# A System V Guide to UNIX and XENIX

# Trademarks

UNIX is a trademark of AT&T Bell Laboratories.
XENIX, MS-DOS, and PC-DOS are registered trademarks of Microsoft Corporation.
PDP-7, PDP-11, VAX, VT52, and VT100 are trademarks of Digital Equipment Corporation.
GE 645 is a trademark of General Electric Corporation.
Idris is a trademark of Whitesmith's Ltd.
Cromix is a trademark of Cromemco, Inc.
PC/IX is a trademark of Interactive Systems, Inc.
CP/M is a registered trademark of Digital Research, Inc.
Ethernet is a trademark of Digital Equipment, Intel, and Xerox Corporations.
IBM is a registered trademark of International Business Machines, Inc.
EBCDIC, BSC, SNA, SDLC, DOS, DOS/VS, DOS/VSE, OS, OS/MFT, OS/MVT, MVS, VM,
CICS, TSO, IMS, and CMS are trademarks of International Business Machines, Inc.
Teletype is a trademark of Teletype Corporation.
NEC and Spinwriter are trademarks of Nippon Electric Company.
COHERENT is a registered trademark of Mark Williams Company.
LSI, ADM-3, and ADM-3A are trademarks of Zentec Corporation.
HP and 2631 are trademarks of Hewlett-Packard.
Wyse 50 is a trademark of Wyse Technology, Inc.
Amdahl is a registered trademark, UTS is a trademark of Amdahl Corporation.

Douglas William Topham

# A System V Guide to UNIX and XENIX

With 61 Figures

Springer-Verlag
New York Berlin Heidelberg
London Paris Tokyo Hong Kong

Library of Congress Cataloging-in-Publication Data
Topham, Douglas W., 1942–
A system V guide to UNIX and XENIX/Douglas William Topham.
p. cm.
ISBN 0-387-97021-5 (alk. paper)
1. UNIX System V (Computer operating system) 2. XENIX (Computer operating system) I. Title.
QA76.76.063T667 1990
005.4'3—dc20 90-32803

Printed on acid-free paper.

Photocomposed copy prepared using LaTeX.

9 8 7 6 5 4 3 2 1

ISBN-13: 978-0-387-97021-9 e-ISBN-13: 978-1-4612-3246-9
DOI: 10.1007_978-1-4612-3246-9

# Preface

As in preparing the earlier edition of this book, we've emphasized presenting UNIX in basic terms for first-time users. Parts I–IV of the book ("Fundamentals," "Text-Editing," "Text-Processing," and "Text-Formatting") are for most readers. If you're interested in just a brief introduction to UNIX, see the short course at the end of this introduction.

For more experienced users, Parts V ("Shell Programming"), VI ("System Administration"), and VII ("Network Administration") delve a little more deeply into the system, and provide information that is either scattered throughout many documents or unavailable. Part V begins with some shortcuts that most readers can use, then covers programming techniques in detail. Part VI presents the concepts that underly the internal operation of UNIX, along with step-by-step procedures for operating and maintaining a system.

## Acknowledgements

I'd like to thank Hai Truong for contributing the chapters on shell programming; Tom Leslie of AT&T and Brigid Fuller and Bill Brothers of the Santa Cruz Operation for help with new features; and Bill Potts, John Sovereign, Chris Swartout, Jim Edele, Danesh Forouhari, Andrew Sharpe, Tom Leslie, and Steve Robertson for reviewing certain key chapters for accuracy.

## Why You Need a Step-by-Step Guide

This book leads you step-by-step through the UNIX operating system. If you are new to UNIX, you need a book like this. Using UNIX after using one of the earlier operating systems for microcomputers is like using a single-lens reflex camera after using a simple aim-and-shoot camera. There are more features than you know what to do with, and it's very difficult to figure out where to begin.

