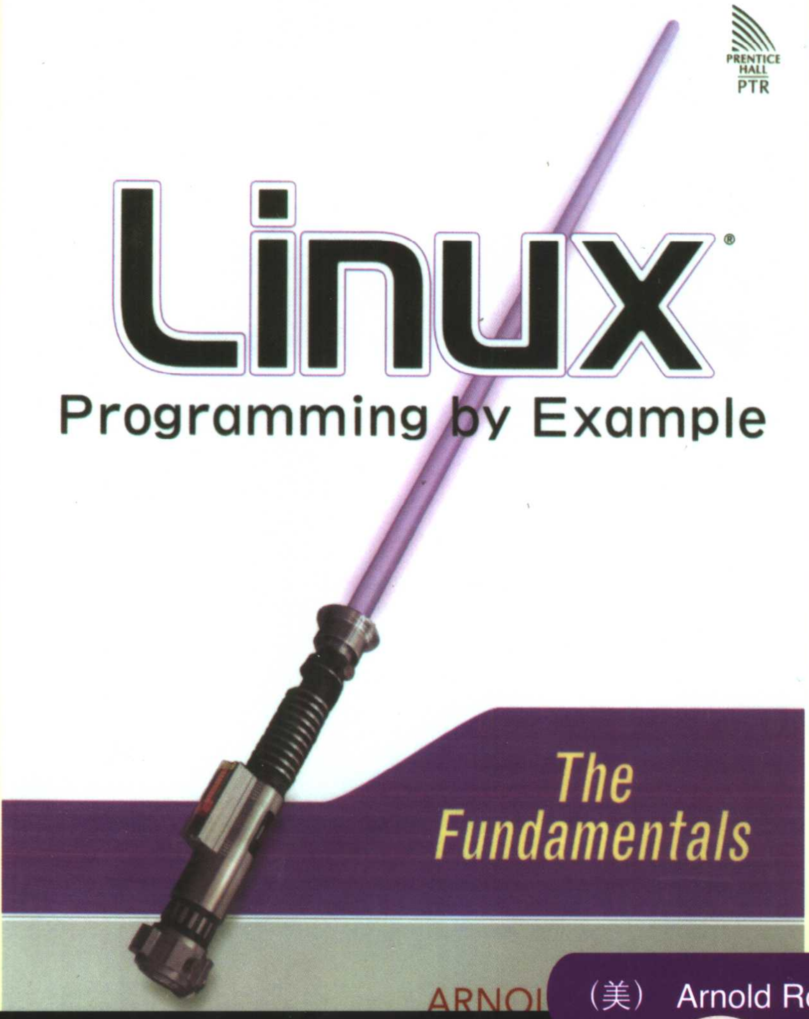
HZ BOOKS

经　典　原　版　书　库

PRENTICE HALL PTR

Linux程序设计

(英文版)


PRENTICE HALL PTR

Linux®

Programming by Example

The Fundamentals

ARNOL

(美) Arnold Robbins 著

机械工业出版社
China Machine Press

Pearson Education

经 典 原 版 书 库

Linux程序设计

（英文版）

Linux Programming by Example

The Fundamentals

（美） Arnold Robbins 著

机械工业出版社
China Machine Press

本书版权登记号：图字：01-2005-3619

图书在版编目（CIP）数据

Linux程序设计（英文版）/（美）罗宾斯（Robbins, A.）著. –北京：机械工业出版社，2005.8

（经典原版书库）
书名原文：Linux Programming by Example: The Fundamentals
ISBN 7-111-16864-X

I. L … II. 罗… III. Linux操作系统－程序设计－英文 IV. TP316.89

中国版本图书馆CIP数据核字（2005）第075105号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
责任编辑：迟振春
北京牛山世兴印刷厂印刷 · 新华书店北京发行所发行
2005年8月第1版第1次印刷
787mm × 1092mm 1/16 · 45.25印张
印数：0 001-3 000册
定价：79.00元

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum’s Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国

家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业的教学度身订造的。其中许多教材均已为M. I. T.，Stanford，U.C. Berkeley，C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

电子邮件：hzjsj@hzbook.com
联系电话：(010) 68995264
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

专家指导委员会

（按姓氏笔画顺序）

To my wife Miriam,
and my children,
Chana, Rivka, Nachum, and Malka.

