大学计算机教育国外著名教材系列（影印版）

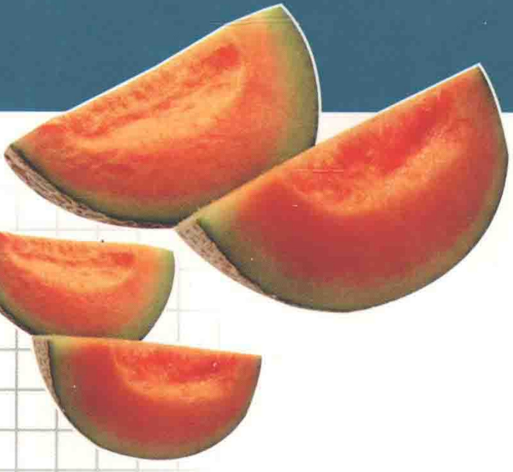
PEARSON Addison Wesley

ADVANCED UNIX PROGRAMMING

SECOND EDITION

高级UNIX程序设计

（第2版）



Marc J. Rochkind 著

PEARSON Education

清华大学出版社

大学计算机教育国外著名教材系列（影印版）

Advanced UNIX Programming

Second Edition

高级 UNIX 程序设计

（第 2 版）

Marc J. Rochkind　著

清 华 大 学 出 版 社

北　京

图书在版编目(CIP)数据

高级 UNIX 程序设计 = Advanced UNIX Programming: 第 2 版 /（美）罗克金德（Rochkind, M. J.）著. —影印本. —北京：清华大学出版社，2006.4
（大学计算机教育国外著名教材系列）
ISBN 7-302-12645-3

Ⅰ. 高…　Ⅱ. 罗…　Ⅲ. UNIX 操作系统－程序设计－高等学校－教材－英文　Ⅳ. TP316.81

中国版本图书馆 CIP 数据核字（2006）第 017815 号

出 版 者：清华大学出版社	地　　址：北京清华大学学研大厦
http://www.tup.com.cn	邮　　编：100084
社总机：010-6277 0175	客户服务：010-6277 6969

印 刷 者：北京四季青印刷厂
装 订 者：三河市李旗庄少明装订厂
发 行 者：新华书店总店北京发行所
开　　本：185×230　印张：45.25
版　　次：2006 年 4 月第 1 版　2006 年 4 月第 1 次印刷
书　　号：ISBN 7-302-12645-3/TP・8082
印　　数：1~3000
定　　价：69.00 元

出 版 说 明

进入 21 世纪，世界各国的经济、科技以及综合国力的竞争将更加激烈。竞争的中心无疑是对人才的竞争。谁拥有大量高素质的人才，谁就能在竞争中取得优势。高等教育，作为培养高素质人才的事业，必然受到高度重视。目前我国高等教育的教材更新较慢，为了加快教材的更新频率，教育部正在大力促进我国高校采用国外原版教材。

清华大学出版社从 1996 年开始，与国外著名出版公司合作，影印出版了“大学计算机教育丛书（影印版）”等一系列引进图书，受到国内读者的欢迎和支持。跨入 21 世纪，我们本着为我国高等教育教材建设服务的初衷，在已有的基础上，进一步扩大选题内容，改变图书开本尺寸，一如既往地请有关专家挑选适用于我国高校本科及研究生计算机教育的国外经典教材或著名教材，组成本套“大学计算机教育国外著名教材系列（影印版）”，以飨读者。深切期盼读者及时将使用本系列教材的效果和意见反馈给我们。更希望国内专家、教授积极向我们推荐国外计算机教育的优秀教材，以利我们把“大学计算机教育国外著名教材系列（影印版）”做得更好，更适合高校师生的需要。

清华大学出版社

Preface

This book updates the 1985 edition of *Advanced UNIX Programming* to cover a few changes that have occurred in the last eighteen years. Well, maybe "few" isn't the right word! And "updates" isn't right either. Indeed, aside from a sentence here and there, this book is all new. The first edition included about 70 system calls; this one includes about 300. And none of the UNIX standards and implementations discussed in this book—POSIX, Solaris, Linux, FreeBSD, and Darwin (Mac OS X)—were even around in 1985. A few sentences from the 1985 Preface, however, are among those that I can leave almost unchanged:

