ID on execution. Directories: New files inherit directory group owner.
l	file locking	Files: Set mandatory file locking on reads/writes (Solaris and Tru64 and sometimes Linux). This mode is set via the group access type and requires that group execute access is off. Displayed as S in `ls -l` listings.

The *t* access type turns on the *sticky bit* (the formal name is *save text mode*, which is where the *t* comes from). For files, this traditionally told the Unix operating system to keep an executable image in memory even after the process that was using it had exited. This feature is seldom implemented in current Unix implementations. It was designed to minimize startup overhead for frequently used programs like vi. We'll consider the sticky bit on directories below.

When the set user ID (setuid) or set group ID (setgid) access mode is set on an executable file, processes that run it are granted access to system resources based upon the file's user or group owner, rather than based on the user who created the process. We'll consider these access modes in detail later in this chapter.

Save-text access on directories

The sticky bit has a different meaning when it is set on directories. If the sticky bit is set on a directory, a user may only delete files that she owns or for which she has explicit write permission granted, even when she has write access to the directory (thus overriding the default Unix behavior). This feature is designed to be used with directories like */tmp,* which are world-writable, but in which it may not be desirable to allow any user to delete files at will.

The sticky bit is set using the user access class. For example, to turn on the sticky bit on */tmp,* use this command:

```
# chmod u+t /tmp
```

Oddly, Unix displays the sticky bit as a "t" in the other execute access slot in long directory listings:

```
$ ls -ld /tmp
drwxrwxrwt   2 root         8704  Mar 21 00:37  /tmp
```

Setgid access on directories

Setgid access on a directory has a special meaning. When this mode is set, it means that files created in that directory will have the same group ownership as the directory itself (rather than the user owner's primary group), emulating the default behavior on BSD-based systems (FreeBSD and Tru64). This approach is useful when you have groups of users who need to share a lot of files. Having them work from a common directory with the setgid attribute means that correct group ownership will be automatically set for new files, even if the people in the group don't share the same primary group.

To place setgid access on a directory, use a command like this one:

```
# chmod g+s /pub/chem2
```

Numerical equivalents for special access modes

The special access modes can also be set numerically. They are set via an additional octal digit prepended to the mode whose bits correspond to the sticky bit (lowest bit: 1), setgid/file locking (middle bit: 2), and setuid (high bit: 4). Here are some examples:

```
# chmod 4755 uid        Setuid access
# chmod 2755 gid        Setgid access
# chmod 6755 both       Setuid and setgid access: 2 highest bits on
# chmod 1777 sticky     Sticky bit
# chmod 2745 locking    File locking (note that group execute is off)
# ls -ld
-rwsr-sr-x   1 root  chem           0 Mar 30 11:37 both
-rwxr-sr-x   1 root  chem           0 Mar 30 11:37 gid
-rwxr-Sr-x   1 root  chem           0 Mar 30 11:37 locking
drwxrwxrwt   2 root  chem        8192 Mar 30 11:39 sticky
-rwsr-xr-x   1 root  chem           0 Mar 30 11:37 uid
```

How to Recognize a File Access Problem

My first rule of thumb about any user problem that comes up is this: it's usually a file ownership or protection problem.* Seriously, though, the majority of the problems users encounter that aren't the result of hardware problems really are file access problems. One classic tip-off of a file protection problem is something that worked yesterday, or last week, or even last year, but doesn't today. Another clue is that something works differently for *root* than it does for other users.

In order to work properly, programs and commands must have access to the input and output files they use, any scratch areas they access, and any permanent files they rely on, including the special files in */dev* (which act as device interfaces).

When such a problem arises, it can come from either the file permissions being wrong or the protection being correct but the ownership (user and/or group) being wrong.

The trickiest problem of this sort I've ever seen was at a customer site where I was conducting a user training course. Suddenly, their main text editor, which happened to be a clone of the VAX/VMS editor EDT, just stopped working. It seemed to start up fine, but then it would bomb out when it got to its initialization file. But the editor worked without a hitch when *root* ran it. The system administrator admitted to "changing a few things" the previous weekend but didn't remember exactly what. I checked the protections on everything I could think of, but found nothing. I even checked the special files corresponding to the physical disks in */dev*. My company ultimately had to send out a debugging version of the editor, and the culprit turned out to be */dev/null*, which the system administrator had decided needed protecting against random users!

There are at least three morals to this story:

- For the local administrator: *always* test every change before going on to the next one—multiple, random changes almost always wreak havoc. Writing them down as you do them also makes troubleshooting easier.
- For me: if you *know* it's a protection problem, check the permissions on *everything*.
- For the programmer who wrote the editor: *always* check the return value of system calls (but that's another book).

If you suspect a file protection problem, try running the command or program as *root*. If it works fine, it's almost certainly a protection problem.

A common, inadvertent way of creating file ownership problems is by accidentally editing files as *root*. When you save the file, the file's owner is changed by some editors. The most obscure variation on this effect that I've heard of is this: someone was

* At least, this was the case before the Internet.

editing a file as *root* using an editor that automatically creates backup files whenever the edited file is saved. Creating a backup file meant writing a new file to the directory holding the original file. This caused the ownership on the *directory* to be set to *root*.* Since this happened in the directory used by UUCP (the Unix-to-Unix copy facility), and correct file and directory ownership are crucial for UUCP to function, what at first seemed to be an innocuous change to an inconsequential file broke an entire Unix subsystem. Running `chown uucp` on the directory fixed everything again.

Mapping Files to Disks

This section will change our focus from files as objects to files as collections of data on disk. Users need not be aware of the actual disk locations of files they access, but administrators need to have at least a basic conception of how Unix maps files to disk blocks in order to understand the different file types and the purpose and functioning of the various filesystem commands.

An *inode* (pronounced "eye-node") is the data structure on disk that describes and stores a file's attributes, including its physical location on disk. When a filesystem is initially created, a specific number of inodes are created. In most cases, this becomes the maximum number of files of all types, including directories, special files, and links (discussed later) that can exist in the filesystem. A typical formula is one inode for every 8 KB of actual file storage. This is more than sufficient in most situations.† Inodes are given unique numbers, and each distinct file has its own inode. When a new file is created, an unused inode is assigned to it.

Information stored in inodes includes the following:

- User owner and group owner IDs.
- File type (regular, directory, etc., or 0 if the inode is unused).
- Access modes (permissions).
- Most recent inode modification, data access, and data modification times. If the file's metadata does not change, the first item will correspond to the file creation time.

* Clearly, the system itself was somewhat "broken" as well, since adding a file to a directory should never change the directory's ownership. However, it is also possible to do this accidentally with text editors that allow you to edit a directory.

† There are a couple of circumstances where this may not hold. One is a filesystem containing an enormous number of very small files. The traditional example of this is the USENET news spool directory tree (although some modern news servers now use a better storage scheme). News files are typically both very small and inordinately numerous, and their numbers have been known to exceed normal inode limits. A second potential problem situation occurs with facilities that make extensive use of symbolic links for functions such as source code version control, again characterized by many, many tiny files. In such cases, you can run out of inodes before disk capacity is exhausted. You will want to take these factors into account when preparing the disk (see Chapter 10). At the other extreme, filesystems that are designed to hold only a few very large files might save a nontrivial amount of space by being configured with far fewer than the normal number of inodes.

- Number of hard links to the file (links are discussed later in this chapter). This is 0 if the inode is unused, and one for most regular files.
- Size of the file.
- Disk addresses of:
  - Disk locations for the data blocks that make up the file, and/or
  - Disk locations of disk blocks that hold the disk locations of the file's data blocks (*indirect blocks*), and/or
  - Disk locations of disk blocks that hold the disk locations of indirect blocks (*double indirect blocks*: two disk addresses removed from the actual data blocks).*

In short, inodes store all available information about the file except its name and directory location. The inodes themselves are stored elsewhere on disk.

On Unix systems, it is reasonably safe to say that "everything is a file": the operating system even represents I/O devices as files. Accordingly, there are several different kinds of files, each with a different function.

Regular files

Regular files are files containing data. They are normally called simply "files." These may be ASCII text files, binary data files, executable program binaries, program input or output, and so on.

Directories

A *directory* is a binary file consisting of a list of the other files it contains, possibly including other directories (try running `od -c` on one to see this). Directory entries are filename-inode number pairs. This is the mechanism by which inodes and directory locations are associated; the data on disk has no knowledge of its (purely logical) location within its filesystem.

Special files: character and block device files

Special files are the mechanism used for device I/O under Unix. They reside in the directory */dev* and its subdirectories, as well as the directory */devices* under Solaris.

Generally, there are two types of special files: *character special files*, corresponding to character-based or raw device access, and *block special files*, corresponding to block I/O device access. Character special files are used for unbuffered data transfers to and from a device (e.g., a terminal). In contrast, block special files are used when data is transferred in fixed-size chunks known as *blocks* (e.g., most file I/O). Both kinds of special files exist for some devices (including disks). Character special files

* In traditional System V filesystems, inode disk addresses can point to triple indirect blocks. FreeBSD also uses triple indirect blocks.

generally have names beginning with *r* (for "raw")—*/dev/rsd0a*, for example—or reside in subdirectories of */dev* whose names begin with *r*—*/dev/rdsk/c0t3d0s7*, for example. The corresponding block special files have the same name, minus the initial *r*: */dev/disk0a*, */dev/dsk/c0t3d0s7*. Special files are discussed in more detail in later in this chapter.

Links

A *link* is a mechanism that allows several filenames (actually, directory entries) to refer to a single file on disk. There are two kinds of links: hard links and symbolic or soft links. A hard link associates two (or more) filenames with the same inode. Hard links are separate directory entries that all share the same disk data blocks. For example, the command:

```
$ ln index hlink
```

creates an entry in the current directory named *hlink* with the same inode number as *index*, and the link count in the corresponding inode is increased by 1. Hard links may not span filesystems, because inode numbers are unique only within a filesystem. In addition, hard links should be used only for files and not for directories, and correctly implemented versions of `ln` won't let you create the latter.

Symbolic links, on the other hand, are pointer files that refer to a different file or directory elsewhere in the filesystem. Symbolic links may span filesystems, because they point to a Unix pathname, not to a specific inode.

Symbolic links are created with the `-s` option to `ln`.

The two types of links behave similarly, but they are not identical. As an example, consider a file *index* to which there is a hard link *hlink* and a symbolic link *slink*. Listing the contents using either name with a command like `cat` will result in the same output. For both *index* and *hlink*, the disk contents pointed to by the addresses in their common inode will be accessed and displayed. For *slink*, the disk contents referenced by the address in its inode contain the pathname for *index*; when it is followed, *index*'s inode will be accessed next, and finally its data blocks will be displayed.

In directory listings, *hlink* will be indistinguishable from *index*. Changes made to either file will affect both of them, since they share the same disk blocks. However, moving either file with the `mv` command will not affect the other one, since moving a file involves only altering a directory entry (keep in mind that pathnames are not stored in the inode). Similarly, deleting *index* will not affect *hlink*, which will still point to the same inode (the corresponding disk blocks are only freed when an inode's link count reaches zero).

If a new file in the current directory named *index* is subsequently created, there will be no connection between it and *hlink*, because when the new file is created, it will be assigned a free inode. Although they are initially created by referencing an existing file, hard links are linked only to an inode, not to the other file. In fact, all regular files are technically hard links (i.e., inodes with a link count ≥ 1).

In contrast, a symbolic link *slink* to *index* will behave differently. The symbolic link appears as a separate entry in directory listings, marked as a link with an "l" as the first character in the mode string:

```
% ls -l
-rw------- 2 chavez  chem  5228 Mar 12 11:36 index
-rw------- 2 chavez  chem  5228 Mar 12 11:36 hlink
lrwxrwxrwx 1 chavez  chem     5 Mar 12 11:37 slink -> index
```

Symbolic links are always very small files, while every hard link to a given file (inode) is exactly the same size (*hlink* is naturally the same length as *index*).

Changes made by referencing either the real filename or the symbolic link will affect the contents of *index*. Deleting *index* will also break the symbolic link; *slink* will point nowhere. But if another file *index* is subsequently recreated, *slink* will once again be linked to it.* Deleting *slink* will have no effect on *index*.

Figure 2-2 illustrates the differences between hard and symbolic links. In the first picture, *index* and *hlink* share the inode N1 and its associated data blocks. The symbolic link *slink* has a different inode, N2, and therefore different data blocks. The contents of inode N2's data blocks refer to the pathname to *index*.† Thus, accessing *slink* eventually reaches the data blocks for inode N1.

When *index* is deleted (in the second picture), *hlink* is associated with inode N1 by its own directory entry. Accessing *slink* will generate an error, however, since the pathname it references does not exist. When a new *index* is created (in the third picture), its gets a new inode, N3. This new file clearly has no relationship to *hlink*, but it does act as the target for *slink*.