Preface

One of the best ways to learn about programming is to read well-written programs. This book teaches the fundamental Linux system call APIs—those that form the core of any significant program—by presenting code from production programs that you use every day.

By looking at concrete programs, you can not only see how to use the Linux APIs, but you also can examine the real-world issues (performance, portability, robustness) that arise in writing software.

While the book's title is *Linux Programming by Example*, everything we cover, unless otherwise noted, applies to modern Unix systems as well. In general we use "Linux" to mean the Linux kernel, and "GNU/Linux" to mean the total system (kernel, libraries, tools). Also, we often say "Linux" when we mean all of Linux, GNU/Linux and Unix; if something is specific to one system or the other, we mention it explicitly.

Audience

This book is intended for the person who understands programming and is familiar with the basics of C, at least on the level of *The C Programming Language* by Kernighan and Ritchie. (Java programmers wishing to read this book should understand C pointers, since C code makes heavy use of them.) The examples use both the 1990 version of Standard C and Original C.

In particular, you should be familiar with all C operators, control-flow structures, variable and pointer declarations and use, the string management functions, the use of `exit()`, and the `<stdio.h>` suite of functions for file input/output.

You should understand the basic concepts of *standard input, standard output,* and *standard error* and the fact that all C programs receive an array of character strings representing invocation options and arguments. You should also be familiar with the fundamental command-line tools, such as `cd`, `cp`, `date`, `ln`, `ls`, `man` (and `info` if you have it), `rmdir`, and `rm`, the use of long and short command-line options, environment variables, and I/O redirection, including pipes.

We assume that you want to write programs that work not just under GNU/Linux but across the range of Unix systems. To that end, we mark each interface as to its availability (GLIBC systems only, or defined by POSIX, and so on), and portability advice is included as an integral part of the text.

The programming taught here may be at a lower level than you're used to; that's OK. The system calls are the fundamental building blocks for higher operations and are thus low-level by nature. This in turn dictates our use of C: The APIs were designed for use from C, and code that interfaces them to higher-level languages, such as C++ and Java, will necessarily be lower level in nature, and most likely, written in C. It may help to remember that "low level" doesn't mean "bad," it just means "more challenging."

What You Will Learn

This book focuses on the basic APIs that form the core of Linux programming:

- Memory management
- File input/output
- File metadata
- Processes and signals
- Users and groups
- Programming support (sorting, argument parsing, and so on)
- Internationalization
- Debugging

We have purposely kept the list of topics short. We believe that it is intimidating to try to learn "all there is to know" from a single book. Most readers prefer smaller, more focused books, and the best Unix books are all written that way.

So, instead of a single giant tome, we plan several volumes: one on Interprocess Communication (IPC) and networking, and another on software development and code portability. We also have an eye toward possible additional volumes in a *Linux*

Programming by Example series that will cover topics such as thread programming and GUI programming.

The APIs we cover include both system calls and library functions. Indeed, at the C level, both appear as simple function calls. A *system call* is a direct request for system services, such as reading or writing a file or creating a process. A *library function*, on the other hand, runs at the user level, possibly never requesting any services from the operating system. System calls are documented in section 2 of the reference manual (viewable online with the `man` command), and library functions are documented in section 3.

Our goal is to teach you the use of the Linux APIs by example: in particular, through the use, wherever possible, of both original Unix source code and the GNU utilities. Unfortunately, there aren't as many self-contained examples as we thought there'd be. Thus, we have written numerous small demonstration programs as well. We stress programming principles: especially those aspects of GNU programming, such as "no arbitrary limits," that make the GNU utilities into exceptional programs.