> The subject of this book is UNIX system calls—the interface between the UNIX kernel and the user programs that run on top of it. Those who interact only with commands, like the shell, text editors, and other application programs, may have little need to know much about system calls, but a thorough knowledge of them is essential for UNIX programmers. System calls are the *only* way to access kernel facilities such as the file system, the multitasking mechanisms, and the interprocess communication primitives.
>
> System calls define what UNIX is. Everything else—subroutines and commands—is built on this foundation. While the novelty of many of these higher-level programs has been responsible for much of UNIX's renown, they could as well have been programmed on any modern operating system. When one describes UNIX as elegant, simple, efficient, reliable, and portable, one is referring not to the commands (some of which are none of these things), but to the kernel.

That's all still true, except that, regrettably, the programming interface to the kernel is no longer elegant or simple. In fact, because UNIX development has splintered into many directions over the last couple of decades, and because the principal standards organization, The Open Group, sweeps up almost everything that's out there (1108 functions altogether), the interface is clumsy, inconsistent, redundant, error-prone, and confusing. But it's still efficient, reliably implemented, and portable, and that's why UNIX and UNIX-like systems are so successful. Indeed, the UNIX system-call interface is the only widely implemented portable one we have and are likely to have in our lifetime.

To sort things out, it's not enough to have complete documentation, just as the Yellow Pages isn't enough to find a good restaurant or hotel. You need a guide that tells you what's good and bad, not just what exists. That's the purpose of this book, and why it's different from most other UNIX programming books. I tell you not only how to use the system calls, but also which ones to stay away from because they're unnecessary, obsolete, improperly implemented, or just plain poorly designed.

Here's how I decided what to include in this book: I started with the 1108 functions defined in Version 3 of the Single UNIX Specification and eliminated about 590 Standard C and other library functions that aren't at the kernel-interface level, about 90 POSIX Threads functions (keeping a dozen of the most important), about 25 accounting and logging functions, about 50 tracing functions, about 15 obscure and obsolete functions, and about 40 functions for scheduling and other things that didn't seem to be generally useful. That left exactly 307 for this book. (See Appendix D for a list.) Not that the 307 are all good—some of them are useless, or even dangerous. But those 307 are the ones you need to know.

This book doesn't cover kernel implementation (other than some basics), writing device drivers, C programming (except indirectly), UNIX commands (shell, vi, emacs, etc.), or system administration.

There are nine chapters: Fundamental Concepts, Basic File I/O, Advanced File I/O, Terminal I/O, Processes and Threads, Basic Interprocess Communication, Advanced Interprocess Communication, Networking and Sockets, and Signals and Timers. Read all of Chapter 1, but then feel free to skip around. There are lots of cross-references to keep you from getting lost.

Like the first edition, this new book includes thousands of lines of example code, most of which are from realistic, if simplified, applications such as a shell, a full-screen menu system, a Web server, and a real-time output recorder. The examples are all in C, but I've provided interfaces in Appendices B and C so you can program in C++, Java, or Jython (a variant of Python) if you like.

The text and example code are just resources; you really learn UNIX programming by doing it. To give you something to do, I've included exercises at the end of each chapter. They range in difficulty from questions that can be answered in a few sentences to simple programming problems to semester-long projects.

I used four UNIX systems for nuts-and-bolts research and to test the examples: Solaris 8, SuSE Linux 8 (2.4 kernel), FreeBSD 4.6, and Darwin (the Mac OS X kernel) 6.8. I kept the source on the FreeBSD system, mounted on the others with NFS or Samba.[1]

I edited the code with TextPad on a Windows system and accessed the four test systems with Telnet or SSH (PuTTY) or with the X Window System (XFree86 and Cygwin). Having the text editor and the four Telnet/SSH/Xterm windows open on the same screen turned out to be incredibly convenient, because it takes only a few minutes to write some code and check it out on the four systems. In addition, I usually had one browser window open to the Single UNIX Specification and one to Google, and another window running Microsoft Word for editing the book. With the exception of Word, which is terrible for big documents like books (crashes, mixed-up styles, weak cross-referencing, flakey document-assembly), all of these tools worked great.[2] I used Perl and Python for various things like extracting code samples and maintaining the database of system calls.

