

PEARSON

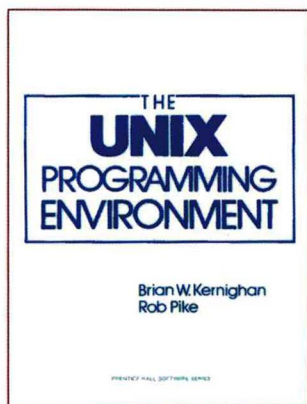
UNIX

编程环境（英文版）

[美] Brian W. Kernighan Rob Pike 著

The Unix Programming Environment

- 经久不衰的UNIX经典教程
- 两位UNIX大师合力之作，浸透了UNIX的设计思想
- 启发读者体会编程方法、思想以及环境的奥秘



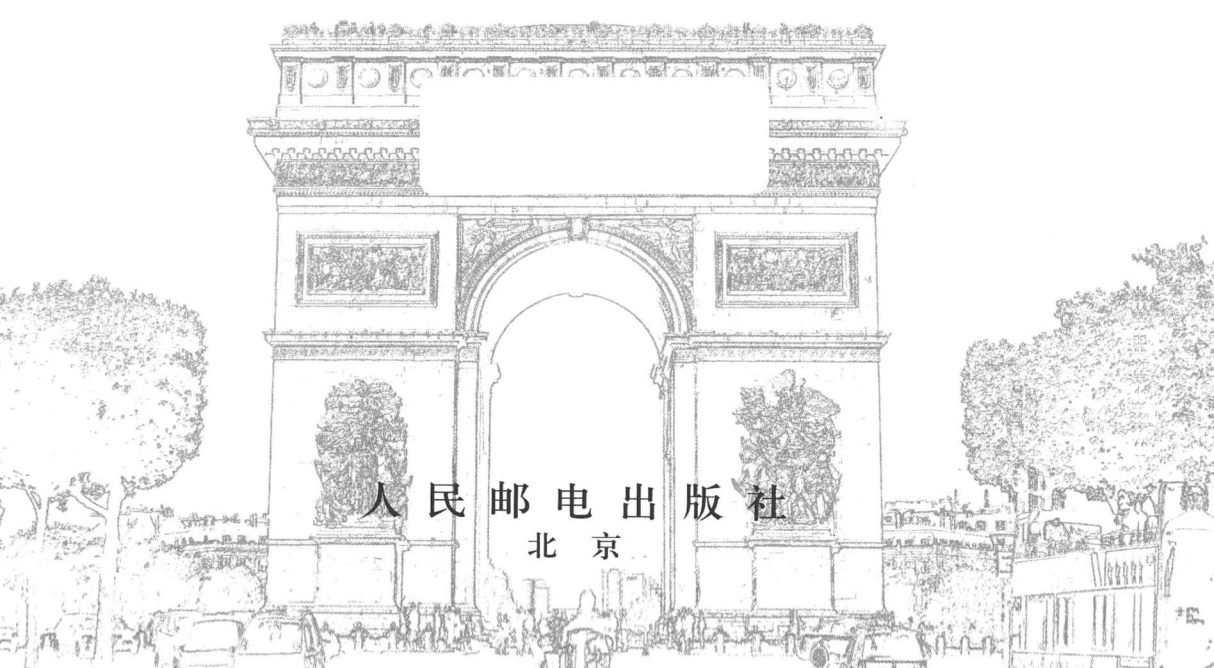
人民邮电出版社
POSTS & TELECOM PRESS

PEARSON

UNIX

编程环境 (英文版)

[美] Brian W. Kernighan Rob Pike 著



人民邮电出版社
北京

图书在版编目 (C I P) 数据

UNIX编程环境 : 英文 / (美) 克尼汉
(Kernighan, B. W.), (美) 派克 (Pike, R.) 著. -- 北京
: 人民邮电出版社, 2013. 2
ISBN 978-7-115-30243-4

I. ①U… II. ①克… ②派… III. ①
UNIX操作系统—程序设计—英文 IV. ①TP316.81

中国版本图书馆CIP数据核字 (2012) 第309153号

版权声明

Original edition, The Unix Programming Environment, 9780139376818, by Brian W. Kernighan, Rob Pike, published by Pearson Education, Inc., publishing as Prentice-Hall, Copyright © 1984.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Inc.