Using the `cd` command can be a bit tricky when dealing with symbolic links to directories, as these examples illustrate:

```
$ pwd; cd ./htdocs
/home/chavez
$ cd ../bin
../bin: No such file or directory.
$ pwd
/public/web2/apache/htdocs
$ ls -l /home/chavez/htdocs
lrwxrwxrwx   1 chavez chem   18 Mar 30 12:06 htdocs ->
                             /public/web2/apache/htdocs
```

The subdirectory *htdocs* in the current directory is a symbolic link (its target is indicated in the final command). Accordingly, the second `cd` command does not work as

* Symbolic links are actually interpreted only when accessed, so they can't really be said to point anywhere at other times. But conceptually, this is what they do.

† Some operating systems, including FreeBSD, store the target of the symbolic link in the inode itself, provided the target is short enough.

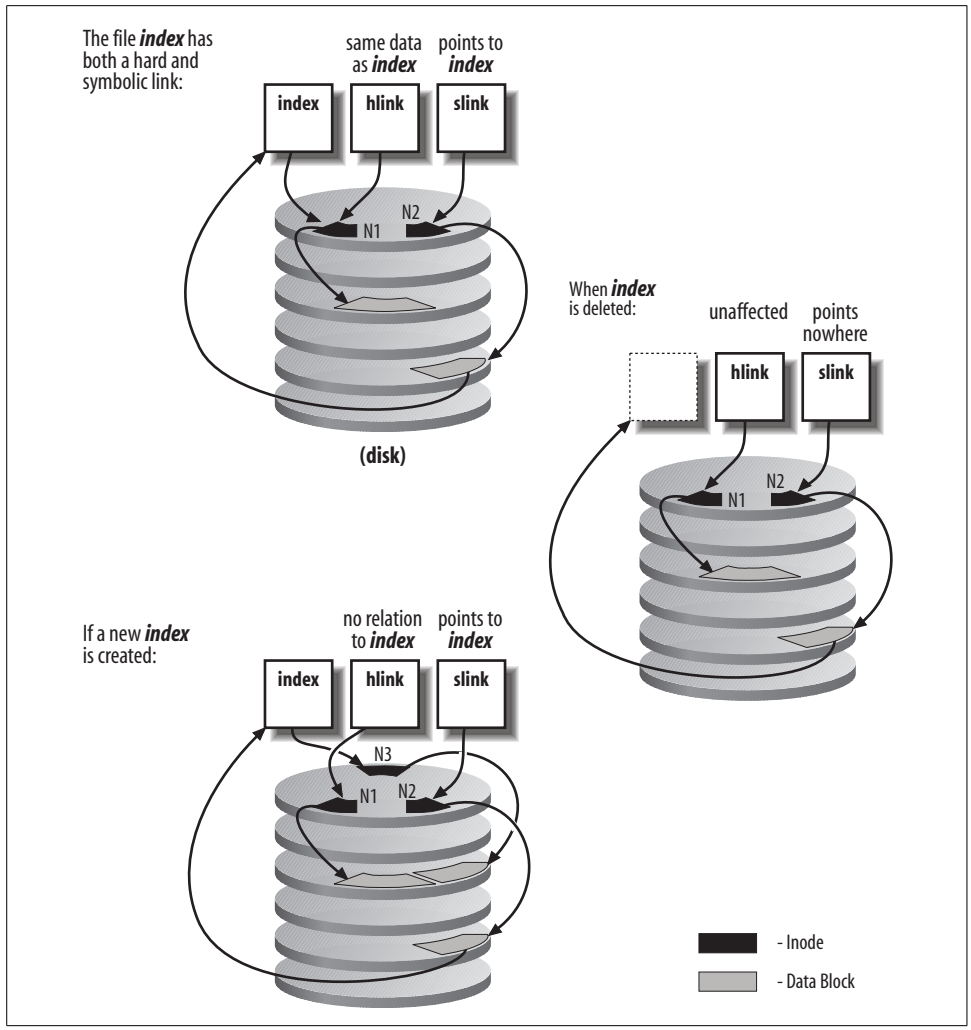


Figure 2-2. Comparing hard and symbolic links

expected, and the current directory does not change to */home/chavez/bin*. Similar effects would occur with a command like this one:

```
$ cd /home/chavez/htdocs/../cgi-bin; pwd
/public/web2/apache/cgi-bin
```

For more information about links, see the `ln` manual page, and experiment with creating and modifying linked files.

Tru64 Context-Dependent Symbolic Links. In a Tru64 clustered environment, many standard system files and directories are actually a type of symbolic link known as

context-dependent symbolic links (CDSLs). They are symbolic links with a variable component that is resolved to a specific cluster host at access time. For example, consider this directory listing (the output is wrapped to fit):

```
$ ls -lF /var/adm/c*
-rw-r--r--   1 root     system  91 May 30 13:07  cdsl_admin.inv
-rw-r--r--   1 root     adm    232 May 30 13:07  cdsl_check_list
lrwxr-xr-x   1 root     adm     43 Jan  3 12:09  collect.dated@ ->
                        ../cluster/members/{memb}/adm/collect.dated
lrwxr-xr-x   1 root     adm     35 Jan  3 12:04  crash@         ->
                        ../cluster/members/{memb}/adm/crash/
lrwxr-xr-x   1 root     adm     34 Jan  3 12:04  cron@          ->
                        ../cluster/members/{memb}/adm/cron/
```

The first two files are regular files that reside in the */var/adm* directory. The remaining three files are context-dependent symbolic links, indicated by the *{memb}* component. When such a file is accessed, this component is resolved to a directory named *member*n, where *n* indicates the host's number within the cluster.

Occasionally, you may need to create such a link. The `mkcdsl` command serves this purpose, as in this example (output is wrapped):

```
# cd /var/adm
# mkcdsl pacct
# ls -l pacct
lrwxr-xr-x   1 root     adm     43 Jan  3 12:09  pacct ->
                  ../cluster/members/{memb}/adm/pacct
```

The `ln -s` command may also be used to create context-dependent symbolic links:

```
# ln -s "../cluster/members/{memb}/adm/pacct" ./pacct
```

The `cdslinvchk -verify` command may be used to verify that all expected CDSLs are present on a system. It reports its findings to the file */var/adm/cdsl_check_list*. Here is some sample output (wrapped to fit):

```
Expected CDSL: ./usr/var/X11/Xserver.conf ->
  ../cluster/members/{memb}/X11/Xserver.conf
An administrator or application has replaced this CDSL with:
-rw-r--r-- 1 root system 4545   Jan 3 12:41
                                    /usr/var/X11/Xserver.conf
```

This report indicates that there is one missing CDSL.

Sockets

A socket, whose official name is a *Unix domain socket*, is a special type of file used for communications between processes. A socket may be thought of as a communications end point, tied to a particular local system port, to which processes may attach. For example, on a BSD-style system, the socket */dev/printer* is used by processes to send messages to the program `lpd` (the line-printer spooling daemon), informing it that it has work to do.

Named pipes

Named pipes are pipes opened by applications for interprocess communication (they are "named" in the sense that applications refer to them by their pathname). They are a System V feature that has migrated to all versions of Unix. Named pipes often reside in the */dev* directory. They are also known as FIFOs (for "first-in, first-out").

Using ls to identify file types

The long directory listing (produced by the `ls -l` command) identifies the type of each file it lists via the initial character of the permissions string:

`-`	Plain file (hard link)
`d`	Directory
`l`	Symbolic link
`b`	Block special file
`c`	Character special file
`s`	Socket
`p`	Named pipe

For example, the following `ls -l` output includes each of the file types discussed above, in the same order:

```
-rw------- 2 chavez  chem     28 Mar 12 11:36  gold.dat
-rw------- 2 chavez  chem     28 Mar 12 11:36  hlink.dat
drwx------ 2 chavez  chem    512 Mar 12 11:36  old_data
lrwxrwxrwx 1 chavez  chem      8 Mar 12 11:37  zn.dat -> gold.dat
brw-r----- 1 root    system    0 Mar  2 15:02  /dev/sd0a
crw-r----- 1 root    system    0 Jun 12  1989  /dev/rsd0a
srw-rw-rw- 1 root    system    0 Mar 11 08:19  /dev/log
prw------- 1 root    system    0 Mar 11 08:32  /usr/lib/cron/FIFO
```

Note that the `-l` option also displays the target file for symbolic links (following the `->` symbol).

`ls` has other options to make identifying file types easy. On many systems, the `-F` option will append a special character to each filename, indicating its type:

```
-rw------- 2 chavez  chem     28 Mar 12 11:36  gold.dat
-rw------- 2 chavez  chem     28 Mar 12 11:36  hlink.dat
drwx------ 2 chavez  chem    512 Mar 12 11:36  old_data/
-rwxr-x--- 1 chavez  chem  23478 Feb 23 09:45  test_prog*
lrwxrwxrwx 1 chavez  chem      8 Mar 12 11:37  zn.dat@ -> gold.dat
srw-rw-rw- 1 root    system    0 Mar 11 08:19  /dev/log=
prw------- 1 root    system    0 Mar 11 08:32  /usr/lib/cron/FIFO|
```

Note than an asterisk indicates an executable file (program or script). Some versions of `ls` also support a `-o` option, which color-codes filenames in the output based on their file type.

You can use the `-i` option to `ls` to determine the equivalent file in the case of hard links. Using `-i` tells `ls` to display the inode number associated with each filename. Here is an example:

```
$ ls -i /dev/rmt0 /dev/rmt/*
290 /dev/rmt0 293 /dev/rmt/c0d6ln
292 /dev/rmt/c0d6h291 /dev/rmt/c0d6m
295 /dev/rmt/c0d6hn294 /dev/rmt/c0d6mn
290 /dev/rmt/c0d6l
```

From this display, we can determine that the special files */dev/rmt0* (the default tape drive for many commands, including `tar`) and */dev/rmt/c0d6l* are equivalent, because they both reference inode number 290.

`ls` can't distinguish between text and binary files (both are "regular" files). You can use the `file` command to do so. Here is an example:

```
# file *
appoint: ... executable not stripped
bin: directory
clean: symbolic link to bin/clean
fort.1: empty
gold.dat: ascii text
intro.ms:   [nt]roff, tbl, or eqn input text
run_me.sh: commands text
xray.c: ascii text
```

The file *appoint* is an executable image; the additional information provided for such files differs from system to system. Note that `file` tries to figure out what the contents of ASCII files are, with varying success.

Processes

In simple terms, a *process* is a single executable program that is running in its own address space.* It is distinct from a job or a command, which, on Unix systems, may be composed of many processes working together to perform a specific task. Simple commands like `ls` are executed as a single process. A compound command containing pipes will execute one process per pipe segment. For Unix systems, managing CPU resources must be done in large part by controlling processes, because the resource allocation and batch execution facilities available with other multitasking operating systems are underdeveloped or missing.

Unix processes come in several types. We'll look at the most common here.

Interactive Processes

Interactive processes are initiated from and controlled by a terminal session. Interactive processes may run either in the *foreground* or the *background*. Foreground processes remain attached to the terminal; the foreground process is the one with which

* I am not distinguishing between processes and threads at this point.

the terminal communicates directly. For example, typing a Unix command and waiting for its output means running a foreground process.

While a foreground process is running, it alone can receive direct input from the terminal. For example, if you run the `diff` command on two very large files, you will be unable to run another command until it finishes (or you kill it with CTRL-C).

Job control allows a process to be moved between the foreground and the background at will. For example, when a process is moved from the foreground to the background, the process is temporarily stopped, and terminal control returns to its parent process (usually a shell). The background job may be resumed and continue executing unattached to the terminal session that launched it. Alternatively, it may eventually be brought to the foreground, and once again become the terminal's current process. Processes may also be started initially as background processes.

Table 2-6 reviews the ways to control foreground and background processes provided by most current shells.

Table 2-6. Controlling processes

Form	Meaning and examples
&	Run command in background. `$ long_cmd &`

^Z	Stop foreground process.
	`$ long_cmd`
	`^Z  Stopped`
	`$`

jobs	List background processes.
	`$ jobs`
	`[1] - Stopped emacs`
	`[2] - big_job &`
	`[3] + Stopped long_cmd`

%n	Refers to background job number *n*. `$ kill %2`
fg	Bring background process to foreground. `$ fg %1`
%?str	Refers to the background job command containing the specified characters. `$ fg %?em`

bg	Restart stopped background process.
	`$ long_cmd`
	`^Z Stopped`
	`$ bg`
	`[3] long_cmd &`

~^Z	Suspend `rlogin` session.
	`bridget-27 $ ~^Z`
	`Stopped`
	`henry-85 $`

Table 2-6. Controlling processes (continued)

Form	Meaning and examples

~~^Z	Suspend second-level `rlogin` session. Useful for nested `rlogin`s; each additional tilde says to pop back to the next highest level of `rlogin`. Thus, one tilde pops all the way back to the lowest level job (the job on the local system), two tildes pops back to the first `rlogin` session, and so on.
	`bridget-28 $ ~~^Z`
	`Stopped`
	`peter-46 $`

Batch Processes

Batch processes are not associated with any terminal. Rather, they are submitted to a queue, from which jobs are executed sequentially. Unix offers a very primitive `batch` command, but vendors whose customers require queuing have generally implemented something more substantial. Some of the best known are the Network Queuing System (NQS), developed by NASA and used on many high-performance computers including Crays, as well as several network-based process-scheduling systems from various vendors. These facilities usually support heterogeneous as well as homogeneous networks, and they attempt to distribute the aggregate CPU load evenly among the workstations in the network, a process known as *load balancing* or *load leveling*.