All of the example code (free open source), errata, and more is on the book Web site at *www.basepath.com/aup*.

I'd like to thank those who reviewed the drafts or who otherwise provided technical assistance: Tom Cargill, Geoff Clare, Andrew Gierth, Andrew Josey, Brian Kernighan, Barry Margolin, Craig Partridge, and David Schwartz. And, special thanks to one dedicated, meticulous reviewer who asked to remain anonymous. Naturally, none of these folks is to be blamed for any mistakes you find—I get full credit for those.

I'd also like to thank my editor, Mary Franz, who suggested this project a year or so ago. Luckily, she caught me at a time when I was looking into Linux in depth and started to get excited about UNIX all over again. Reminds me a bit of 1972....

I hope you enjoy the book! If you find anything wrong, or if you port the code to any new systems, or if you just want to share your thoughts, please email me at aup@basepath.com.

Marc J. Rochkind
Boulder, Colorado
April, 2004

1. The four systems are running on various old PCs I've collected over the years and on a Mac I bought on eBay for $200. I had no particular reason for using SuSE Linux and have since switched that machine over to RedHat 9.

2. I could have used any of the systems as my base. Windows turned out to be convenient because my big LCD monitor is attached to that system and because I like TextPad (*www.textpad.com*). Information on PuTTY is at *www.chiark.greenend.org.uk/~sgtatham/putty/*. (Google "PuTTY" if that link doesn't work.)

Contents

1 Fundamental Concepts

1.1 A Whirlwind Tour of UNIX and Linux

This section takes you on a quick tour of the facilities provided by the UNIX and Linux kernels. I won't deal with the user programs (commands) that normally come with UNIX, such as `ls`, `vi`, and `grep`. A discussion of these is well outside the scope of this book. And I won't say much about the internals of the kernel (such as how the file system is implemented) either. (From now on, whenever I say UNIX, I mean Linux, too, unless I say otherwise.)

This tour is meant to be a refresher. I'll use terms such as *process* before defining them, because I assume you already know roughly what they mean. If too much sounds new to you, you may want to become more familiar with UNIX before proceeding. (If you don't know what a process is, you definitely need to get more familiar!) There are lots of introductory UNIX books to start with. Two good ones are *The UNIX Programming Environment* [Ker1984] and *UNIX for the Impatient* [Abr1996] (Chapter 2 is a great introduction).[1]

1.1.1 Files

There are several kinds of UNIX files: regular files, directories, symbolic links, special files, named pipes (FIFOs), and sockets. I'll introduce the first four here and the last two in Section 1.1.7.

1.1.1.1 Regular Files

Regular files contain bytes of data, organized into a linear array. Any byte or sequence of bytes may be read or written. Reads and writes start at a byte loca-

1. You'll find the References at the end of the book.

tion specified by the *file offset,* which can be set to any value (even beyond the end of the file). Regular files are stored on disk.

It isn't possible to insert bytes into the middle of a file (spreading the file apart), or to delete bytes from the middle (closing the gap). As bytes are written onto the end of a file, it gets bigger, one byte at a time. A file can be shrunk or enlarged to any length, discarding bytes or adding bytes of zeroes.

Two or more processes can read and write the same file concurrently. The results depend on the order in which the individual I/O requests occur and are in general unpredictable. To maintain order, there are file-locking facilities and *semaphores*, which are system-wide flags that processes can test and set (more on them in Section 1.1.7).

Files don't have names; they have numbers called *i-numbers.* An i-number is an index into an array of *i-nodes,* kept at the front of each region of disk that contains a UNIX file system. Each i-node contains important information about one file. Interestingly, this information doesn't include either the name or the data bytes. It does include the following: type of file (regular, directory, socket, etc.); number of links (to be explained shortly); owner's user and group ID; three sets of access permissions—for the owner, the group, and others; size in bytes; time of last access, last modification, and status change (when the i-node itself was last modified); and, of course, pointers to disk blocks containing the file's contents.