English reprint published by Pearson Education North Asia Limited and Posts & Telecommunication Press, Copyright © 2013.

This edition is manufactured in the People's Republic of China, and is authorized for sale and distribution in the People's Republic of China exclusively (except Taiwan, Hong Kong SAR and Macau SAR).

本书封面贴有 Pearson Education 出版集团激光防伪标签, 无标签者不得销售。

UNIX 编程环境 (英文版)

- ◆ 著 [美] Brian W. Kernighan, Rob Pike
责任编辑 汪 振
- ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京铭成印刷有限公司印刷
- ◆ 开本: 700×1000 1/16
印张: 23
字数: 493 千字
印数: 1—2 500 册
- 2013 年 2 月第 1 版
2013 年 2 月北京第 1 次印刷

著作权合同登记号 图字: 01-2012-8862 号

ISBN 978-7-115-30243-4

定价: 59.00 元

读者服务热线: (010)67132692 印装质量热线: (010)67129223
反盗版热线: (010)67171154

内容提要

本书系 UNIX 编程领域内的经典畅销书。作者本着“授之以渔”的态度，不仅向读者介绍了 UNIX 系统编程的基本技巧及编程规范，更是将 UNIX 的编程哲学融入其中，以帮助读者更加深刻地理解 UNIX 系统。本书的主要内容包括 UNIX 系统基本操作、文件系统、Shell 编程、过滤器、标准 I/O 库编程、系统调用、程序开发工具、文档准备工具等。

本书适合 UNIX 系统的初学者以及 UNIX 系统编程的爱好者阅读。

前言

“UNIX 系统的安装数量已经增长到了 10 个，预期还会有更多。”

《UNIX 程序员手册（第 2 版）》，1972 年 6 月

UNIX 操作系统¹始于 1969 年贝尔实验室的一台废弃的 DEC PDP-7 计算机。Ken Thompson 在 Rudd Canaday、Doug McIlroy、Joe Ossanna 和 Dennis Ritchie 的群策群力之下编写了一个小型的通用分时系统，这一系统的丰富内容足以吸引热心的用户，并且最终具备了在更高级的机器（一台 PDP-11/20）上运行所需的可靠性。Ritchie 是早期的用户之一，1970 年，他帮助将这个系统移植到一台 PDP-11 上。他还设计并编写了 C 语言编译器。1973 年，Ritchie 和 Thompson 用 C 语言重新编写了 UNIX 内核，打破了用汇编语言编写系统软件的传统。这次改写之后，UNIX 系统逐渐成为了今天的样子。

1974 年前后，UNIX 被授权给各个大学，用于“教育用途”，几年之后开始用于商业用途。在这段时间里，UNIX 系统在贝尔实验室中取得了成功，进入了实验室、软件开发项目、字处理中心和电话公司的运营支持系统。从此，它传播到全球各个角落，安装了成千上万次，涵盖了从微机到大型主机的各类机型。

是什么使 UNIX 系统如此成功？我们能够发现多种原因。首先，它使用 C 语言编写，易于移植，UNIX 系统可以运行在从微机到大型主机的各类机器上，这是强大的商业优势；其次，源代码使用高级语言编写，使得该系统很容易适应特别的需求；最后也是最重要的一点，它是一个好操作系统，对程序员尤其如此，UNIX 编程环境也是非常丰富且具有效率的。

尽管 UNIX 系统引入了许多创造性的程序和技术，但是它的成功并不是因为某个程序或者思路，相反，它的高效来源于一种编程的方法，一种使用计算机的哲学。

1 UNIX 是贝尔实验室的一个商标。“UNIX”不是一个缩写词，而是从 Thompson 与 Ritchie 从 UNIX 之前开发的操作系统 MULTICS 派生的双关语。

虽然这种哲学不是用一两句话就可以说明白的，但它的核心可以理解为：系统的能力更多的是源于程序之间的关系，而非程序本身。许多 UNIX 程序本身只完成很简单的任务，但是与其他程序相结合，就成为通用的实用工具。