Even though the reference manuals for UNIX seem to come in truckloads, they often leave out essential information. You read page after page about a feature you have to use, only to find that the basic facts you need have been left out. The writers assume that you already know what the feature is for and that you know how to use it. The result is text that is COIAK (clear only if already known).

This book starts at the beginning and explains the basics step by step. The fundamental facts are given to you, not assumed. The background explanations and short, simple examples will help you learn UNIX and XENIX quickly. While this approach is intended mainly for non-technical people, many technical professionals have also profited from reading this book. Any reader will finish this book well prepared to move on to more complex topics.

## Organization of This Book

The main text of this book is divided into six parts, each more or less independent of the others. Within each part, the discussion begins with simpler topics, then builds to more involved concepts in subsequent chapters. We hope this makes it a little easier for you to select the material you need and get the most out of this book.

I. *Fundamentals.* These chapters, intended for all readers, give you basic instruction in using UNIX for the first time and finding out how it works.

II. *Text-Editing.* These chapters show you how to use the **vi** and **ex** programs to enter and edit text.

III. *Text-Processing.* These chapters show you how to perform searching, sorting, and programming to process text and numbers.

IV. *Text-Formatting.* These chapters show you how to format and otherwise process text files, usually in preparation for printing.

V. *Shell Programming.* These chapters, intended for more experienced users, explain some short-cuts and provide you with the tools for designing your own procedures and customizing your UNIX system.

VI. *System Administration.* These chapters, also intended for experienced users, describe some of the inner workings of UNIX and show you procedures for taking care of disks and tapes, backup and recovery, startup and shutdown, terminals and printers, system security, and system accounting.

VII. *Network Administration.* These chapters continue the discussion of administration with an emphasis on networking applications, such as communication and resource-sharing between different machines.

The appendices summarize information presented in the main text and provide technical reference material. The glossary, which can be an education in itself, explains the technical terms that you'll need to know.

We have done everything we can to ensure that each topic has been introduced before being mentioned in connection with other topics. In nearly every instance we were able to do this. However, because of the way concepts are interrelated in UNIX, we faced many "chicken or egg" dilemmas in arranging the topics. As a result, there are a handful of instances where a topic is mentioned shortly before being described in full.

As a general rule, each section of the book is more technical than the previous one, and each section becomes more detailed as you progress from the beginning of the section to the end. Part I is for everyone; Parts II–IV are for those who will be working with text; Parts V–VII are mainly for those who will be modifying the operation of UNIX or maintaining a UNIX system.

## Shorter Course

Feel free to turn directly to the parts of the book that are of greatest interest to you. You can use it just for an introduction to UNIX, or you can delve into it and learn how to produce formatted documentation, program the shell, and perform administrative functions. To become quickly familiar with UNIX without wading into technical details, you can start out with the following chapters:

1 Introduction to UNIX
2 Getting Started with UNIX
3 The UNIX File System
4 Using UNIX Commands
5 Communication in UNIX
6 Introduction to **vi**
13 Searching and Sorting
16 Introduction to **mm**

Some readers may also wish to read the following chapters:
21 Introduction to the Bourne Shell
28 Basic Information (about System Administration)
37 Introduction to Networking

## Versions Covered

This book covers UNIX System V, Releases 1, 2, and 3, along with the corresponding releases of XENIX System V (XENIX 2.1 through 2.3). As this book was going to press, the Santa Cruz Operation had just released XENIX 2.3, and AT&T and the Santa Cruz Operation had just announced a new combined UNIX and XENIX, known as UNIX System V/386, Release 3.2. If you have any of these versions of UNIX or XENIX, you should find this book useful.

## Corrections

The author and the publisher will appreciate receiving any suggestions and corrections from readers.