1.1.1.2 Directories and Symbolic Links

Since it's inconvenient to refer to files by i-number, *directories* are provided to allow names to be used. In practice, a directory is almost always used to access a file.

Each directory consists, conceptually, of a two-column table, with a name in one column and its corresponding i-number in the other column. A name/i-node pair is called a *link.* When the UNIX kernel is told to access a file by name, it automatically looks in a directory to find the i-number. Then it gets the corresponding i-node, which contains more information about the file (such as who can access it). If the data itself is to be accessed, the i-node tells where to find it on the disk.

Directories, which are almost like regular files, occupy an i-node and have data. Therefore, the i-node corresponding to a particular name in a directory could be the i-node of another directory. This allows users to arrange their files into the hierarchical structure that's familiar to users of UNIX. A *path* such as `memo/july/smith` instructs the kernel to get the i-node of the *current directory* to locate its data bytes, find `memo` among those data bytes, take the corresponding i-number, get that i-node to locate the `memo` directory's data bytes, find `july` among those, take the corresponding i-number, get the i-node to locate the `july` directory's data bytes, find `smith`, and, finally, take the corresponding i-node, the one associated with `memo/july/smith`.

In following a *relative path* (one that starts with the current directory), how does the kernel know where to start? It simply keeps track of the i-number of the current directory for each process. When a process changes its current directory, it must supply a path to the new directory. That path leads to an i-number, which then is saved as the i-number of the new current directory.

An *absolute path* begins with a / and starts with the *root* directory. The kernel simply reserves an i-number (2, say) for the root directory. This is established when a file system is first constructed. There is a system call to change a process's root directory (to an i-number other than *2*).

Because the two-column structure of directories is used directly by the kernel (a rare case of the kernel caring about the contents of files), and because an invalid directory could easily destroy an entire UNIX system, a program (even if run by the superuser) cannot write a directory as if it were a regular file. Instead, a program manipulates a directory by using a special set of system calls. After all, the only legal writing actions are to add or remove a link.

It is possible for two or more links, in the same or different directories, to refer to the same i-number. This means that the same file may have more than one name. There is no ambiguity when accessing a file by a given path, since only one i-number will be found. It might have been found via another path also, but that's irrelevant. When a link is removed from a directory, however, it isn't immediately clear whether the i-node and the associated data bytes can be thrown away too. That is why the i-node contains a link count. Removing a link to an i-node merely decrements the link count; when the count reaches zero, the kernel discards the file.

There is no structural reason why there can't be multiple links to directories as well as to regular files. However, this complicates the programming of commands that scan the entire file system, so most kernels outlaw it.

Multiple links to a file using i-numbers work only if the links are in the same file system, as i-numbers are unique only within a file system. To get around this, there are also *symbolic links*, which put the path of the file to be linked to in the data part of an actual file. This is more overhead than just making a second directory link somewhere, but it's more general. You don't read and write these symbolic-link files, but instead use special system calls just for symbolic links.

1.1.1.3 Special Files

A *special file* is typically some type of *device* (such as a CD-ROM drive or communications link).[2]

There are two principal kinds of device special files: block and character. *Block special files* follow a particular model: The device contains an array of fixed-size blocks (say, 4096 bytes each), and a pool of kernel *buffers* are used as a cache to speed up I/O. *Character special files* don't have to follow any rules at all. They might do I/O in very small chunks (characters) or very big chunks (disk tracks), and so they're too irregular to use the buffer cache.

The same physical device could have both block and character special files, and, in fact, this is usually true for disks. Regular files and directories are accessed by the file-system code in the kernel via a block special file, to gain the benefits of the buffer cache. Sometimes, primarily in high-performance applications, more direct access is needed. For instance, a database manager can bypass the file system entirely and use a character special file to access the disk (but not the same area that's being used by the file system). Most UNIX systems have a character special file for this purpose that can directly transfer data between a process's address space and the disk using *direct memory access* (*DMA*), which can result in orders-of-magnitude better performance. More robust error detection is another benefit, since the indirectness of the buffer cache tends to make error detection difficult to implement.