本书旨在传播 UNIX 编程哲学。因为这种哲学基于程序之间的关系，所以我们必须将大部分篇幅用于讨论各个工具，但是自始至终贯穿着组合程序以及利用程序来构建程序的主题。为了很好地使用 UNIX 系统及其组件，不仅必须理解如何使用程序，还要理解如何让它们适应环境。

随着 UNIX 系统的扩展，精于其应用程序的用户已经越来越少。很多时候，我们都会发现，即使是经验丰富的用户（甚至包括作者自己），也只能找到一些笨拙的解决方案，或者亲自编写一段程序，去完成现有工具可以轻松完成的任务。当然，没有一定的经验和理解，很难发现精巧的解决方案。我们希望通过阅读本书，不管你是新用户还是老用户，都能够拓展自己的理解，有效地使用这个系统，并乐在其中。我们希望你真正用好 UNIX。

本书的目标读者是个体的程序员，希望通过提高他们的工作效率来使工作团队具有更高的生产率。尽管我们的目标读者是程序员，但阅读前四五章并不需要编程经验，所以这部分内容对其他用户也是有帮助的。

只要有可能，我们会尽量地用真实的例子而非人为的例子去阐明我们的观点。其中有些程序示例实际上已经成为了我们日常程序的一部分。本书的所有示例均已在文本中进行过测试，确保在电脑中可读。

本书的组织如下。第 1 章是对系统基本操作的简单介绍，包括登录、邮件、文件系统、常用命令和命令解释程序的基本原理，有经验的用户可以跳过这一章。

第 2 章讨论 UNIX 文件系统。文件系统是系统操作和使用的核心，所以为了更好地使用系统，你必须了解它。本章描述了文件和目录、权限和文件模式以及索引节点。本章以文件系统层次结构和设备文件的解释作为结束。

第 3 章描述了如何使用命令解释程序（也称 Shell），主要包括创建新命令、命令参数、Shell 变量、基本控制流和输入/输出重定向。Shell 是一个基础工具，不仅用于运行程序，也可用于编写程序。

第 4 章主要介绍过滤器，一种对数据流进行简单转换的程序。4.1 节介绍 `grep` 模式搜索命令及其相关内容；4.2 节讨论几个更常见的过滤器，如 `sort`；剩下的几节专门介绍 `sed` 和 `awk` 这两个通用的数据转换程序。`sed` 是一个流编辑器，可以对经过的数据流进行编辑。`awk` 是一种用于简单信息检索和报告生成的编程语言。使用这些程序（有时与 Shell 协作），往往可以完全避免常规编程。

第 5 章讨论如何使用 Shell 编写出可供其他人使用的程序。主题包括更高级的控制流和变量、陷阱和中断处理。这一章的例子很大程度上利用了 `sed`、`awk` 和 Shell。

前言

人们最终会达到 Shell 和现有程序的极限。第 6 章讨论如何使用标准的 I/O 库编写新程序。这些程序用 C 语言编写，这里假定读者已经了解，或者正在学习这种语言。这一章主要内容包括为新程序的设计和制定明智的策略、将编程划分为若干可控的阶段以及使用现有工具的方法。

第 7 章介绍系统调用，这是所有其他软件层次的基础。主题包括输入输出、文件创建、错误处理、目录、索引节点、进程和信号。

第 8 章讨论程序开发工具：**yacc** 是一个语法分析生成器，**make** 控制大程序的编译过程，**lex** 生成词法分析器。全章将会以一个真实的程序开发案例（一个用类 C 语言编程的计算器）为基础进行讲解。

第 9 章讨论文档准备工具，用为第 8 章示例所准备的用户级说明和操作指南页面阐述这些工具。这一章可以独立于其他章节阅读。

附录 1 总结了标准编辑器 **ed**。虽然许多读者在日常使用中可能会偏爱其他编辑器，但是 **ed** 是通用而高效的。它的正则表达式是 **grep** 和 **sed** 等程序的核心，仅仅这个理由就值得一读。

附录 2 包含了第 8 章计算器语言的参考手册。

附录 3 是计算器程序最终版本的清单，在一个地方列出所有代码，便于阅读。

要注意一些实践中的问题。首先，UNIX 系统已经非常流行，有许多广泛使用的版本。例如，第 7 版来自贝尔实验室计算科学研究中心原创的源代码。**System III** 和 **System V** 是贝尔实验室官方支持的版本。加州大学伯克利分校发布的系统继承自第 7 版，通常被称为 **UCB 4.x BSD**。此外，还有许多从第 7 版派生的变种，特别是用在小型计算机的版本。