# Typographic Conventions

In this book, typewriter is used for names for directories and files, and **bold typewriter** is used for commands. ***Slanted typewriter*** indicates what you are to type at the keyboard. Here are examples:

| | |
|---|---|
| /usr/lib | Directory name |
| /unix | File name |
| **date** | Command name |
| $ ***who*** | User entry |

*Italic* is used for descriptions of information that you are to enter. When you see italic, type what is described, not the actual characters, as shown in these examples:

| | |
|---|---|
| *name* | Type a name, not the letters n-a-m-e |
| *type* | Enter a type, not the letters t-y-p-e |

In some instances, italic is used to distinguish a command from information to be used by the command. In this example, italic separates "won" from the command name that precedes it:

| | |
|---|---|
| **cw***won* | The command is **cw**, while "won" is a word to be used by the command |

Some names are printed partly bold typewriter and partly italic. These names are part literal, part descriptive. Here is an example:

| | |
|---|---|
| **LCK.***name* | Here, **LCK** is a literal name, while *name* indicates a name that you are to enter |

Braces are used to indicate a list of items from which you must choose one. Here is an example from Chapter 34 that displays three choices (**-m** *model*, **-e** *printer*, and **-i** *custom*):

$$\texttt{lp lpadmin -p}\ \textit{printer}\ \texttt{-v}\ \textit{device} \left\{\begin{array}{l}\texttt{-m}\ \textit{model}\\ \texttt{-e}\ \textit{printer}\\ \texttt{-i}\ \textit{custom}\end{array}\right\}$$

Brackets are used to indicate optional items, either a single item or a list of items. The following example from Chapter 36 shows several optional items, some belonging to a list, others by themselves:

```
      ┌-a┐
$ sar │-b│  [-o  file]  t  [ n]
      └-c┘
```

In this example, three optional items are displayed in a list (**-a**, **-b**, and **-c**), while two other optional items are displayed by themselves (**-o** *file* and *n*). The only two things required here are **sar** and *t*; everything else is optional. Note also that **sar** is a command name, which you are to enter literally, while *t* is a generic name, which in this instance represents a length of time in seconds.

## About the Author

Douglas Topham grew up in Los Angeles in the San Fernando Valley, received his B.S. and A.M. degrees from Stanford University. After teaching math courses at the high school and college levels, he wrote a set of programs to provide live displays for ABC's "Password" show. Later he wrote the *WordStar Training Guide* and designed the screen displays for WordStar 3.0. He is now a consultant in the San Francisco Bay area. Other works by the author include *UNIX and XENIX: A Step-by-Step Guide*, *Using WordStar*, *WordStar in a Flash*, and *Introduction to WordPerfect.*

## Limits of Liability and Disclaimer of Warranty

The author and publisher of this book have used their best efforts in preparing this book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

# Contents

# Part I

# Fundamentals

In Part I you will learn some basic facts about UNIX. You will also learn how to begin working with UNIX, how to use its file system to organize your work, how to execute UNIX commands to perform daily tasks, and how to communicate with other UNIX users on either your own system or another system. Finally, you will learn about the basics of communication.

# 1

# Introduction to UNIX

## 1.1 Operating systems

### Operating systems in general

The main reason people use computers is to run application software, such as word processing, data base, spreadsheet, and accounting programs. An operating system provides programmers with a common environment within which to develop software for users. An operating system provides programmers with a simpler target to aim at than a computer system. The more computer systems the operating system runs on, the more computer systems the programmer can reach with software.

An operating system also provides users with a common environment within which to run their applications. The operating system furnishes certain utility programs that support the user and the applications. It can also offer a "friendly face" in the form of a simplified graphical representation on the screen. Finally, a more sophisticated operating system may provide additional conveniences: a way of allowing more than one person to use the computer at the same time (*time-sharing*), some means of communication between these different users, a way of allowing different programs to run at the same time (*multi-tasking*), tools for entering and processing text, programming tools to ease the task of software development, and various security measures.