A special file has an i-node, but there aren't any data bytes on disk for the i-node to point to. Instead, that part of the i-node contains a *device number*. This is an index into a table used by the kernel to find a collection of subroutines called a *device driver*.

When a system call is executed to perform an operation on a special file, the appropriate device driver subroutine is invoked. What happens then is entirely up to the designer of the device driver; since the driver runs in the kernel, and not as

2. Sometimes named pipes are considered special files, too, but we'll consider them a category of their own.

a user process, it can access—and perhaps modify—any part of the kernel, any user process, and any registers or memory of the computer itself. It is relatively easy to add new device drivers to the kernel, so this provides a hook with which to do many things besides merely interfacing to new kinds of I/O devices. It's the most popular way to get UNIX to do something its designers never intended it to do. Think of it as the approved way to do something wild.

1.1.2 Programs, Processes, and Threads

A *program* is a collection of *instructions* and *data* that is kept in a regular file on disk. In its i-node the file is marked executable, and the file's contents are arranged according to rules established by the kernel. (Another case of the kernel caring about the contents of a file.)

Programmers can create executable files any way they choose. As long as the contents obey the rules and the file is marked executable, the program can be run. In practice, it usually goes like this: First, the source program, in some programming language (C or C++, say), is typed into a regular file, often referred to as a *text file,* because it's arranged into text lines. Next, another regular file, called an *object file,* is created that contains the machine-language translation of the source program. This job is done by a compiler or assembler (which are themselves programs). If this object file is complete (no missing subroutines), it is marked executable and may be run as is. If not, the *linker* (sometimes called a "loader" in UNIX jargon) is used to bind this object file with others previously created, possibly taken from collections of object files called *libraries.* Unless the linker couldn't find something it was looking for, its output is complete and executable.[3]

In order to run a program, the kernel is first asked to create a new *process,* which is an environment in which a *program* executes. A process consists of three segments: *instruction segment,*[4] *user data segment,* and *system data segment.* The program is used to initialize the instructions and user data. After this initialization, the process begins to deviate from the program it is running. Although modern programmers don't normally modify instructions, the data does get modi-

3. This isn't how interpretive languages like Java, Perl, Python, and shell scripts work. For them, the executable is an interpreter, and the program, even if compiled into some intermediate code, is just data for the interpreter and isn't something the UNIX kernel ever sees or cares about. The kernel's customer is the interpreter.

4. In UNIX jargon, the instruction segment is called the "text segment," but I'll avoid that confusing term.

fied. In addition, the process may acquire resources (more memory, open files, etc.) not present in the program.

While the process is running, the kernel keeps track of its *threads,* each of which is a separate flow of control through the instructions, all potentially reading and writing the same parts of the process's data. (Each thread has its own stack, however.) When you're programming, you start with one thread, and that's all you get unless you execute a special system call to create another. So, beginners can think of a process as being single-threaded.[5]

Several concurrently running processes can be initialized from the same program. There is no functional relationship, however, between these processes. The kernel might be able to save memory by arranging for such processes to share instruction segments, but the processes involved can't detect such sharing. By contrast, there is a strong functional relationship between threads in the same process.

A process's *system data* includes attributes such as current directory, open file descriptors, accumulated CPU time, and so on. A process can't access or modify its system data directly, since it is outside of its address space. Instead, there are various system calls to access or modify attributes.

A process is created by the kernel on behalf of a currently executing process, which becomes the *parent* of the new *child* process. The child inherits most of the parent's system-data attributes. For example, if the parent has any files open, the child will have them open too. Heredity of this sort is absolutely fundamental to the operation of UNIX, as we shall see throughout this book. This is different from a thread creating a new thread. Threads in the same process are equal in most respects, and there's no inheritance. All threads have equal access to all data and resources, not copies of them.

1.1.3 Signals

The kernel can send a *signal* to a process. A signal can be originated by the kernel itself, sent from a process to itself, sent from another process, or sent on behalf of the user.

5. Not every version of UNIX supports multiple threads. They're part of an optional feature called POSIX Threads, or "pthreads," and were introduced in the mid-1990s. More on POSIX in Section 1.5 and threads in Chapter 5.