Daemons

Daemons are server processes, often initiated at boot time, that run continuously while the system is up, waiting in the background until a process requires their service.* For example, network daemons are idle until a process requests network access.

Table 2-7 provides a brief overview of the most important Unix daemons.

Table 2-7. Important Unix daemons

Facility	Description	Daemon Names
`init`	First created process	`init`
`syslog`	System status/error message logging	`syslogd`
email	Mail message transport	`sendmail`
printing	Print spooler	`lpd`, `lpsched`, `qdaemon`, `rlpdaemon`

* Daemon is an ancient Greek word meaning "divinity" or "spirit" (but keep the character of the Greek gods in mind). The OED defines it as a "tutelary deity": the guardian of a particular person, place or thing. More recently, the poet Yeats wrote at length about daemons, defining them as that which we continually struggle against yet paradoxically need in order to survive, simultaneously the source of our pain and of our strength, even in some sense, the very essence of our being. For Yeats, the daemon is "of all things not impossible the most difficult."

Table 2-7. Important Unix daemons (continued)

Facility	Description	Daemon Names
cron	Periodic process execution	`crond`
tty	Terminal support.	`getty` (and similar)
sync	Disk buffer flushing	`update`, `syncd`, `syncher`, `fsflush`, `bdflush`, `kupdated`
paging and swapping	Daemons to support virtual memory management	`pagedaemon`, `vhand`, `kpiod`, `pageout`, `swapper`, `kswapd`, `kreclaimd`
`inetd`	Master TCP/IP daemon, responsible for starting many others on demand: `telnetd`, `ftpd`, `rshd`, `imapd`, `pop3d`, `fingerd`, `rwhod` (see */etc/inetd.conf* for a full list)	`inetd`
name resolution	DNS server process	`named`
routing	Routing daemon	`routed`, `gated`
DHCP	Dynamic network client configuration	`dhcpd`, `dhcpsd`
RPC	Remote procedure call facility network port-to-service mapper	`portmap`, `rpcbind`
NFS	Network File System: native Unix network file sharing	`nfsd`, `rpc.mountd`, `rpc.nfsd`, `rpc.statd`, `rpc.lockd`, `nfsiod`
Samba	File/print sharing with Windows systems	`smbd`, `nmbd`
WWW	HTTP server	`httpd`
network time	Network time synchronization	`timed`, `ntpd`

Process Attributes

Unix processes have many associated attributes. Some of the most important are:

Process ID (PID)
: A unique identifying number used to refer to the process.

Parent process ID (PPID)
: The PID of the process's *parent* process (the process that created it).

Nice number
: The process's scheduling priority, which is a number indicating its importance relative to other processes. This needs to be distinguished from its actual execution priority, which is dynamically changed based on both the process's nice number and its recent CPU usage. See "Managing CPU Resources" in Chapter 15 for a detailed discussion of nice numbers and their effect on execution priority.

TTY
: The terminal (or pseudo-terminal) device associated with the process.

Real and effective user ID (RUID, EUID)
: A process's real UID is the UID of the user who started it. Its effective UID is the UID that is used to determine the process's access to system resources (such as

files and devices). Usually the real and effective UIDs are the same, and the process accordingly has the same access rights as the user who launched it. However, when the setuid access mode is set on an executable image, then the EUIDs of processes executing it are set to the UID of the file's user owner, and they are accorded corresponding access rights.

Real and effective group ID (RGID, EGID)
: A process's real GID is the user's primary or current group. Its effective GID, used to determine the process's access rights, is the same as the real GID except when the setgid access mode is set on an executable image. The EGIDs of processes executing such files are set to the GID of the file's group owner, and they are given corresponding access to system resources.

The life cycle of a process

A new process is created in the following manner. An existing process makes an exact copy of itself, a procedure known as *forking*. The new process, called the *child process*, has the same environment as its *parent process*, although it is assigned a different process ID. Then, this image in the child process's address space is overwritten by the one the child will run; this is done via the `exec` system call. Hence, the often-used phrase *fork-and-exec*. The new program (or command) completely replaces the one duplicated from the parent. However, the environment of the parent still remains, including the values of environment variables; the assignments of standard input, standard output, and standard error; and its execution priority.

Let's make this picture a bit more concrete. What happens when a user runs a command like `grep`? First, the user's shell process forks, creating a new shell process to run the command. Then, the new shell process execs `grep`, which overlays the shell's executable image in memory with `grep`'s, which begins executing. When the `grep` command finishes, the process dies.

This is the way that all Unix processes are created. The ultimate ancestor for every process on a Unix system is the process with PID 1, `init`, created during the boot process (see Chapter 4). `init` creates many other processes (all by fork-and-exec). Among them are usually one or more executing the `getty` program. The `getty`s are each assigned to a different serial line; they display the login prompt and wait for someone to respond to it. When someone does, the `getty` process execs the `login` program, which validates user logins, among other activities.*

Once the username and password are verified,† `login` execs the user's shell. Forking is not always required to run a new program, and `login` does not fork in this case. After

* The process is similar for an X terminal window. In the latter case, the `xterm` or other process is created by the window manager in use, which was itself started by a series of other X-related processes, ultimately deriving from a command issued from the login shell (e.g., `startx`) or as part of the login process itself.

† If the login attempt fails, `login` exits, sending a signal to its parent process, `init`, indicating it should create a new `getty` process for the terminal.

logging in, the user's shell is the same process as the getty that was watching the unused serial line. That process changed programs twice by execing a new executable, and it will go on to create new processes to execute the commands that the user types. Figure 2-3 illustrates Unix process creation in the context of initial user login.

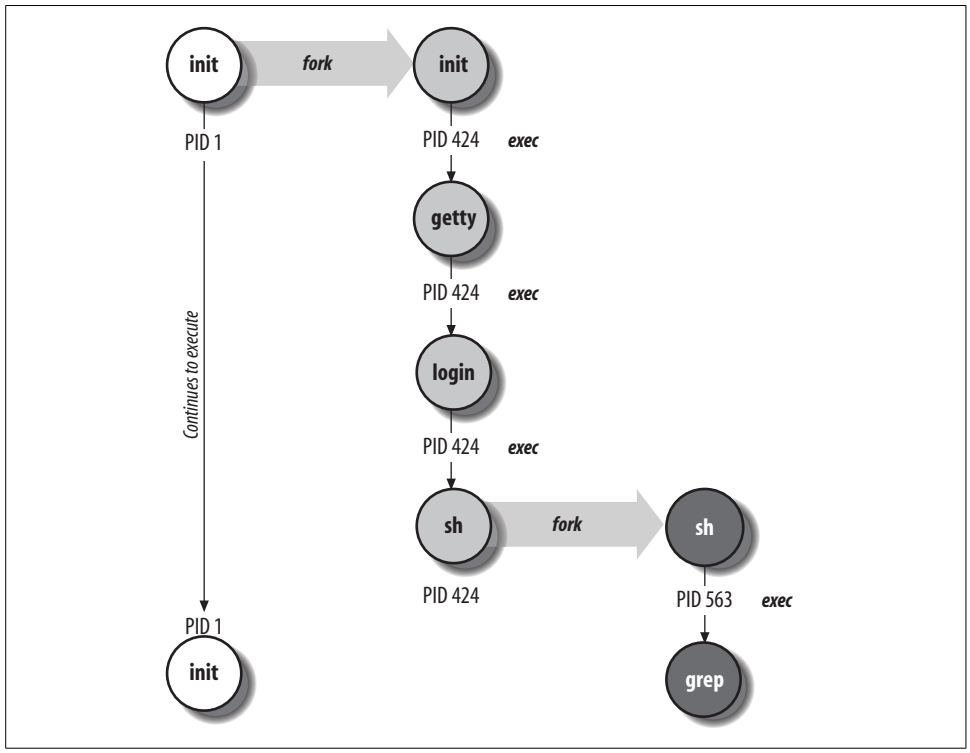


Figure 2-3. Unix process creation: fork and exec

When any process exits, it sends a signal to inform its parent process that is has completed. So, when a user logs out, her login shell sends a signal to its parent, `init`, as it dies, letting `init` know that it's time to create a new `getty` process for the terminal. `init` forks again and starts the `getty`, and the whole cycle repeats itself again and again as different users use that terminal.

Setuid and setgid file access and process execution

The purpose of the setuid and setgid access modes is to allow ordinary users to perform tasks requiring privileges and access rights that are ordinarily denied to them. For example, on many systems the `write` command is owned by the *tty* group, which also owns all of the terminal and pseudo-terminal device files. The `write` command has setgid access, allowing any user to use it to write a message to another user's terminal or window (to which they do not normally have any access). When users execute `write`, their effective GID is set to that of the group owner of the executable file (often */usr/bin/write*) for the duration of the command.

Setuid and/or setgid access are also used by the printing subsystem, by programs like mailers, and by some other system facilities. However, setuid programs are also notorious security risks. In practice, setuid almost always means setuid to *root,* and the danger is that somehow, through program stupidity or their own cleverness or both, users will figure out a way to perform additional, unauthorized functions while the setuid command is running or to retain their inherited *root* status after the command ends. In general, setuid access should be avoided since it involves greater security risks than setgid, and almost any function can be performed by using the latter in conjunction with carefully designed groups. See Chapter 7 for a more detailed discussion of the security issues involved with setuid and setgid programs. Keep in mind, though, that while setgid programs are safer than setuid ones, they are not risk-free themselves.

The relationship between commands and files

The Unix operating system does not distinguish between commands and files in the ways that some systems do. Aside from a few commands that are built into each Unix shell, Unix commands are executable files stored in one of several standard locations within the filesystem. Access to commands is exactly equivalent to access to these files. By default, there is no other privilege mechanism. Even I/O is handled via *special files*, stored in the directory */dev*, which function as interfaces to the device drivers. All I/O operations look just like ordinary file operations from the user's point of view.

Unix shells use *search paths* to locate the executable's images for commands that users enter. In its simplest form, a search path is simply an ordered list of directories in which to look for command executables, and it is typically set in an initialization file *($HOME/.profile* or *$HOME/.login*). A faulty (incomplete) search path is the most common cause for "Command not found" error messages.

Search paths are stored in the *PATH* environment variable. Here is a typical *PATH*:

```
$ echo $PATH
/bin:/usr/ucb:/usr/bin:/usr/local/bin:.:$HOME/bin
```

The various directories in the *PATH* are separated by colons. The search path is used whenever a command name is entered without an explicit directory location. As an example, consider the following command:

```
$ od data.raw
```

The `od` command is used to display a raw dump of a file. To locate this command, the operating system first looks for a file named `od` in */bin*. If such a file exists, it is executed. If there is no `od` file in the */bin* directory, */usr/ucb* is checked next, followed by */usr/bin* (where `od` is in fact usually located). If it were necessary, the search would continue in */usr/local/bin*, the current directory, and finally the *bin* subdirectory of the user's home directory.

The order of the directories in the search path is important when more than one version of a command exists. Such effects come into play most frequently when both

the BSD and the System V versions of commands are available on a system. In this case, you should put the directory holding the versions you want to use first in your search path. For example, if you want to use the BSD versions of commands such as `ls` and `ln` on a System V–based system, then put */usr/ucb* ahead of */usr/bin* in your search path. Similarly, if you want to use the System V–compatible commands available on some systems, put */usr/5bin* ahead of */usr/bin* and */usr/ucb* in your search path. These same considerations will obviously apply to users' search paths that you define for them in their initialization files (see "Initialization Files and Boot Scripts" in Chapter 4).

Most of the Unix administrative utilities are located in the directories */sbin* and */usr/sbin*. However, the locations of administrative commands can vary widely between Unix versions. These directories typically aren't in the search path unless you put them there explicitly. When executing administrative commands, you can either add these directories to your search path or provide the full pathname for the command, as in the example below:

```
# /usr/sbin/ping hamlet
```

I'm going to assume in my examples that the administrative directories have been added to the search path. Thus, I won't be including the full pathname for any of the commands I'll be discussing.

The Unix Way of System Administration

System administrators are stereotypically arrogant, single-minded, and opinionated. For Unix system administrators, the stereotype was born in the days when Unix was this bizarre operating system that ran on only a few systems, and the local Unix guru was some guy who generally kept to himself, locked away with his system—or so the story goes.

The skepticism I'm exhibiting with this view of Unix system managers does not mean that there is no truth in it at all. Like most caricatures, this one has roots in reality. For example, it is all too easy to find people who will tell you that there is one right editor to use, one right shell for writing scripts, one right way to do anything you care to name. Discussing the advantages and liabilities of alternative approaches to problems can be both useful and entertaining, but only within reason.

Since you're reading this introductory chapter, I'm assuming that you are only beginning your exploration of Unix administration. I certainly want to encourage you to consider for yourself all the tasks and issues you will face as you proceed and to provide help when I can. You'll quickly form your own opinions and define what system administration is for you. Doing so is a process, which can continue for as long and range as widely as you want it to. However, if you get to a point where fanaticism replaces thinking, you've gone too far.