我们尽量使用在各种版本中相同的特征，来应对这种多样性的问题。尽管我们希望教授的是独立于任何特定版本的知识，但是对于具体的细节，我们选择使用第 7 版的方式来表现，因为它构成了大部分广泛使用的 UNIX 系统的基础。我们还在贝尔实验室的 **System V** 和 **Berkeley 4.1 BSD** 上运行了书中的例子。不管你的机器使用的是什么版本，都会发现差异很小。

其次，虽然本书的内容很多，但是它不是一本参考手册。我们觉得更重要的是传授一种方法和一种使用风格，而不只是细节。《UNIX 程序员手册》是信息的标准来源。你需要用它来了解我们没有介绍的内容，或者用它来确定你用的系统与其他系统的差异。

最后，我们相信最好的学习方法就是实践。本书应该在终端上阅读，这样你就可以试验、验证或者反驳我们所说的，探索极限和变化。读一小段，试一下，然后再阅读新的内容。

我们相信，UNIX 系统尽管并不完美，但仍是一个杰出的计算环境，我们希望本

书能够帮助你理解这其中的奥秘。

我们要感谢曾对本书提出过建设性意见和评论的人们，也要感谢他们为改进我们的代码所做出的贡献。特别要感谢 Jon Bentley、John Linderman、Doug McIlroy 和 Peter Weinberger 认真地阅读本书的书稿的多个版本。感谢 Al Aho、Ed Bradford、Bob Flanrena、Dave Hanson、Ron Hardin、Marion Harris、Gerard Holzmann、Steve Johnson、Nico Lomuto、Bob Martin、Larry Rosler、Chris Van Wyk 和 Jim Weythman 为第一稿提出意见。我们还要感谢 Mike Bianchi、Elizabeth Bimmler、Joe Carfagno、Don Carter、Tom De Marco、Tom Duff、David Gay、Steve Mahaney、Ron Pinter、Dennis Ritchie、Ed Sitar、Ken Thompson、Mike Tilson、Paul Tukey 和 Larry Wehr 提出的宝贵建议。

Brian Kernighan

Rob Pike

CONTENTS

1.	UNIX for Beginners	1
1.1	Getting started	2
1.2	Day-to-day use: files and common commands	11
1.3	More about files: directories	21
1.4	The shell	26
1.5	The rest of the UNIX system	38
2.	The File System	41
2.1	The basics of files	41
2.2	What's in a file?	46
2.3	Directories and filenames	48
2.4	Permissions	52
2.5	Inodes	57
2.6	The directory hierarchy	63
2.7	Devices	65
3.	Using the Shell	71
3.1	Command line structure	71
3.2	Metacharacters	74
3.3	Creating new commands	80
3.4	Command arguments and parameters	82
3.5	Program output as arguments	86
3.6	Shell variables	88
3.7	More on I/O redirection	92
3.8	Looping in shell programs	94
3.9	bundle: putting it all together	97
3.10	Why a programmable shell?	99
4.	Filters	101
4.1	The <code>grep</code> family	102
4.2	Other filters	106