### The UNIX system

Now that you have some general notion of what an operating system is, what distinguishes UNIX from other operating systems? Here are some of the most prominent features:

- Structured file system with multiple levels

- Ability to allow many users to work from the same computer at the same time (*multi-user*)

- Ability to allow any of the users active on the computer to run more than one program at a time (this is called *multi-tasking*)

- Mechanisms that allow one program to pass its results directly to another program, making it unnecessary to use extra storage space
- A scheme that allows any user to redirect the results of a program from one peripheral device to another (for example, from the video monitor to a disk drive)
- A built-in command interpreter and language (known as the *shell*)
- A structured language called **C** for systems programming
- Extensive tools for writing and developing programs in **C** and other programming languages
- Extensive tools for entering, changing, and processing written text and formatting it for printing
- Extensive tools for connecting computer systems (UNIX and non-UNIX)
- Practically limitless "modifiability"

## 1.2 UNIX operation

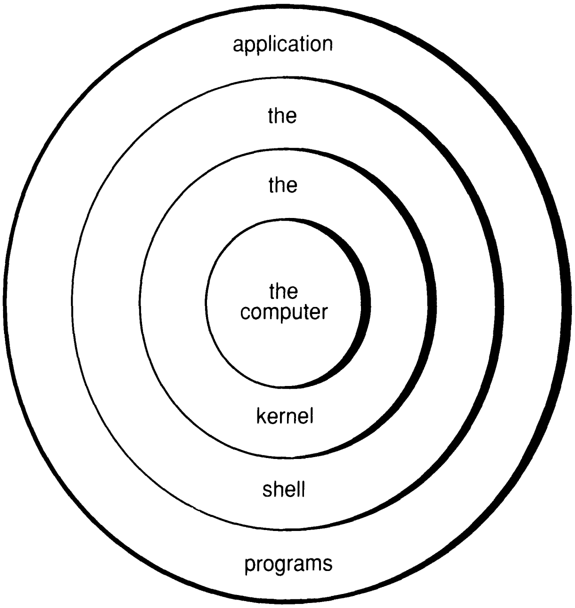
Whenever you run an application program under UNIX, three programs work together. The program that interacts directly with the computer is called the *kernel*. As an everyday user of UNIX, you will seldom be aware of the kernel's presence. You will be more aware of the program that interprets what you type at your keyboard and arranges for other programs to run—the *shell*. Strictly speaking, the shell is just another program. However, because the shell plays such an important role in interacting with users, it is customary to depict the shell as a middle layer between the kernel and applications, as shown in Figure 1.1.

### The kernel

You use an operating system by sitting at a keyboard in front of a screen and typing a command to perform a function. For example, you may enter a command that says to the operating system, "Let me begin an editing session" or "Let me print the text in this file." When you make such a request through a utility program, you cause a *process* to be activated.

The request is fulfilled when the process calls on the kernel to carry it out. The kernel, as its name implies, is the central core of the UNIX operating system. The kernel's routines schedule processes, route data to and from peripheral devices, manage memory resources, and maintain files in file systems.

FIGURE 1.1. The different parts of UNIX.



The utility programs that you use every day (and that you probably identify with UNIX) are not actually part of the operating system. These programs, which are described throughout this book, pass requests to the kernel through *system calls*, which are not discussed in this book.

Because the kernel interacts directly with the hardware, it is different for each computer. Since the kernel is only about 10 percent of the entire UNIX system, it is relatively easy to produce a new version of UNIX for a different machine. (A program that can be run on a variety of machines is said to be *portable*. Portability is discussed later in this chapter.)

## The shell

It is the shell that greets you when you begin a session with UNIX, and it is the shell that accepts instructions from you and carries them out. The shell is UNIX's command interpreter, a program that runs as a software layer over the kernel. The shell presents you with a prompt on the screen, followed by a cursor, like this:

```
$ _
```

When you type a command after this prompt, the shell begins a UNIX process. This process may be a simple routine to display today's date and time on the screen, or it may be a sophisticated text-editing system (the shell itself is also a process). You activate any process, large or small, by typing a command and letting the shell take care of carrying it out. If anything goes wrong while you are *invoking* the process, the shell will display an error message on your screen. Once a process has completed, the shell returns to display another prompt on your screen, indicating that it is ready for your next command.