Devices

One of the strengths of Unix is that users don't need to worry about the specific characteristics of devices and device I/O very often. They don't need to know, for example, what disk drive a file they want to access physically sits on. And the Unix special file mechanism allows many device I/O operations to look just like file I/O. As we've noted, the administrator doesn't have these same luxuries, at least not all the time. This section discusses Unix device handling and then surveys the special files used to access devices.

Device files are characterized by their *major* and *minor numbers*, which allow the kernel to determine which device driver to use to access the device (via the major number), as well as its specific method of access (via the minor number).

Major and minor numbers appear in place of the file size in long directory listings. For example, consider these device files related to the mouse from a Linux system:

```
$ cd /dev; ls -l *mouse
crw-rw-r--    1 root     root      10,  10 Jan 19 03:36 adbmouse
crw-rw-r--    1 root     root      10,   4 Jan 19 03:35 amigamouse
crw-rw-r--    1 root     root      10,   5 Jan 19 03:35 atarimouse
crw-rw-r--    1 root     root      10,   8 Jan 19 03:35 smouse
crw-rw-r--    1 root     root      10,   6 Jan 19 03:35 sunmouse
crw-rw-r--    1 root     root      13,  32 Jan 19 03:36 usbmouse
```

The major number for all but the last special file is 10; only the minor number differs for these devices. Thus, all of these mouse device variations are handled by the same device driver, and the minor number indicates the variation within that general family. The final item, corresponding to a USB mouse, has a different major number, indicating that a different device driver is used.

Device files are created with the `mknod` command, and it takes the desired device name and major and minor numbers as its arguments. Many systems provide a script named *MAKEDEV* (located in */dev*), which is an easy-to-use interface to `mknod`.

An In-Depth Device Example: Disks

We'll use disk drives as an example in this overview discussion of Unix devices.* As we've noted before, Unix organizes all user-accessible files into a single hierarchical directory structure. The files and directories it contains may be spread across several different disk drives.

On most Unix systems, disks are divided into one or more fixed-size *partitions*: physical subsets of the disk drive that are separately accessed by the operating system.

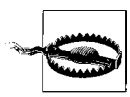
* This discussion will describe traditional ways of handling disks and filesystems. Unix versions that require or offer a logical volume manager do things quite differently at the lowest level, but this overview is still conceptually true for those systems (for "disk partition," read "logical volume"). See Chapter 10 for details.

There may be several partitions or just one on each physical disk. The disk partition containing the root filesystem is called the *root partition* and sometimes the *root disk*, although it obviously needn't comprise the entire disk drive. The disk containing the root partition is generally called the *system disk*.

The root filesystem is the first one *mounted*, early in the Unix boot process, and the remaining ones are mounted afterwards. On many operating systems, mounting a disk refers to the process of making the device's contents available. For Unix, it means something more. Like the overall Unix filesystem, the files and directories physically located on each disk partition are arranged in a tree structure.* An integral part of the process of mounting a disk partition involves grafting its local directory structure into the overall Unix directory tree. Once this is done, the files physically residing on that device may be accessed via the usual Unix pathname syntax; Unix takes care of mapping pathnames to the correct physical device and data blocks.

For administrators, however, there are a few times when the disk partition must be accessed directly. The actual mount operation is the most common. Remember that disk partitions may be accessed in two modes, block mode and raw (or character) mode, and different special files are used from each mode. Character access mode does unbuffered I/O, generally making a data transfer to or from the device with every read or write system call. Block devices do buffered I/O on a block basis, collecting data in a buffer until the operating system can transfer an entire block of data at one time.

For example, the disk partition containing the root filesystem traditionally corresponded to the special files */dev/disk0a* and */dev/rdisk0a*, specifying the first partition on the first disk (disk 0, partition a), accessed in block and raw mode respectively,† with the *r* designating raw device access.



Most disk partition–related commands require a specific type of special file and won't accept the other kind.

* For this reason, each separate disk partition may also be referred to as a filesystem. Thus, "filesystem" is used to refer both to the overall system directory tree (as in "the Unix filesystem"), comprising every user-accessible disk partition on the system, and to the files and directories on individual disk partitions (as in "build a filesystem on the disk partition" or "mounting the user filesystems"). Whether the overall Unix directory tree or an individual disk partition is meant will be clear from the context. On a related note, the terms partition and filesystem are often used synonymously. Thus, while technically only filesystems can be mounted, common usage often refers to "mounting a disk" or "mounting a partition."

† The names given to the two types of special files are overdetermined. For example, the special file */dev/disk0a* is referred to as a *block special file*, and */dev/rdisk0a* is called a *character special file*. However, block special files are also sometimes called *block devices*, and character special files may be referred to as *character devices* or *raw devices*.

Note that most Linux versions and newer versions of BSD do not distinguish between the two types of special files for IDE disks and provide only one special file per disk partition.

As an example of the use of special files to access disk partitions, consider the `mount` commands below:

```
# mount /dev/disk0a /
# mount /dev/disk1e /home
```

Naturally, the command to mount a disk partition needs to specify the physical disk partition to be mounted (`mount`'s first argument) and the location to place it in the filesystem, its *mount point* (the second argument).* Thus, the first command makes the files in the first partition on drive 0 available, placing them at the root of the Unix filesystem. The second command accesses a partition on drive 1, placing it at */home* in the overall directory tree. Thus, regular files in the top-level directory on this second disk partition will appear in */home*, and top-level directories on the disk partition become subdirectories of */home*. The `mount` command is discussed in greater detail in Chapter 10.

Fixed-disk special files

Currently used special file names for disk partitions are highly implementation-dependent. However, a common logic underlies all of the various naming schemes. Disk special files can encode the type of disk, the disk controller, the disk location on its controller, and the disk partition within the physical disk (as well as the access mode) within the special file name.

Let's take the Tru64 special files for disks as an example; these special files have names of the following form, where *n* is the disk number (beginning at 0), and *x* is a letter from a to h designating the partition on the physical disk:

*/dev/disk/dsk*nx
: Block device

*/dev/rdisk/dsk*nx
: Character (raw) device

The partitions have conventional uses, and not all partitions are used on every disk (see Chapter 10 for more details). Traditionally, the a partition on the root disk contains the root filesystem. b partitions are conventionally used as swap partitions. On the root disk, other partitions might be used for various system directories: for example, e for */usr*, h for */var*, d for other filesystems, and so on.

* In fact, on most Unix systems, `mount` is smarter than this. If you give it a single argument—either the physical disk partition or the mount point—it will look up the other argument in a table. But you can always supply both arguments, which means that you can rearrange your filesystem at will. (Why you would want to is a different question.)

The c partition often refers to the entire disk as a whole: every bit of space on the disk, including areas that should be accessed only by the kernel (such as the partition table at the beginning of the drive). For this reason, using the c partition for a filesystem was not allowed under older versions of Unix. More recent versions generally do not have this restriction.

System V-based systems use a similar naming philosophy, although the actual names differ. Special filenames for disk partitions are often of the form */dev/dsk/c*k*t*m*d*p*s*n*, where *k* is the controller number, *m* is the drive number on that controller (often the SCSI target ID), and *n* is the partition (section) number on that drive (all numbers start at 0). *p* refers to the logical unit number (LUN) for SCSI devices and is thus usually 0. HP-UX uses this form but typically omits the *s* component.

In this scheme, character and block special files have the same names, but they are stored in two different subdirectories of */dev*: */dev/dsk* and */dev/rdsk*, respectively. Thus, the special file */dev/dsk/c1t4d0s2* is the block special file for the third partition on the disk with SCSI ID 4 on controller 1 (the second controller). The corresponding character device is */dev/rdsk/c1t4d0s2*.

Names in this format, known as *controller-drive-section identifiers*, are specified for all disk and tape devices under the System V.4 standard. Actual System V–based implementations start with this framework and may vary it somewhat according to the devices actually supported. Sometimes, they also provide links to more mnemonically or intuitively-named special files. For example, on some (mostly older) Solaris systems, */dev/sd0a* might be linked to */dev/dsk/c0t3d0s0*, allowing the conventional SunOS name to be used for the 0 partition on the disk with SCSI ID 3 on the first controller.*

Table 2-8 illustrates the similarities among disk special file names. The special files in the table all refer to a partition on the second SCSI disk drive on the first controller, using SCSI ID 4.

Table 2-8. Interpreting disk special file names

	FreeBSD	HP-UX	Linux	Solaris	Tru64[a]
Special file	*/dev/rda1d*	*/dev/rdsk/c0t4d0*	*/dev/sdb1*	*/dev/rdsk/c0t4d0s3*	*/dev/rdisk/dsk1c*
Raw access	*/dev/**r**da1d*	*/dev/**r**dsk/c0t4d0*	*/dev/sdb1*	*/dev/**r**dsk/c0t4d0s3*	*/dev/**r**disk/dsk1c*
Device = Disk	*/dev/r**d**a1d*	*/dev/r**dsk**/c0t4d0*	*/dev/s**d**b1*	*/dev/r**dsk**/c0t4d0s3*	*/dev/r**disk**/dsk1c*
Type = SCSI	*/dev/r**da**1d*		*/dev/s**d**b1*		
Controller #		*/dev/rdsk/**c0**t4d0*		*/dev/rdsk/**c0**t4d0s3*	
SCSI ID		*/dev/rdsk/c0**t4**d0*		*/dev/rdsk/c0**t4**d0s3*	

* Even this isn't the full truth about Solaris special files. The files in */dev* are usually links to the real device files in the */devices* directory subtree.

Table 2-8. Interpreting disk special file names (continued)

	FreeBSD	HP-UX	Linux	Solaris	Tru64[a]
Device #	*/dev/rda**1**d*		*/dev/sd**b**1*		*/dev/rdisk/dsk**1**c*
Disk Partition	*/dev/rda1**d***	assumed	*/dev/sdb**1***	*/dev/rdsk/c0t4d0**s3***	*/dev/rdisk/dsk1**c***

[a] Older Tru64 systems use the now-obsolete device names of the form */dev/rz**, */dev/ra**, and */dev/re**.

In yet another twist, systems that use logical volume managers (including AIX by default) allow the system administrator to specify names for the special files for logical volumes—virtual disk partitions—when they are created. These special files often have names of the form */dev/*name, where *name* is chosen when the filesystem is created. On such systems, it is logical volumes rather than physical partitions that hold filesystems. We'll leave the rest of the gory details about these topics until Chapter 10.

Special Files for Other Devices

Other device types have special files named differently, but they follow the same basic conventions. Some of the most common are summarized in Table 2-9 (they will be discussed in more detail as appropriate in later chapters). In some cases, only the more commonly used form (block versus character) of the file is listed. For example, tape drives are seldom, if ever, accessed via the block device, and on many systems, the block special files do not even exist.

Table 2-9. Common Unix special file names

Device/use	Special file forms	Example
Floppy disk	/dev/[r]fd*n** /dev/floppy	*/dev/fd0*
Tape devices[a]	/dev/rmt*n*	*/dev/rmt1*
	/dev/rmt/*n*	*/dev/rmt/0*
nonrewinding	/dev/nrmt*n*	*/dev/nrmt0*
SCSI	/dev/rst*n*	*/dev/rst0*
default tape drive	/dev/tape	
CD-ROM devices	/dev/cd*n* /dev/cdrom	*/dev/cd0*
Serial lines	/dev/tty*n* /dev/term/*n*	*/dev/tty1* */dev/tty01* */dev/term/01*
Slave virtual terminal (windows, network sessions, etc.)	/dev/tty[p-s]*n* /dev/pts/*n*	*/dev/ttyp1* */dev/pts/2*
Master/control virtual terminal devices	/dev/pty[p-s]*n*	*/dev/ptyp3*
Console device	/dev/console	
some System V	/dev/syscon	
AIX	/dev/lft0	

Table 2-9. Common Unix special file names (continued)

Device/use	Special file forms	Example
Process controlling TTY (used to ensure I/O comes from/goes to terminal, regardless of any I/O redirection)	/dev/tty	
Memory maps:		
physical	/dev/mem	
kernel virtual	/dev/kmem	
Mouse interface	/dev/mouse	
Null devices: all output is discarded; reads return nothing (0 characters, 0 bytes) or a zero-filled buffer, respectively.	/dev/null /dev/zero	

[a] Tape devices often have suffixes that specify the tape density.

Commands for listing the devices on a system

Most Unix versions provide commands that make it easy to quickly determine what devices are present on the system, as well as their current status. Table 2-10 lists the commands for the systems we are considering.

Table 2-10. Device listing and information commands

Unix Version	Command(s)	Description
AIX	`lscfg`	List all devices.
	`lscfg -v -l device`	Device configuration detail.
	`lsdev -C -s scsi`	List all SCSI IDs.
	`lsattr -E -H -l device`	Display device attributes.
FreeBSD	`pciconf -l -v`	List PCI devices
	`camcontrol devlist`	List SCSI devices.
HP-UX	`ioscan -f -n`	Detailed device listing.
	`ioscan -f -n -C disk`	Limit to device class.
Linux	`lsdev`	List major devices.
	`scsiinfo -l`	List SCSI devices.
	`lspci`	List PCI devices.
Solaris[a]	`dmesg`[b]	Boot messages identify all devices.
	`getdev`	List devices.
	`getdev type=disk`	Limit to device class.
	`devattr -v device`	Device detail.
Tru64	`dsfmgr -s`	List devices.