The choice of everyday programs to study is deliberate. If you've been using GNU/Linux for any length of time, you already understand what programs such as `ls` and `cp` do; it then becomes easy to dive straight into *how* the programs work, without having to spend a lot of time learning *what* they do.

Occasionally, we present both higher-level and lower-level ways of doing things. Usually the higher-level standard interface is implemented in terms of the lower-level interface or construct. We hope that such views of what's "under the hood" will help you understand how things work; for all the code you write, you should always use the higher-level, standard interface.

Similarly, we sometimes introduce functions that provide certain functionality and then recommend (with a provided reason) that these functions be avoided! The primary reason for this approach is so that you'll be able to recognize these functions when you see them and thus understand the code using them. A well-rounded knowledge of a topic requires understanding not just what you can do, but what you should and should not do.

Finally, each chapter concludes with exercises. Some involve modifying or writing code. Others are more in the category of "thought experiments" or "why do you think ..." We recommend that you do all of them—they will help cement your understanding of the material.

Small Is Beautiful: Unix Programs

Hoare's law:
"Inside every large program is a small program
struggling to get out."
—C.A.R. Hoare—

Initially, we planned to teach the Linux API by using the code from the GNU utilities. However, the modern versions of even simple command-line programs (like `mv` and `cp`) are large and many-featured. This is particularly true of the GNU variants of the standard utilities, which allow long and short options, do everything required by POSIX, and often have additional, seemingly unrelated options as well (like output highlighting).

It then becomes reasonable to ask, "Given such a large and confusing forest, how can we focus on the one or two important trees?" In other words, if we present the current full-featured program, will it be possible to see the underlying core operation of the program?

That is when *Hoare's law*[1] inspired us to look to the original Unix programs for example code. The original V7 Unix utilities are small and straightforward, making it easy to see what's going on and to understand how the system calls are used. (V7 was released around 1979; it is the common ancestor of all modern Unix systems, including GNU/Linux and the BSD systems.)

For many years, Unix source code was protected by copyrights and trade secret license agreements, making it difficult to use for study and impossible to publish. This is still true of all commercial Unix source code. However, in 2002, Caldera (currently operating as SCO) made the original Unix code (through V7 and 32V Unix) available under an Open Source style license (see Appendix B, "Caldera Ancient UNIX License," page 655). This makes it possible for us to include the code from the early Unix system in this book.

Standards

Throughout the book we refer to several different formal standards. A *standard* is a document describing how something works. Formal standards exist for many things, for example, the shape, placement, and meaning of the holes in the electrical outlet in

[1] This famous statement was made at *The International Workshop on Efficient Production of Large Programs* in Jablonna, Poland, August 10–14, 1970.

your wall are defined by a formal standard so that all the power cords in your country work in all the outlets.

So, too, formal standards for computing systems define how they are supposed to work; this enables developers and users to know what to expect from their software and enables them to complain to their vendor when software doesn't work.

Of interest to us here are:

1. *ISO/IEC International Standard 9899: Programming Languages — C, 1990.* The first formal standard for the C programming language.
2. *ISO/IEC International Standard 9899: Programming Languages — C, Second edition, 1999.* The second (and current) formal standard for the C programming language.
3. *ISO/IEC International Standard 14882: Programming Languages — C++, 1998.* The first formal standard for the C++ programming language.
4. *ISO/IEC International Standard 14882: Programming Languages — C++, 2003.* The second (and current) formal standard for the C++ programming language.
5. *IEEE Standard 1003.1–2001: Standard for Information Technology — Portable Operating System Interface (POSIX®).* The current version of the POSIX standard; describes the behavior expected of Unix and Unix-like systems. This edition covers both the system call and library interface, as seen by the C/C++ programmer, and the shell and utilities interface, seen by the user. It consists of several volumes:
   - *Base Definitions.* The definitions of terms, facilities, and header files.
   - *Base Definitions — Rationale.* Explanations and rationale for the choice of facilities that both are and are not included in the standard.
   - *System Interfaces.* The system calls and library functions. POSIX terms them all "functions."
   - *Shell and Utilities.* The shell language and utilities available for use with shell programs and interactively.