4.3	The stream editor <code>sed</code>	108
4.4	The <code>awk</code> pattern scanning and processing language	114
4.5	Good files and good filters	130
5.	Shell Programming	133
5.1	Customizing the <code>cal</code> command	133
5.2	Which command is which?	138
5.3	<code>while</code> and <code>until</code> loops: watching for things	144
5.4	Traps: catching interrupts	150
5.5	Replacing a file: <code>overwrite</code>	152
5.6	<code>zap</code> : killing processes by name	156
5.7	The <code>pick</code> command: blanks vs. arguments	159
5.8	The <code>news</code> command: community service messages	162
5.9	<code>get</code> and <code>put</code> : tracking file changes	165
5.10	A look back	169
6.	Programming with Standard I/O	171
6.1	Standard input and output: <code>vis</code>	172
6.2	Program arguments: <code>vis</code> version 2	174
6.3	File access: <code>vis</code> version 3	176
6.4	A screen-at-a-time printer: <code>p</code>	180
6.5	An example: <code>pick</code>	186
6.6	On bugs and debugging	187
6.7	An example: <code>zap</code>	190
6.8	An interactive file comparison program: <code>idiff</code>	192
6.9	Accessing the environment	199
7.	UNIX System Calls	201
7.1	Low-level I/O	201
7.2	File system: directories	208
7.3	File system: inodes	214
7.4	Processes	220
7.5	Signals and interrupts	225
8.	Program Development	233
8.1	Stage 1: A four-function calculator	234
8.2	Stage 2: Variables and error recovery	242
8.3	Stage 3: Arbitrary variable names; built-in functions	245
8.4	Stage 4: Compilation into a machine	258
8.5	Stage 5: Control flow and relational operators	266
8.6	Stage 6: Functions and procedures; input/output	273
8.7	Performance evaluation	284
8.8	A look back	286

CONTENTS

9.	Document Preparation	289
9.1	The <code>ms</code> macro package	290
9.2	The <code>troff</code> level	297
9.3	The <code>tbl</code> and <code>eqn</code> preprocessors	301
9.4	The manual page	308
9.5	Other document preparation tools	313
10.	Epilog	315
	Appendix 1: Editor Summary	319
	Appendix 2: <code>hoc</code> Manual	329
	Appendix 3: <code>hoc</code> Listing	335
	Index	349

CHAPTER 1: **UNIX FOR BEGINNERS**

What is “UNIX”? In the narrowest sense, it is a time-sharing operating system *kernel*: a program that controls the resources of a computer and allocates them among its users. It lets users run their programs; it controls the peripheral devices (discs, terminals, printers, and the like) connected to the machine; and it provides a file system that manages the long-term storage of information such as programs, data, and documents.

In a broader sense, “UNIX” is often taken to include not only the kernel, but also essential programs like compilers, editors, command languages, programs for copying and printing files, and so on.

Still more broadly, “UNIX” may even include programs developed by you or other users to be run on your system, such as tools for document preparation, routines for statistical analysis, and graphics packages.

Which of these uses of the name “UNIX” is correct depends on which level of the system you are considering. When we use “UNIX” in the rest of this book, context should indicate which meaning is implied.

The UNIX system sometimes looks more difficult than it is — it’s hard for a newcomer to know how to make the best use of the facilities available. But fortunately it’s not hard to get started — knowledge of only a few programs should get you off the ground. This chapter is meant to help you to start using the system as quickly as possible. It’s an overview, not a manual; we’ll cover most of the material again in more detail in later chapters. We’ll talk about these major areas:

- basics — logging in and out, simple commands, correcting typing mistakes, mail, inter-terminal communication.
- day-to-day use — files and the file system, printing files, directories, commonly-used commands.
- the command interpreter or *shell* — filename shorthands, redirecting input and output, pipes, setting erase and kill characters, and defining your own search path for commands.

If you’ve used a UNIX system before, most of this chapter should be familiar; you might want to skip straight to Chapter 2.

You will need a copy of the *UNIX Programmer's Manual*, even as you read this chapter; it's often easier for us to tell you to read about something in the manual than to repeat its contents here. This book is not supposed to replace it, but to show you how to make best use of the commands described in it. Furthermore, there may be differences between what we say here and what is true on your system. The manual has a permuted index at the beginning that's indispensable for finding the right programs to apply to a problem; learn to use it.

Finally, a word of advice: don't be afraid to experiment. If you are a beginner, there are very few accidental things you can do to hurt yourself or other users. So learn how things work by trying them. This is a long chapter, and the best way to read it is a few pages at a time, trying things out as you go.

1.1 Getting started

Some prerequisites about terminals and typing

To avoid explaining everything about using computers, we must assume you have some familiarity with computer terminals and how to use them. If any of the following statements are mystifying, you should ask a local expert for help.

The UNIX system is *full duplex*: the characters you type on the keyboard are sent to the system, which sends them back to the terminal to be printed on the screen. Normally, this *echo* process copies the characters directly to the screen, so you can see what you are typing, but sometimes, such as when you are typing a secret password, the echo is turned off so the characters do not appear on the screen.