[a] Unfortunately, the *getdev* and *devattr* commands are often of limited use.
[b] *dmesg* is also available under FreeBSD, HP-UX, and Linux.

The AIX Object Data Manager

Under AIX, information about the devices on the system and other system configuration is stored in a binary database. The management apparatus for this database is known as the Object Data Manager (ODM), although "ODM" is also used colloquially to refer to the database itself, as well. Information is stored in the ODM as *objects*: items of various predefined types, with a collection of attributes and their associated sets or ranges of legal values.

Here is a textual representation of a sample entry for a disk drive:

```
name = "hdisk0"
status = 1
chgstatus = 2
ddins = "scdisk"
location = "00-00-0S-0,0"
parent = "scsi0"
connwhere = "0,0"
PdDvLn = "disk/scsi/1000mb"
```

This entry illustrates the general form for a device; most devices use the same fields, although their meaning varies somewhat depending on the device type. This entry describes a 1 GB SCSI disk drive.

The preceding entry came from the current devices database, stored in */etc/objrepos/CuDv*. The attributes for this object (as well as those for the other objects on the system) are stored in a separate, current attributes database (found in */etc/objrepos/CuAt*). This database may have several entries for any given object, one for each defined attribute for that class of object for which a nondefault value is set. For example, here are two of the attributes for the logical volume *hd6* (one of the disk partitions on *hdisk0*):

```
name = "hd6"
attribute = "type"
value = "paging"
type = "R"
generic = "DU"
rep = "s"
nls_index = 639
name = "hd6"
attribute = "size"
value = "16"
type = "R"
generic = "DU"
rep = "r"
nls_index = 647
```

The first entry indicates that this is a paging space, and the second indicates that its size is 16 logical partitions (64 MB, assuming the default partition size).

SMIT and the AIX commands it runs retrieve information from the ODM, as well as adding and modifying entries as necessary.

The Unix Filesystem Layout

Now that we've considered the Unix approach to major system components, it's time to acquaint you with the structure of the Unix filesystem. This brief tour will begin with the root directory and its most important subdirectories.

The basic layout of traditional Unix filesystems is illustrated in Figure 2-4, which shows an idealized directory structure (actually a superset of the items found on any one system). Note that in practice, there are lots of variations with respect to this paradigm.

You'll find small deviations from this on most Unix systems you encounter, but the basic structure will be quite similar. We'll consider each of the major directories in turn.

The Root Directory

This is the base of the filesystem's tree structure; all other files and directories, regardless of their physical disk locations, are logically contained underneath the root directory (described in detail in Chapter 10).

There are a variety of important first-level directories under the / directory:

/bin
: The traditional location for executable (binary) files for the various Unix user commands and utilities. On many current systems, some files within */bin* are merely symbolic links to files in */usr/bin*, and */bin* is sometimes a link to */usr/bin*. Other directories that hold Unix commands are */usr/bin* and */usr/ucb*.

/dev
: The device directory, containing special files as described previously. The */dev* directory is divided into subdirectories in most System V–based versions of Unix, with each subdirectory holding special files of a given type. Subdirectory names indicate the type of devices it contains: *dsk* and *rdsk* for disks accessed in block and raw mode, *mt* and *rmt* for tape drives, *term* for terminals (serial lines), *pts* and *ptc* for pseudo-terminals, and so on.

  Solaris introduces a new device directory tree, beginning at */devices*, and many files under */dev* are links to files in subdirectories of */devices*.

/etc and /sbin
: System configuration files and executables. These directories contain many administrative files and configuration files. Among the most important files are the System V–style boot script subdirectories, named *rc*n*.d* and *init.d*, which are located under one of these two locations on systems using this style of booting.

  /etc also traditionally contained the executable binaries for most administrative commands. In recent Unix versions, these files have moved to */sbin* and */usr/sbin*. Conventionally, the former is used for files required to boot the system, and the latter contains all other administrative commands.

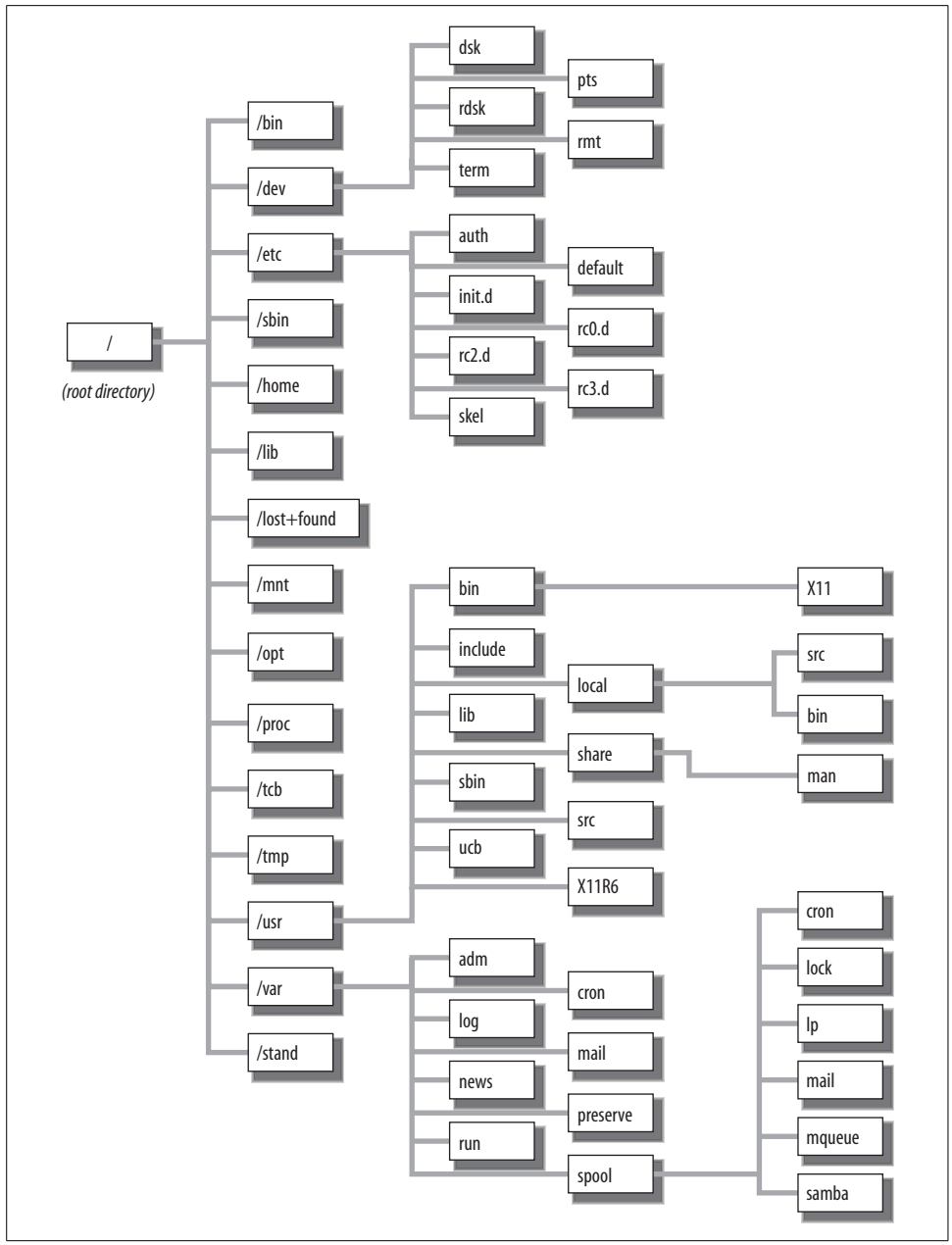


Figure 2-4. Generic Unix directory structure

On many systems, */etc* also contains a subdirectory *default*, which holds files containing default parameter values for various commands.

On Linux systems, the *sysconfig* subdirectory holds network configuration and other package-specific, boot-related configuration files.

Under AIX, */etc* contains two additional directories of note: */etc/objrepos* stores the device configuration databases, and */etc/security* stores most security-related configuration files.

/home
: This directory is a conventional location for users' home directories. For example, user *chavez*'s home directory is often */home/chavez*. The name is completely arbitrary, however, and is often changed by the local site. It may also be a separate filesystem.

/lib
: Location of shared libraries required for booting the system (i.e., before */usr* is mounted).

/lost+found
: Lost files directory. Disk errors or incorrect system shutdown may cause files to become *lost*: lost files refer to disk locations that are marked as in use in the data structures on the disk, but that are not listed in any directory (i.e., an inode with a link count greater than zero that isn't listed in any directory). When the system is booting, it runs a program called `fsck` that, among other things, finds these files.

  There is usually a *lost+found* directory on every disk partition; */lost+found* is the one on the root disk. However, some Unix systems do not create the directory until it is needed.

/mnt
: Temporary mount directory: an empty directory conventionally designed for temporarily mounting filesystems.

/opt
: Directory tree into which optional software is often installed. On some systems, optional software products are installed instead under */var/opt*. On AIX systems, this function is provided by the directory */usr/lpp*.

/proc
: Process directory, designed to enable processes to be manipulated using Unix file access system calls. Files in this directory correspond to active processes (entries in the kernel process table). On Linux systems, there are also additional files containing various information about the system configuration: interrupt usage, I/O port use, DMA channel allocation, CPU type, and the like. The HP-UX operating system does not use */proc*.

/stand
: Boot-related files, including the kernel executable. Solaris uses */kernel*, and Linux systems use */boot* for the same purpose. FreeBSD systems use */stand* for installation and system configuration–related programs and use */boot* for kernels and related files used for booting.

/tcb

Directory tree for security-related database files on some systems offering enhanced security features, including HP-UX and Tru64 (the name stands for "trusted computing base"). Configuration files related to the TCB are also stored under */etc/auth*. */usr/tcb* may also be used for this purpose.

/tmp

Temporary directory, available to all users as a scratch directory. The system administrator should see that all the files in this directory are deleted occasionally. Normally, one of the Unix startup scripts will clear */tmp*.

/usr

This directory contains subdirectories for locally generated programs, executables for user and administrative commands, shared libraries, and other parts of the Unix operating system. The most important subdirectories of */usr* are discussed in more detail in the next section. */usr* also sometimes contains application programs.

/var

Spooling and other volatile directories (*varying* data). Important subdirectories are described below.

The /usr Directory

The directory */usr* contains a number of important subdirectories:

/usr/bin

Command binary files and shell scripts. This directory contains public executable programs that are part of the Unix system. Many executables for the X Window System are stored in */usr/bin/X11* or */usr/X11R6/bin*.

/usr/include

Include files. This directory contains C-language header files that define the C programmer's interface to standard system features and program libraries. For example, it contains the file *stdio.h*, which defines the user's interface to the C standard I/O library. The directory */usr/include/sys* contains operating system include files.

/usr/lib

Library directory, for public library files. Among other things, this directory contains the standard C libraries for mathematics and I/O. Library files generally have names of the form *lib*x*.a* or *lib*x*.so*, where *x* is one or more characters related to the library's contents; the extensions specify a regular (statically linked) and shared library, respectively.

/usr/local

Local files. By convention, the directory */usr/local/bin* holds executable programs that were developed locally or retrieved from the Internet and any sources

other than the operating-system vendor. There may be other subdirectories here to hold related files: *man* (manual pages), *lib* (libraries), *src* (source code), *doc* (documentation), and so on.

/usr/sbin
: Administrative commands (except ones required for booting, which are in */sbin*).

/usr/share
: Shared data. On some recent systems, certain CPU architecture-independent static data files (such as the online manual pages, font directories, the dictionary files for `spell`, and the like) are stored in subdirectories under */usr/share*. The name *share* reflects the idea that such files could be shared among a group of networked systems, eliminating the need for separate copies on every system.

/usr/share/man
: One location for the manual pages directory tree. This directory contains the online version of the Unix reference manuals. It is divided into subdirectories for the various sections of the manual.

  Traditionally, the subdirectory structure contains several *man*n subdirectories holding the raw source for the manual pages in that section and corresponding *cat*n subdirectories storing the formatted versions. On many current systems, however, the latter are eliminated, and manual pages are formatted as needed. In many cases, the source files are stored in compressed form to save even more space.

  The significance of the manual sections is described in the Table 2-11.

Table 2-11. Manual-page sections

Contents	BSD style	System V style
User commands	1	1
System calls	2	2
Functions and library routines	3	3
Special files and hardware	4	7
Configuration files and file formats	5	4
Games and demos	6	6 or 1
Miscellaneous: character sets, filesystem types, data type definitions, etc.	7	5
System administration commands	8	1m
Maintenance commands	8	8
Device drivers	4	7 or 9

Among the systems we are considering, the BSD-style organization is used by FreeBSD, Linux, and Tru64, and the System V–style organization is more or less followed by AIX, HP-UX, and Solaris.