Although language standards aren't exciting reading, you may wish to consider purchasing a copy of the C standard: It provides the final definition of the language. Copies can be purchased from ANSI[2] and from ISO.[3] (The PDF version of the C standard is quite affordable.)

The POSIX standard can be ordered from The Open Group.[4] By working through their publications catalog to the items listed under "CAE Specifications," you can find individual pages for each part of the standard (named "C031" through "C034"). Each one's page provides free access to the online HTML version of the particular volume.

The POSIX standard is intended for implementation on both Unix and Unix-like systems, as well as non-Unix systems. Thus, the base functionality it provides is a subset of what Unix systems have. However, the POSIX standard also defines optional *extensions*—additional functionality, for example, for threads or real-time support. Of most importance to us is the *X/Open System Interface* (XSI) extension, which describes facilities from historical Unix systems.

Throughout the book, we mark each API as to its availability: ISO C, POSIX, XSI, GLIBC only, or nonstandard but commonly available.

Features and Power: GNU Programs

Restricting ourselves to just the original Unix code would have made an interesting history book, but it would not have been very useful in the 21st century. Modern programs do not have the same constraints (memory, CPU power, disk space, and speed) that the early Unix systems did. Furthermore, they need to operate in a multilingual world—ASCII and American English aren't enough.

More importantly, one of the primary freedoms expressly promoted by the Free Software Foundation and the GNU Project[5] is the "freedom to study." GNU programs are intended to provide a large corpus of well-written programs that journeyman programmers can use as a source from which to learn.

[2] `http://www.ansi.org`

[3] `http://www.iso.ch`

[4] `http://www.opengroup.org`

[5] `http://www.gnu.org`

By using GNU programs, we want to meet both goals: show you well-written, modern code from which you will learn how to write good code and how to use the APIs well.

We believe that GNU software is better because it is free (in the sense of "freedom," not "free beer"). But it's also recognized that GNU software is often *technically* better than the corresponding Unix counterparts, and we devote space in Section 1.4, "Why GNU Programs Are Better," page 14, to explaining why.

A number of the GNU code examples come from `gawk` (GNU `awk`). The main reason is that it's a program with which we're very familiar, and therefore it was easy to pick examples from it. We don't otherwise make any special claims about it.

Summary of Chapters

Driving a car is a holistic process that involves multiple simultaneous tasks. In many ways, Linux programming is similar, requiring understanding of multiple aspects of the API, such as file I/O, file metadata, directories, storage of time information, and so on.

The first part of the book looks at enough of these individual items to enable studying the first significant program, the V7 `ls`. Then we complete the discussion of files and users by looking at file hierarchies and the way filesystems work and are used.

Chapter 1, "Introduction," page 3,
describes the Unix and Linux file and process models, looks at the differences between Original C and 1990 Standard C, and provides an overview of the principles that make GNU programs generally better than standard Unix programs.

Chapter 2, "Arguments, Options, and the Environment," page 23,
describes how a C program accesses and processes command-line arguments and options and explains how to work with the environment.

Chapter 3, "User-Level Memory Management," page 51,
provides an overview of the different kinds of memory in use and available in a running process. User-level memory management is central to every nontrivial application, so it's important to understand it early on.

Chapter 4, "Files and File I/O," page 83,
discusses basic file I/O, showing how to create and use files. This understanding is important for everything else that follows.

Chapter 5, "Directories and File Metadata," page 117,
describes how directories, hard links, and symbolic links work. It then describes file metadata, such as owners, permissions, and so on, as well as covering how to work with directories.

Chapter 6, "General Library Interfaces — Part 1," page 165,
looks at the first set of general programming interfaces that we need so that we can make effective use of a file's metadata.

Chapter 7, "Putting It All Together: `ls`*," page 207,*
ties together everything seen so far by looking at the V7 `ls` program.