Most of the keyboard characters are ordinary printing characters with no special significance, but a few tell the computer how to interpret your typing. By far the most important of these is the RETURN key. The RETURN key signifies the end of a line of input; the system echoes it by moving the terminal's cursor to the beginning of the next line on the screen. RETURN must be pressed before the system will interpret the characters you have typed.

RETURN is an example of a *control character* — an invisible character that controls some aspect of input and output on the terminal. On any reasonable terminal, RETURN has a key of its own, but most control characters do not. Instead, they must be typed by holding down the CONTROL key, sometimes called CTL or CNTL or CTRL, then pressing another key, usually a letter. For example, RETURN may be typed by pressing the RETURN key or, equivalently, holding down the CONTROL key and typing an 'm'. RETURN might therefore be called a control-m, which we will write as *ctl-m*. Other control characters include *ctl-d*, which tells a program that there is no more input; *ctl-g*, which rings the bell on the terminal; *ctl-h*, often called backspace, which can be used to correct typing mistakes; and *ctl-i*, often called tab, which

advances the cursor to the next tab stop, much as on a regular typewriter. Tab stops on UNIX systems are eight spaces apart. Both the backspace and tab characters have their own keys on most terminals.

Two other keys have special meaning: DELETE, sometimes called RUBOUT or some abbreviation, and BREAK, sometimes called INTERRUPT. On most UNIX systems, the DELETE key stops a program immediately, without waiting for it to finish. On some systems, *ctl-c* provides this service. And on some systems, depending on how the terminals are connected, BREAK is a synonym for DELETE or *ctl-c*.

A Session with UNIX

Let's begin with an annotated dialog between you and your UNIX system. Throughout the examples in this book, what you type is printed in *slanted letters*, computer responses are in *typewriter-style characters*, and explanations are in *italics*.

```

Establish a connection: dial a phone or turn on a switch as necessary.
Your system should say
login: you                Type your name, then press RETURN
Password:                Your password won't be echoed as you type it
You have mail.           There's mail to be read after you log in
$                         The system is now ready for your commands
$                         Press RETURN a couple of times
$ date                   What's the date and time?
Sun Sep 25 23:02:57 EDT 1983
$ who                    Who's using the machine?
jlb      tty0      Sep 25 13:59
you      tty2      Sep 25 23:01
mary     tty4      Sep 25 19:03
doug     tty5      Sep 25 19:22
egb      tty7      Sep 25 17:17
bob      tty8      Sep 25 20:48
$ mail                    Read your mail
From doug Sun Sep 25 20:53 EDT 1983
give me a call sometime monday

?                         RETURN moves on to the next message
From mary Sun Sep 25 19:07 EDT 1983    Next message
Lunch at noon tomorrow?

? d                        Delete this message
$                          No more mail
$ mail mary               Send mail to mary
lunch at 12 is fine
ctl-d                     End of mail
$                          Hang up phone or turn off terminal
                           and that's the end

```

Sometimes that's all there is to a session, though occasionally people do

some work too. The rest of this section will discuss the session above, plus other programs that make it possible to do useful things.

Logging in

You must have a login name and password, which you can get from your system administrator. The UNIX system is capable of dealing with a wide variety of terminals, but it is strongly oriented towards devices with *lower case*; case distinctions matter! If your terminal produces only upper case (like some video and portable terminals), life will be so difficult that you should look for another terminal.

Be sure the switches are set appropriately on your device: upper and lower case, full duplex, and any other settings that local experts advise, such as the speed, or *baud rate*. Establish a connection using whatever magic is needed for your terminal; this may involve dialing a telephone or merely flipping a switch. In either case, the system should type

```
login:
```

If it types garbage, you may be at the wrong speed; check the speed setting and other switches. If that fails, press the BREAK or INTERRUPT key a few times, slowly. If nothing produces a login message, you will have to get help.

When you get the login: message, type your login name *in lower case*. Follow it by pressing RETURN. If a password is required, you will be asked for it, and printing will be turned off while you type it.

The culmination of your login efforts is a *prompt*, usually a single character, indicating that the system is ready to accept commands from you. The prompt is most likely to be a dollar sign \$ or a percent sign %, but you can change it to anything you like; we'll show you how a little later. The prompt is actually printed by a program called the *command interpreter* or *shell*, which is your main interface to the system.