/usr/src
: Source code for locally built software packages (FreeBSD and Linux). FreeBSD also uses the */usr/ports* directory tree for retrieving and building additional software packages.

/usr/ucb
: A directory that contains standard Unix commands originally developed under BSD. Recent System V–based systems also provide BSD versions of commands so that users may use the form that they prefer. Some BSD-based versions have similar directories for System V versions of commands, conventionally */usr/5bin*. */usr/opt/s5/bin* and */usr/opt/s5/sbin* perform a similar function under Tru64.

The /var Directory

As we noted, the */var* directory tree holds data that changes over time. These are its most important subdirectories:

/var/adm
: Administrative directory (home directory of the special *adm* user). This directory traditionally contains the Unix accounting files although many Unix versions have moved them.

/var/cron, /var/news
: */var* contains subdirectories used by many system facilities. These examples are used by the cron and Usenet news facilities, respectively.

/var/log
: Location for log files maintained by many system facilities.

/var/mail
: User mailbox location.

/var/run
: Contains files holding the current process IDs of various system daemons and other server and/or execution instance-specific data.

/var/spool
: Contains subdirectories for Unix subsystems that provide different kinds of spooling services. Some of the tools using */var/spool* subdirectories are the print spooling system, the mail system, and the cron facility.

CHAPTER 3

Essential Administrative Tools and Techniques

The right tools make any job easier, and the lack of them can make some tasks almost impossible. When you need an Allen wrench, nothing but an Allen wrench will do. On the other hand, if you need a Phillips head screwdriver, you might be able to make do with a pocket knife, and occasionally it will even work better.

The first section of this chapter will consider ways the commands and utilities that Unix provides can make system administration easier. Sometimes that means applying common user commands to administrative tasks, sometimes it means putting commands together in unexpected ways, and sometimes it means making smarter and more efficient use of familiar tools. And, once in a while, what will make your life easier is creating tools for users to use, so that they can handle some things for themselves. We'll look at this last topic in Chapter 14.

The second section of this chapter will consider some essential administrative facilities and techniques, including the `cron` subsystem, the syslog facility, strategies for handling the many system log files, and management software packages. We'll close the chapter with a list of Internet software sources.

Getting the Most from Common Commands

In this section, we consider advanced and administrative uses of familiar Unix commands.

Getting Help

The manual page facility is the quintessentially Unix approach to online help: superficially minimalist, often obscure, but mostly complete. It's also easy to use, once you know your way around it.

Undoubtedly, the basics of the `man` command are familiar: getting help for a command, specifying a specific section, using `-k` (or `apropos`) to search for entries for a specific topic, and so on.

There are a couple of man features that I didn't discover until I'd been working on Unix systems for years (I'd obviously never bothered to run man man). The first is that you can request multiple manual pages within a single man command:

```
$ man umount fsck newfs
```

man presents the pages as separate files to the display program, and you can move among them using its normal method (for example, with :n in more).

On FreeBSD, Linux, and Solaris systems, man also has a -a option, which retrieves the specified manual page(s) from every section of the manual. For example, the first command below displays the introductory manual page for every section for which one is available, and the second command displays the manual pages for both the chown command and system call:

```
$ man -a intro
$ man -a chown
```

Manual pages are generally located in a predictable location within the filesystem, often */usr/share/man*. You can configure the man command to search multiple *man* directory trees by setting the MANPATH environment variable to the colon-separated list of desired directories.

Changing the search order

The man command searches the various manual page sections in a predefined order: commands first, followed by system calls and library functions, and then the other sections (i.e., 1, 6, 8, 2, 3, 4, 5, and 7 for BSD-based schemes). The first manual page matching the one specified on the command line is displayed. In some cases, a different order might make more sense. Many operating systems allow this ordering scheme to be customized via the *MANSECTS* entry within a configuration file. For example, Solaris allows the search order to be customized via the *MANSECTS* entry in the */usr/share/man/man.cf* configuration file. You specify a list of sections in the order in which you want them to be searched:

```
MANSECTS=8,1,2,3,4,5,6,7
```

This ordering brings administrative command sections to the beginning of the list.

Here are the available ordering customization locations for the versions we are considering that offer this feature:

FreeBSD
: *MANSECT* environment variable (colon-separated)

Linux (Red Hat)
: *MANSECT* in */etc/man.config* (colon-separated)

Linux (SuSE)
: *SECTION* in */etc/manpath.config* (space-separated)

Solaris
: *MANSECTS* in */usr/share/man/man.cf* and/or the top level directory of any manual page tree (comma-separated)

Setting up man –k

It's probably worth mentioning how to get `man -k` to work if your system claims to support it, but nothing comes back when you use it. This command (and its alias `apropos`) uses a data file indexing all available manual pages. The file often must be initially created by the system administrator, and it may also need to be updated from time to time.

On most systems, the command to create the index file is `makewhatis`, and it must be run by *root*. The command does not require any arguments except on Solaris systems, where the top-level manual page subdirectory is given:

```
# makewhatis                  Most systems
# makewhat /usr/share/man     Solaris
```

On AIX, HP-UX, and Tru64, the older `catman -w` command is used instead.

Piping into grep and awk

As you undoubtedly already know, the `grep` command searches its input for lines containing a given pattern. Users commonly use `grep` to search files. What might be new is some of the ways `grep` is useful in pipes with many administrative commands. For example, if you want to find out about all of a certain user's current processes, pipe the output of the `ps` command to `grep` and search for her username:

```
% ps aux | grep chavez
chavez   8684 89.5  9.627680 5280 ?  R N  85:26 /home/j90/l988
root    10008 10.0  0.8 1408  352 p2 S     0:00 grep chavez
chavez   8679  0.0  1.4 2048  704 ?  I N   0:00 -csh (csh)
chavez   8681  0.0  1.3 2016  672 ?  I N   0:00 /usr/nqs/sc1
chavez   8683  0.0  1.3 2016  672 ?  I N   0:00 csh -cb rj90
chavez   8682  0.0  2.6 1984 1376 ?  I N   0:00 j90
```

This example uses the BSD version of `ps`, using the options that list every single process on the system,* and then uses `grep` to pick out the ones belonging to user *chavez*. If you'd like the header line from `ps` included as well, use a command like:

```
% ps -aux | egrep 'chavez|PID'
```

Now that's a lot to type every time, but you could define an alias if your shell supports them. For example, in the C shell you could use this one:

```
% alias pu "ps -aux | egrep '\!:1|PID'"
% pu chavez
```

* Under HP-UX and for Solaris' */usr/bin/ps*, the corresponding command is `ps -ef`.

```
USER    PID %CPU  %MEM SZ    RSS TT  STAT TIME  COMMAND
chavez  8684 89.5  9.6 27680 5280 ?  R N  85:26 /home/j90/l988
...
```

Another useful place for grep is with man -k. For instance, I once needed to figure out where the error log file was on a new system—the machine kept displaying annoying messages from the error log indicating that disk 3 had a hardware failure. Now, I already knew that, and it had even been fixed. I tried man -k error: 64 matches; man -k log was even worse: 122 manual pages. But man -k log | grep error produced only 9 matches, including a nifty command to blast error log entries older than a given number of days.

The awk command is also a useful component in pipes. It can be used to selectively manipulate the output of other commands in a more general way than grep. A complete discussion of awk is beyond the scope of this book, but a few examples will show you some of its capabilities and enable you to investigate others on your own.

One thing awk is good for is picking out and possibly rearranging columns within command output. For example, the following command produces a list of all users running the quake game:

```
$ ps -ef | grep "[q]uake" | awk '{print $1}'
```

This awk command prints only the first field from each line of ps output passed to it by grep. The search string for grep may strike you as odd, since the brackets enclose only a single character. The command is constructed that way so that the ps line for the grep command itself will not be selected (since the string "quake" does not appear in it). It's basically a trick to avoid having to add grep -v grep to the pipe between the grep and awk commands.

Once you've generated the list of usernames, you can do what you need to with it. One possibility is simply to record the information in a file:

```
$ (date ; ps -ef | grep "[q]uake" | awk '{print $1 " [" $7 "]"}' \
          | sort | uniq) >> quaked.users
```

This command sends the list of users currently playing quake, along with the CPU time used so far enclosed in square brackets, to the file *quaked.users*, preceding the list with the current date and time. We'll see a couple of other ways to use such a list in the course of this chapter.

awk can also be used to sum up a column of numbers. For example, this command searches the entire local filesystem for files owned by user *chavez* and adds up all of their sizes:

```
# find / -user chavez -fstype 4.2 ! -name /dev/\* -ls | \
  awk '{sum+=$7}; END {print "User chavez total disk use = " sum}'
User chavez total disk use = 41987453
```

The awk component of this command accumulates a running total of the seventh column from the find command that holds the number of bytes in each file, and it

prints out the final value after the last line of its input has been processed. awk can also compute averages; in this case, the average number of bytes per file would be given by the expression *sum/NR* placed into the command's *END* clause. The denominator *NR* is an awk internal variable. It holds the line number of the current input line and accordingly indicates the total number of lines read once all of them have been processed.

awk can be used in a similar way with the date command to generate a filename based upon the current date. For example, the following command places the output of the *sys_doc* script into a file named for the current date and host:

```
$ sys_doc  > `date | awk '{print $3 $2 $6}'`.`hostname`.sysdoc
```

If this command were run on October 24, 2001, on host *ophelia*, the filename generated by the command would be *24Oct2001.ophelia.sysdoc*.

Recent implementations of date allow it to generate such strings on its own, eliminating the need for awk. The following command illustrates these features. It constructs a unique filename for a scratch file by telling date to display the literal string *junk_* followed by the day of the month, short form month name, 2-digit year, and hour, minutes and seconds of the current time, ending with the literal string *.junk*:

```
$ date +junk_%d%b%y%H%M%S.junk
junk_08Dec01204256.junk
```

We'll see more examples of grep and awk later in this chapter.

Is All of This Really Necessary?

If all of this fancy pipe fitting seems excessive to you, be assured that I'm not telling you about it for its own sake. The more you know the ins and outs of Unix commands—both basic and obscure—the better prepared you'll be for the inevitable unexpected events that you will face. For example, you'll be able to come up with an answer quickly when the division director (or department chair or whoever) wants to know what percentage of the aggregate disk space in a local area network is used by the *chem* group. Virtuosity and wizardry needn't be goals in themselves, but they will help you develop two of the seven cardinal virtues of system administration: *flexibility* and *ingenuity*. (I'll tell you what the others are in future chapters.)

Finding Files

Another common command of great use to a system administrator is find. find is one of those commands that you wonder how you ever lived without—once you learn it. It has one of the most obscure manual pages in the Unix canon, so I'll spend a bit of time explaining it (skip ahead if it's already familiar).

`find` locates files with common, specified characteristics, searching anywhere on the system you tell it to look. Conceptually, `find` has the following syntax:*

```
# find starting-dir(s) matching-criteria-and-actions
```

Starting-dir(s) is the set of directories where `find` should start looking for files. By default, `find` searches all directories underneath the listed directories. Thus, specifying / as the starting directory would search the entire filesystem.

The *matching-criteria* tell `find` what sorts of files you want to look for. Some of the most useful are shown in Table 3-1.

Table 3-1. find command matching criteria options

Option	Meaning
`-atime` *n*	File was last accessed exactly *n* days ago.
`-mtime` *n*	File was last modified exactly *n* days ago.
`-newer` *file*	File was modified more recently than *file* was.
`-size` *n*	File is *n* 512-byte blocks long (rounded up to next block).
`-type` *c*	Specifies the file type: *f*=plain file, *d*=directory, etc.
`-fstype` *typ*	Specifies filesystem type.
`-name` *nam*	The filename is *nam*.
`-perm` *p*	The file's access mode is *p*.
`-user` *usr*	The file's owner is *usr*.
`-group` *grp*	The file's group owner is *grp*.
`-nouser`	The file's owner is not listed in the password file.
`-nogroup`	The file's group owner is not listed in the group file.

These may not seem all that useful—why would you want a file accessed exactly three days ago, for instance? However, you may precede time periods, sizes, and other numeric quantities with a plus sign (meaning "more than") or a minus sign (meaning "less than") to get more useful criteria. Here are some examples:

```
-mtime +7       Last modified more than 7 days ago
-atime -2       Last accessed less than 2 days ago
-size +100      Larger than 50K
```

You can also include wildcards with the `-name` option, provided that you quote them. For example, the criteria `-name '*.dat'` specifies all filenames ending in *.dat*.

Multiple conditions are joined with AND by default. Thus, to look for files last accessed more than two months ago and last modified more than four months ago, you would use these options:

```
-atime +60 -mtime +120
```

* Syntactically, `find` does not distinguish between file-selection options and action-related options, but it is often helpful to think of them as separate types as you learn to use `find`.