Chapter 8, "Filesystems and Directory Walks," page 227,
describes how filesystems are mounted and unmounted and how a program can tell what is mounted on the system. It also describes how a program can easily "walk" an entire file hierarchy, taking appropriate action for each object it encounters.

The second part of the book deals with process creation and management, interprocess communication with pipes and signals, user and group IDs, and additional general programming interfaces. Next, the book first describes internationalization with GNU `gettext` and then several advanced APIs.

Chapter 9, "Process Management and Pipes," page 283,
looks at process creation, program execution, IPC with pipes, and file descriptor management, including nonblocking I/O.

Chapter 10, "Signals," page 347,
discusses signals, a simplistic form of interprocess communication. Signals also play an important role in a parent process's management of its children.

Chapter 11, "Permissions and User and Group ID Numbers," page 403,
looks at how processes and files are identified, how permission checking works, and how the setuid and setgid mechanisms work.

Chapter 12, "General Library Interfaces — Part 2," page 427,
looks at the rest of the general APIs; many of these are more specialized than the first general set of APIs.

Chapter 13, "Internationalization and Localization," page 485,
explains how to enable your programs to work in multiple languages, with almost no pain.

Chapter 14, "Extended Interfaces," page 529,
describes several extended versions of interfaces covered in previous chapters, as well as covering file locking in full detail.

We round the book off with a chapter on debugging, since (almost) no one gets things right the first time, and we suggest a final project to cement your knowledge of the APIs covered in this book.

Chapter 15, "Debugging," page 567,
describes the basics of the GDB debugger, transmits as much of our programming experience in this area as possible, and looks at several useful tools for doing different kinds of debugging.

Chapter 16, "A Project That Ties Everything Together," page 641,
presents a significant programming project that makes use of just about everything covered in the book.

Several appendices cover topics of interest, including the licenses for the source code used in this book.

Appendix A, "Teach Yourself Programming in Ten Years," page 649,
invokes the famous saying, "Rome wasn't built in a day." So too, Linux/Unix expertise and understanding only come with time and practice. To that end, we have included this essay by Peter Norvig which we highly recommend.

Appendix B, "Caldera Ancient UNIX License," page 655,
covers the Unix source code used in this book.

Appendix C, "GNU General Public License," page 657,
covers the GNU source code used in this book.

Typographical Conventions

Like all books on computer-related topics, we use certain typographical conventions to convey information. *Definitions* or first uses of terms appear in italics, like the word "Definitions" at the beginning of this sentence. Italics are also used for *emphasis*, for citations of other works, and for commentary in examples. Variable items such as arguments or filenames, appear `like this`. Occasionally, we use a bold font when a point needs to be made **strongly**.

Things that exist on a computer are in a constant-width font, such as filenames (`foo.c`) and command names (`ls`, `grep`). Short snippets that you type are additionally enclosed in single quotes: '`ls -l *.c`'.

`$` and `>` are the Bourne shell primary and secondary prompts and are used to display interactive examples. **`User input`** appears in a different font from regular `computer output` in examples. Examples look like this:

```
$ ls -1                     Look at files. Option is digit 1, not letter l
foo
bar
baz
```

We prefer the Bourne shell and its variants (`ksh93`, Bash) over the C shell; thus, all our examples show only the Bourne shell. Be aware that quoting and line-continuation rules are different in the C shell; if you use it, you're on your own![6]

When referring to functions in programs, we append an empty pair of parentheses to the function's name: `printf()`, `strcpy()`. When referring to a manual page (accessible with the `man` command), we follow the standard Unix convention of writing the command or function name in italics and the section in parentheses after it, in regular type: *awk*(1), *printf*(3).

Where to Get Unix and GNU Source Code

You may wish to have copies of the programs we use in this book for your own experimentation and review. All the source code is available over the Internet, and your GNU/Linux distribution contains the source code for the GNU utilities.

6 See the *csh*(1) and *tcsh*(1) manpages and the book *Using csh & tcsh*, by Paul DuBois, O'Reilly & Associates, Sebastopol, CA, USA, 1995. ISBN: 1-56592-132-1.