There may be a message of the day just before the prompt, or a notification that you have mail. You may also be asked what kind of terminal you are using; your answer helps the system to use any special properties the terminal might have.

Typing commands

Once you receive the prompt, you can type *commands*, which are requests that the system do something. We will use *program* as a synonym for command. When you see the prompt (let's assume it's \$), type *date* and press RETURN. The system should reply with the date and time, then print another prompt, so the whole transaction will look like this on your terminal:

```
$ date
Mon Sep 26 12:20:57 EDT 1983
$
```

Don't forget RETURN, and don't type the \$. If you think you're being

ignored, press RETURN; something should happen. RETURN won't be mentioned again, but you need it at the end of every line.

The next command to try is `who`, which tells you everyone who is currently logged in:

```
$ who
rlm      tty0      Sep 26 11:17
pjlw     tty4      Sep 26 11:30
gerard   tty7      Sep 26 10:27
mark     tty9      Sep 26 07:59
you      ttya      Sep 26 12:20
$
```

The first column is the user name. The second is the system's name for the connection being used ("tty" stands for "teletype," an archaic synonym for "terminal"). The rest tells when the user logged on. You might also try

```
$ who am i
you      ttya      Sep 26 12:20
$
```

If you make a mistake typing the name of a command, and refer to a non-existent command, you will be told that no command of that name can be found:

```
$ whom
whom: not found
$
```

*Misspelled command name ...
... so system didn't know how to run it*

Of course, if you inadvertently type the name of an actual command, it will run, perhaps with mysterious results.

Strange terminal behavior

Sometimes your terminal will act strangely, for example, each letter may be typed twice, or RETURN may not put the cursor at the first column of the next line. You can usually fix this by turning the terminal off and on, or by logging out and logging back in. Or you can read the description of the command `stty` ("set terminal options") in Section 1 of the manual. To get intelligent treatment of tab characters if your terminal doesn't have tabs, type the command

```
$ stty -tabs
```

and the system will convert tabs into the right number of spaces. If your terminal does have computer-settable tab stops, the command `tabs` will set them correctly for you. (You may actually have to say

```
$ tabs terminal-type
```

to make it work — see the `tabs` command description in the manual.)

Mistakes in typing

If you make a typing mistake, and see it before you have pressed RETURN, there are two ways to recover: *erase* characters one at a time or *kill* the whole line and re-type it.

If you type the *line kill* character, by default an at-sign @, it causes the whole line to be discarded, just as if you'd never typed it, and starts you over on a new line:

```
$ ddtae@                               Completely botched; start over
date                                   on a new line
Mon Sep 26 12:23:39 EDT 1983
$
```

The sharp character # erases the last character typed; each # erases one more character, back to the beginning of the line (but not beyond). So if you type badly, you can correct as you go:

```
$ dd#atte##e                           Fix it as you go
Mon Sep 26 12:24:02 EDT 1983
$
```

The particular erase and line kill characters are *very* system dependent. On many systems (including the one we use), the erase character has been changed to backspace, which works nicely on video terminals. You can quickly check which is the case on your system:

```
$ datee+                               Try +
datee+: not found                       It's not +
$ datee#                               Try #
Mon Sep 26 12:26:08 EDT 1983           It is #
$
```

(We printed the backspace as + so you can see it.) Another common choice is *ctl-u* for line kill.

We will use the sharp as the erase character for the rest of this section because it's visible, but make the mental adjustment if your system is different. Later on, in "tailoring the environment," we will tell you how to set the erase and line kill characters to whatever you like, once and for all.

What if you must enter an erase or line kill character as part of the text? If you precede either # or @ by a backslash \, it loses its special meaning. So to enter a # or @, type \# or \@. The system may advance the terminal's cursor to the next line after your @, even if it was preceded by a backslash. Don't worry — the at-sign has been recorded.

The backslash, sometimes called the *escape character*, is used extensively to indicate that the following character is in some way special. To erase a backslash, you have to type two erase characters: \##. Do you see why?

The characters you type are examined and interpreted by a sequence of programs before they reach their destination, and exactly how they are interpreted