Options may also be joined with `-o` for OR combination, and grouping is allowed using escaped parentheses. For example, the matching criteria below specifies files last accessed more than seven days ago or last modified more than 30 days ago:

```
\( -atime +7 -o -mtime +30 \)
```

An exclamation point may be used for NOT (be sure to quote it if you're using the C shell). For example, the matching criteria below specify all *.dat* files except *gold.dat*:

```
! -name gold.dat -name \*.dat
```

The `-perm` option allows you to search for files with a specific access mode (numeric form). Using an unsigned value specifies files with exactly that permission setting, and preceding the value with a minus sign searches for files with *at least* the specified access. (In other words, the specified permission mode is XORed with the file's permission setting.) Here are some examples:

`-perm 755`	*Permission = rwxr-xr-x*
`-perm -002`	*World-writeable files*
`-perm -4000`	*Setuid access is set*
`-perm -2000`	*Setgid access is set*

The *actions* options tell `find` what to do with each file it locates that matches all the specified criteria. Some available actions are shown in Table 3-2.

Table 3-2. find actions

Option	Meaning
`-print`	Display pathname of matching file.
`-ls`[a]	Display long directory listing for matching file.
`-exec cmd`	Execute command on file.
`-ok cmd`	Prompt before executing command on file.
`-xdev`	Restrict the search to the filesystem of the starting directory (typically used to bypass mounted remote filesystems).
`-prune`	Don't descend into directories encountered.

[a] Not available under HP-UX.

The default on many newer systems is `-print`, although forgetting to include it on older systems like SunOS will result in a successful command with no output. Commands for `-exec` and `-ok` must end with an escaped semicolon (`\;`). The form `{}` may be used in commands as a placeholder for the pathname of each found file. For example, to delete each matching file as it is found, specify the following option to the `find` command:

```
-exec rm -f {} \;
```

Note that there are no spaces between the opening and closing curly braces. The curly braces may only appear once within the command.

Now let's put the parts together. The command below lists the pathname of all C source files under the current directory:

```
$ find . -name \*.c -print
```

The starting directory is "." (the current directory), the matching criteria specify filenames ending in *.c*, and the action to be performed is to display the pathname of each matching file. This is a typical user use for `find`. Other common uses include searching for misplaced files and feeding file lists to `cpio`.

`find` has many administrative uses, including:

- Monitoring disk use
- Locating files that pose potential security problems
- Performing recursive file operations

For example, `find` may be used to locate large disk files. The command below displays a long directory listing for all files under */chem* larger than 1 MB (2048 512-byte blocks) that haven't been modified in a month:

```
$ find /chem -size +2048 -mtime +30 -exec ls -l {} \;
```

Of course, we could also use `-ls` rather than the `-exec` clause. In fact, it is more efficient because the directory listing is handled by `find` internally (rather than having to spawn a subshell for every file). To search for files not modified in a month or not accessed in three months, use this command:

```
$ find /chem -size +2048 \( -mtime +30 -o -atime +120 \) -ls
```

Such old, large files might be candidates for tape backup and deletion if disk space is short.

`find` can also delete files automatically as it finds them. The following is a typical administrative use of `find`, designed to automatically delete old junk files on the system:

```
# find / \( -name a.out -o -name core -o -name '*~'\
    -o -name '.*~' -o -name '#*#' \) -type f -atime +14 \
    -exec rm -f {} \; -o -fstype nfs -prune
```

This command searches the entire filesystem and removes various editor backup files, core dump files, and random executables (*a.out*) that haven't been accessed in two weeks and that don't reside on a remotely mounted filesystem. The logic is messy: the final `-o` option ORs all the options that preceded it with those that followed it, each of which is computed separately. Thus, the final operation finds files that match either of two criteria:

- The filename matches, it's a plain file, and it hasn't been accessed for 14 days.
- The filesystem type is *nfs* (meaning a remote disk).

If the first criteria set is true, the file gets removed; if the second set is true, a "prune" action takes place, which says "don't descend any lower into the directory tree."

Thus, every time `find` comes across an NFS-mounted filesystem, it will move on, rather than searching its entire contents as well.

Matching criteria and actions may be placed in any order, and they are evaluated from left to right. For example, the following `find` command lists all regular files under the directories */home* and */aux1* that are larger than 500K and were last accessed over 30 days ago (done by the options through `-print`); additionally, it removes those named *core*:

```
# find /home /aux1 -type f -atime +30 -size +1000 -print \
    -name core -exec rm {} \;
```

`find` also has security uses. For example, the following `find` command lists all files that have setuid or setgid access set (see Chapter 7).

```
# find / -type f \( -perm -2000 -o -perm -4000 \) -print
```

The output from this command could be compared to a saved list of setuid and setgid files, in order to locate any newly created files requiring investigation:

```
# find / \( -perm -2000 -o -perm -4000 \) -print | \
  diff - files.secure
```

`find` may also be used to perform the same operation on a selected group of files. For example, the command below changes the ownership of all the files under user *chavez*'s home directory to user *chavez* and group *physics*:

```
# find /home/chavez -exec chown chavez {} \; \
                    -exec chgrp physics {} \;
```

The following command gathers all C source files anywhere under */chem* into the directory */chem1/src*:

```
# find /chem -name '*.c' -exec mv {} /chem1/src \;
```

Similarly, this command runs the script *prettify* on every C source file under */chem*:

```
# find /chem -name '*.c' -exec /usr/local/bin/prettify {} \;
```

Note that the full pathname for the script is included in the `-exec` clause.

Finally, you can use the `find` command as a simple method for tracking changes that have been made to a system in the course of a certain time period or as the result of a certain action. Consider these commands:

```
# touch /tmp/starting_time
# perform some operation
# find / -newer /tmp/starting_time
```

The output of the final `find` command displays all files modified or added as a result of whatever action was performed. It does not directly tell you about deleted files, but it lists modified directories (which can be an indirect indication).

Repeating Commands

find is one solution when you need to perform the same operation on a group of files. The xargs command is another way of automating similar commands on a group of objects; xargs is more flexible than find because it can operate on any set of objects, regardless of what kind they are, while find is limited to files and directories.

xargs is most often used as the final component of a pipe. It appends the items it reads from standard input to the Unix command given as its argument. For example, the following command increases the nice number of all quake processes by 10, thereby lowering each process's priority:

```
# ps -ef | grep "[q]uake" | awk '{print $2}' | xargs renice +10
```

The pipe preceding the xargs command extracts the process ID from the second column of the ps output for each instance of quake, and then xargs runs renice using all of them. The renice command takes multiple process IDs as its arguments, so there is no problem sending all the PIDs to a single renice command as long as there are not a truly inordinate number of quake processes.

You can also tell xargs to send its incoming arguments to the specified command in groups by using its -n option, which takes the number of items to use at a time as its argument. If you wanted to run a script for each user who is currently running quake, for example, you could use this command:

```
# ps -ef | grep "[q]uake" | awk '{print $1}' | xargs -n1 warn_user
```

The xargs command will take each username in turn and use it as the argument to warn_user.

So far, all of the xargs commands we've look at have placed the incoming items at the end of the specified command. However, xargs also allows you to place each incoming line of input at a specified position within the command to be executed. To do so, you include its -i option and use the form {} as placeholder for each incoming line within the command. For example, this command runs the System V chargefee utility for each user running quake, assessing them 10000 units:

```
# ps -ef | grep "[q]uake" | awk '{print $1}' | \
  xargs -i chargefee {} 10000
```

If curly braces are needed elsewhere within the command, you can specify a different pair of placeholder characters as the argument to -i.

Substitutions like this can get rather complicated. xargs's -t option displays each constructed command before executing, and the -p option allows you to selectively execute commands by prompting you before each one. Using both options together provides the safest execution mode and also enables you to nondestructively debug a command or script by answering no for every offered command.

-i and -n don't interact the way you might think they would. Consider this command:

```
$ echo a b c d e f | xargs -n3 -i echo before {} after
before a b c d e f after
$ echo a b c d e f | xargs -i -n3 echo before {} after
before {} after a b c
before {} after d e f
```

You might expect that these two commands would be equivalent and that they would both produce two lines of output:

```
before a b c after
before d e f after
```

However, neither command produces this output, and the two commands do not operate identically. What is happening is that -i and -n conflict with one another, and the one appearing last wins. So, in the first command, -i is what is operative, and each *line* of input is inserted into the echo command. In the second command, the -n3 option is used, three arguments are placed at the end of each echo command, and the curly braces are treated as literal characters.

Our first use of -i worked properly because the usernames are coming from separate lines in the ps command output, and these lines are retained as they flow through the pipe to xargs.

If you want xargs to execute commands containing pipes, I/O redirection, compound commands joined with semicolons, and so on, there's a bit of a trick: use the -c option to a shell to execute the desired command. I occasionally want to look at the final lines of a group of files and then view all of them a screen at a time. In other words, I'd like to run a command like this and have it "work":

```
$ tail test00* | more
```

On most systems, this command displays lines only from the last file. However, I can use xargs to get what I want:

```
$ ls -1 test00* | xargs -i /usr/bin/sh -c \
  'echo "****** {}:"; tail -15 {}; echo ""' | more
```

This displays the last 15 lines of each file, preceded by a header line containing the filename and followed by a blank line for readability.

You can use a similar method for lots of other kinds of repetitive operations. For example, this command sorts and de-dups all of the *.dat* files in the current directory:

```
$ ls *.dat | xargs -i /usr/bin/sh -c "sort -u -o {} {}"
```

Creating Several Directory Levels at Once

Many people are unaware of the options offered by the mkdir command. These options allow you to set the file mode at the same time as you create a new directory and to create multiple levels of subdirectories with a single command, both of which can make your use of mkdir much more efficient.

For example, each of the following two commands sets the mode on the new directory to *rwxr-xr-x*, using `mkdir`'s `-m` option:

```
$ mkdir -m 755 ./people
$ mkdir -m u=rwx,go=rx ./places
```

You can use either a numeric mode or a symbolic mode as the argument to the `-m` option. You can also use a relative symbolic mode, as in this example:

```
$ mkdir -m g+w ./things
```

In this case, the mode changes are applied to the default mode as set with the `umask` command.

`mkdir`'s `-p` option tells it to create any missing parents required for the subdirectories specified as its arguments. For example, the following command will create the subdirectories *./a* and *./a/b* if they do not already exist and then create *./a/b/c*:

```
$ mkdir -p ./a/b/c
```

The same command without `-p` will give an error if all of the parent subdirectories are not already present.

Duplicating an Entire Directory Tree

It is fairly common to need to move or duplicate an entire directory tree, preserving not only the directory structure and file contents but also the ownership and mode settings for every file. There are several ways to accomplish this, using `tar`, `cpio`, and sometimes even `cp`. I'll focus on `tar` and then look briefly at the others at the end of this section.

Let's make this task more concrete and assume we want to copy the directory */chem/olddir* as */chem1/newdir* (in other words, we want to change the name of the *olddir* subdirectory as part of duplicating its entire contents). We can take advantage of `tar`'s `-p` option, which restores ownership and access modes along with the files from an archive (it must be run as *root* to set file ownership), and use these commands to create the new directory tree:

```
# cd /chem1
# tar -cf - -C /chem olddir | tar -xvpf -
# mv olddir newdir
```

The first `tar` command creates an archive consisting of */chem/olddir* and all of the files and directories underneath it and writes it to standard output (indicated by the `-` argument to the `-f` option). The `-C` option sets the current directory for the first `tar` command to */chem*. The second `tar` command extracts files from standard input (again indicated by `-f -`), retaining their previous ownership and protection. The second `tar` command gives detailed output (requested with the `-v` option). The final `mv` command changes the name of the newly created subdirectory of */chem1* to *newdir*.

If you want only a subset of the files and directories under *olddir* to be copied to *newdir*, you would vary the previous commands slightly. For example, these

commands copy the *src*, *bin*, and *data* subdirectories and the *logfile* and *.profile* files from *olddir* to *newdir*, duplicating their ownership and protection:

```
# mkdir /chem1/newdir
set ownership and protection for newdir if necessary
# cd /chem1/olddir
# tar -cvf - src bin data logfile.* .profile  |\
  tar -xvpf - -C /chem/newdir
```

The first two commands are necessary only if */chem1/newdir* does not already exist.

This command performs a similar operation, copying only a single branch of the subtree under *olddir*:

```
# mkdir /chem1/newdir
set ownership and protection for newdir if necessary
# cd /chem1/newdir
# tar -cvf - -C /chem/olddir src/viewers/rasmol | tar -xvpf -
```

These commands create */chem1/newdir/src* and its *viewers* subdirectory but place nothing in them but *rasmol*.

If you prefer `cpio` to `tar`, `cpio` can perform similar functions. For example, this command copies the entire *olddir* tree to */chem1* (again as *newdir*):

```
# mkdir /chem1/newdir
set ownership and protection for newdir if necessary
# cd /chem1/olddir
# find . -print | cpio -pdvm /chem1/newdir
```

On all of the systems we are considering, the `cp` command has a `-p` option as well, and these commands create *newdir*:

```
# cp -pr /chem/olddir /chem1
# mv /chem1/olddir /chem1/newdir
```

The `-r` option stands for recursive and causes `cp` to duplicate the source directory structure in the new location.

Be aware that `tar` works differently than `cp` does in the case of symbolic links. `tar` recreates links in the new location, while `cp` converts symbolic links to regular files.

Comparing Directories

Over time, the two directories we considered in the last section will undoubtedly both change. At some future point, you might need to determine the differences between them. `dircmp` is a special-purpose utility designed to perform this very operation.* `dircmp` takes the directories to be compared as its arguments:

```
$ dircmp /chem/olddir /chem1/newdir
```

* On FreeBSD and Linux systems, `diff -r` provides the equivalent functionality.

dircmp produces voluminous output even when the directories you're comparing are small. There are two main sections to the output. The first one lists files that are present in only one of the two directory trees:

```
Mon Jan 4 1995 /chem/olddir only and /chem1/newdir only  Page 1
./water.dat                  ./hf.dat
./src/viewers/rasmol/init.c  ./h2f.dat
...
```

All pathnames in the report are relative to the directory locations specified on the command line. In this case, the files in the left column are present only under */chem/olddir*, and those in the right column are present only at the new location.

The second part of the report indicates whether the files present in both directory trees are the same or different. Here are some typical lines from this section of the report:

```
same        ./h2o.dat
different   ./hcl.dat
```

The default output from dircmp indicates only whether the corresponding files are the same or not, and sometimes this is all you need to know. If you want to know exactly what the differences are, you can include the -d to dircmp, which tells it to run diff for each pair of differing files (since it uses diff, this works only for text files). On the other hand, if you want to decrease the amount of output by limiting the second section of the report to files that differ, include the -s option on the dircmp command.

Deleting Pesky Files

When I teach courses for new Unix users, one of the early exercises consists of figuring out how to delete the files *–delete_me* and *delete me* (with the embedded space in the second case).* Occasionally, however, a user winds up with a file that he just can't get rid of, no matter how creative he is in using rm. At that point, he will come to you. If there is a way to get rm to do the job, show it to him, but there are some files that rm just can't handle. For example, it is possible for some buggy application program to put a file into a bizarre, inconclusive state. Users can also create such files if they experiment with certain filesystem manipulation tools (which they probably shouldn't be using in the first place).

One tool that can take care of such intransigent files is the directory editor feature of the GNU emacs text editor. It is also useful to show this feature to users who just can't get the hang of how to quote strange filenames.

This is the procedure for deleting a file with emacs:

1. Invoke emacs on the directory in question, either by including its path on the command line or by entering its name at the prompt produced by Ctrl-X Ctrl-F.

* There are lots of solutions. One of the simplest is rm delete\ me ./-delete_me.

2. Opening the directory causes `emacs` to automatically enter its directory editing mode. Move the cursor to the file in question using the usual `emacs` commands.
3. Enter a `d`, which is the directory editing mode subcommand to mark a file for deletion. You can also use `u` to unmark a file, `#` to mark all auto-save files, and `~` to mark all backup files.
4. Enter the `x` subcommand, which says to delete all marked files, and answer the confirmation prompt in the affirmative.
5. At this point the file will be gone, and you can exit from `emacs`, continue other editing, or do whatever you need to do next.

`emacs` can also be useful for viewing directory contents when they include files with bizarre characters embedded within them. The most amusing example of this that I can cite is a user who complained to me that the `ls` command beeped at him every time he ran it. It turned out that this only happened in his home directory, and it was due to a file with a Ctrl-G in the middle of the name. The filename looked fine in `ls` listings because the Ctrl-G character was being interpreted, causing the beep. Control characters become visible when you look at the directory in `emacs`, and so the problem was easily diagnosed and remedied (using the `r` subcommand to `emacs`'s directory editing mode that renames a file).

Putting a Command in a Cage

As we'll discuss in detail later, system security inevitably involves tradeoffs between convenience and risk. One way to mitigate the risks arising from certain inherently dangerous commands and subsystems is to isolate them from the rest of the system. This is accomplished with the `chroot` command.

The `chroot` command runs another command from an alternate location within the filesystem, making the command think that that the location is actually the root directory of the filesystem. `chroot` takes one argument, which is the alternate top-level directory. For example, the following command runs the `sendmail` daemon, using the directory */jail* as the new root directory:

```
# chroot /jail sendmail -bd -q1Om
```

The `sendmail` process will treat */jail* as its root directory. For example, when `sendmail` looks for the mail aliases database, which it expects to be located in */etc/aliases*, it will actually access the file */jail/etc/aliases*. In order for `sendmail` to work properly in this mode, a minimal filesystem needs to be set up under */jail* containing all the files and directories that `sendmail` needs.

Running a daemon or subsystem as a user created specifically for that purpose (rather than *root*) is sometimes called *sandboxing*. This security technique is recommended wherever feasible, and it is often used in conjunction with chrooting for added security. See "Managing DNS Servers" in Chapter 8 for a detailed example of this technique.

FreeBSD also has a facility called `jail`, which is a stronger versions of `chroot` that allows you to specify access restrictions for the isolated command.

Starting at the End

Perhaps it's appropriate that we consider the `tail` command near the end of this section on administrative uses of common commands. `tail`'s principal function is to display the last 10 lines of a file (or standard input). `tail` also has a `-f` option that displays new lines as they are added to the end of a file; this mode can be useful for monitoring the progress of a command that writes periodic status information to a file. For example, these commands start a background backup with `tar`, saving its output to a file, and monitor the operation using `tail -f`:

```
$ tar -cvf /dev/rmt1 /chem /chem1 > 24oct94_tar.toc &
$ tail -f 24oct94_tar.toc
```

The information that `tar` displays about each file as it is written to tape is eventually written to the table of contents file and displayed by `tail`. The advantage that this method has over the `tee` command is that the `tail` command may be killed and restarted as many times as you like without affecting the `tar` command.

Some versions of `tail` also include a `-r` option, which will display the lines in a file in reverse order, which is occasionally useful. HP-UX does not support this option, and Linux provides this feature in the `tac` command.

Be Creative

As a final example of the creative use of ordinary commands, consider the following dilemma. A user tells you his workstation won't reboot. He says he was changing his system's boot script but may have deleted some files in */etc* accidentally. You go over to it, type `ls`, and get a message about some missing shared libraries. How do you poke around and find out what files are there?

The answer is to use the simplest Unix command there is, *echo*, along with the wildcard mechanism, both of which are built into every shell, including the statically linked one available in single user mode.

To see all the files in the current directory, just type:

```
$ echo *
```

This command tells the shell to display the value of "*", which of course expands to all files not beginning with a period in the current directory.

By using `echo` together with `cd` (also a built-in shell command), I was able to get a pretty good idea of what had happened. I'll tell you the rest of this story at the end of Chapter 4.

Essential Administrative Techniques

In this section, we consider several system facilities with which system administrators need to be intimately familiar.

Periodic Program Execution: The cron Facility

`cron` is a Unix facility that allows you to schedule programs for periodic execution. For example, you can use `cron` to call a particular remote site every hour to exchange email, to clean up editor backup files every night, to back up and then truncate system log files once a month, or to perform any number of other tasks. Using `cron`, administrative functions are performed without any explicit action by the system administrator (or any other user).*

For administrative purposes, `cron` is useful for running commands and scripts according to a preset schedule. `cron` can send the resulting output to a log file, as a mail or terminal message, or to a different host for centralized logging. The `cron` command starts the `crond` daemon, which has no options. It is normally started automatically by one of the system initialization scripts.

Table 3-3 lists the components of the `cron` facility on the various Unix systems we are considering. We will cover each of them in the course of this section.

Table 3-3. Variations on the cron facility

Component	Location and information
crontab files	**Usual:** */var/spool/cron/crontabs* **FreeBSD:** */var/cron/tabs, /etc/crontab* **Linux:** ***/var/spool/cron*** (Red Hat) */var/spool/cron/tabs* (SuSE), */etc/crontab* (both)
crontab format	**Usual:** System V (no username field) **BSD:** */etc/crontab* (requires username as sixth field)
cron.allow and *cron.deny* files	**Usual:** */var/adm/cron* **FreeBSD:** */var/cron* **Linux:** */etc* (Red Hat), */var/spool/cron* (SuSE) **Solaris:** */etc/cron.d*
Related facilities	**Usual:** none **FreeBSD:** `periodic` utility **Linux:** */etc/cron.** (*hourly*,*daily*,*weekly*,*monthly*) **Red Hat:** `anacron` utility[a]

* Note that `cron` is not a general facility for scheduling program execution off-hours; for the latter, use a batch processing command (discussed in "Managing CPU Resources" in Chapter 15).

Table 3-3. Variations on the cron facility (continued)

Component	Location and information
`cron` log file	**Usual:** */var/adm/cron/log* **FreeBSD:** */var/log/cron* **Linux:** */var/log/cron* (Red Hat), not configured (SuSE) **Solaris:** */var/cron/log*
File containing PID of `crond`	**Usual:** not provided **FreeBSD:** */var/run/cron.pid* **Linux:** */var/run/crond.pid* (Red Hat), */var/run/cron.pid* (SuSE)
Boot script that starts `cron`	**AIX:** */etc/inittab* **FreeBSD:** */etc/rc* **HP-UX:** */sbin/init.d/cron* **Linux:** */etc/init.d/cron* **Solaris:** */etc/init.d/cron* **Tru64:** */sbin/init.d/cron*
Boot script configuration file: `cron`-related entries	**AIX:** none used **FreeBSD:** */etc/rc.conf*: *cron_enable="YES"* and *cron_flags="*args-to-cron*"* **HP-UX:** */etc/rc.config.d/cron: CRON=1* **Linux:** none used (Red Hat, SuSE 8), */etc/rc.config: CRON="YES"* (SuSE 7) **Solaris:** */etc/default/cron: CRONLOG=yes* **Tru64:** none used

[a] The Red Hat Linux `anacron` utility is very similar to cron, but it also runs jobs missed due to the system being down when it reboots.

crontab files

What to run and when to run it are specified by *crontab entries*, which comprise the system's `cron` schedule. The name comes from the traditional `cron` configuration file named *crontab*, for "`cron` table."

By default, any user may add entries to the `cron` schedule. Crontab entries are stored in separate files for each user, usually in the directory called */var/spool/cron/crontabs* (see Table 3-3 for exceptions). Users' crontab files are named after their username: for example, */var/spool/cron/crontabs/root*.

The preceding is the System V convention for crontab files. BSD systems traditionally use a single file, */etc/crontab*. FreeBSD and Linux systems still use this file, in addition to those just mentioned.

Crontab files are not ordinarily edited directly but are created and modified with the `crontab` command (described later in this section).

Crontab entries direct `cron` to run commands at regular intervals. Each one-line entry in the crontab file has the following format:

```
minutes  hours  day-of-month  month  weekday  command
```

Whitespace separates the fields. However, the final field, *command*, can contain spaces within it (i.e., the *command* field consists of everything after the space following *weekday*); the other fields must not contain embedded spaces.

The first five fields specify the times at which `cron` should execute *command*. Their meanings are described in Table 3-4.

Table 3-4. Crontab file fields

Field	Meaning	Range
minutes	Minutes after the hour	0-59
hours	Hour of the day	0-23 (0=midnight)
day-of-month	Numeric day within a month	1-31
month	The month of the year	1-12
weekday	The day of the week	0-6 (0=Sunday)

Note that hours are numbered from midnight (0), and weekdays are numbered beginning with Sunday (also 0).

An entry in any of these fields can be a single number, a pair of numbers separated by a dash (indicating a range of numbers), a comma-separated list of numbers and/or ranges, or an asterisk (a wildcard that represents all valid values for that field).

If the first character in an entry is a number sign (#), `cron` treats the entry as a comment and ignores it. This is also an easy way to temporarily disable an entry without permanently deleting it.

Here are some example crontab entries:

```
0,15,30,45 * * * *  (echo ""; date; echo "") >/dev/console
0,10,20,30,40,50 7-18 * * * /usr/sbin/atrun
0 0 * * *  find / -name "*.bak" -type f -atime +7 -exec rm {} \;
0 4 * * *  /bin/sh /var/adm/mon_disk 2>&1 >/var/adm/disk.log
0 2 * * *  /bin/sh /usr/local/sbin/sec_check 2>&1 | mail root
30 3 1 * * /bin/csh /usr/local/etc/monthly 2>&1 >/dev/null
#30 2 * * 0,6  /usr/local/newsbin/news.weekend
```

The first entry displays the date on the console terminal every fifteen minutes (on the quarter hour); notice that the multiple commands are enclosed in parentheses in order to redirect their output as a group. (Technically, this says to run the commands together in a single subshell.) The second entry runs */usr/sbin/atrun* every 10 minutes from 7 A.M. to 6 P.M. daily. The third entry runs a `find` command to remove all *.bak* files not accessed in seven days.

The fourth and fifth lines run a shell script every day, at 4 A.M. and 2 A.M., respectively. The shell to execute the script is specified explicitly on the command line in both cases; the system default shell, usually the Bourne shell, is used if none is explicitly specified. Both lines' entries redirect standard output and standard error, sending both of them to a file in one case and as electronic mail to *root* in the other.