Actually, there is more than one shell available for UNIX. First, there is the shell developed for UNIX Version 7 at Bell Telephone Laboratories by Stephen R. Bourne (known as the *Bourne shell*). Then there is another version of the shell that was developed at the University of California by William N. Joy in the late 1970s (known as the *C shell*). Finally, in the past few years, a third shell has been developed by David Korn (known as the *Korn shell*). The Korn shell combines the best features of the Bourne shell and the C shell.

## Application programs

Because of the increasing popularity of UNIX, the list of application programs available is growing every day. There are word processing programs, spreadsheet programs, data base management programs, compilers for BASIC, FORTRAN, COBOL, C, and other programming languages, and assemblers. Many of the most popular programs for microcomputers are being rewritten in C, so that they can be offered to users of UNIX. In addition, sophisticated applications are being moved from mainframes and minicomputers down to microcomputers running UNIX.

Before long, it will be possible to run most of the programs that became popular under CP/M, MS-DOS, and PC-DOS, while at the same time enjoying the versatility and power of UNIX. Until a year or so ago, microcomputers simply could not provide the extensive computing resources required by UNIX. They could not match the large amounts of internal memory and the fast, large-capacity disks of the minicomputers. But with recent technical advances in both areas, microcomputers have quickly become powerful enough to assume the rigors of running UNIX.

## C and portability

UNIX was originally written for the DEC PDP-7 minicomputer. Later it was modified to run on other minicomputers, and then finally on microcomputers. It is this *portability* that has contributed to the popularity of UNIX. One reason for the portability of UNIX is that most of it is written in the **C** language. Many of the early programs for microcomputers were written in low-level assembly language, close to machine code, to obtain maximum performance. Later on, applications programs were written in high-level languages like BASIC and Pascal to achieve portability. C is a sort of middle-level language, combining the high performance of a low-level language with the portability and ease of use of a high-level language.

UNIX was the first major operating system that was *not* written primarily in assembly language.

Originally associated closely with UNIX and part of the development of UNIX, C has recently become recognized as a useful product in its own right. Over a dozen versions of C are being sold to microcomputer owners.

## 1.3 UNIX and standards

Now that you've had a brief introduction to UNIX, we'll discuss a few other operating systems that are widely used today, and show how some of them resemble UNIX.

### Microcomputer operating systems

When microcomputers became popular in the late 1970s, the operating system that was most widely used was called *CP/M* (control program/monitor). Originally designed for Intel's 8080 microprocessor, later versions of CP/M were designed to work with Intel's derivative 8085 and Zilog's derivative Z80. Others also widely used were Apple's DOS and Tandy/Radio Shack's TRS-DOS. All of these operating systems were severely limited in storage space, or *memory*.

When IBM introduced its Personal Computer late in 1981, it also announced a new operating system, called *PC-DOS* (*MS-DOS* for non-IBM systems). The first version of PC-DOS was practically identical to CP/M, but it was designed for one of Intel's successors to the 8080, known as the 8088. The 8088, like its more powerful brother, the 8086, allowed much more memory than the 8080, 8085, and Z80. Recent versions of PC-DOS have been incorporating more and more UNIX features, such as subdirectories and redirection. Future versions will probably resemble UNIX even more strongly.

As the 1990s approach, nearly every operating system widely used on microcomputers is feeling the influence of the graphical interfaces first developed in the 1970s at XEROX's Palo Alto Research Center (PARC). These interfaces employ screen icons to represent computer functions and files, along with mouse operation. (A mouse is a hand-held device that you can move around on the top of your desk to make selections from the screen.) These concepts have been embodied in Apple's Macintosh operating system, IBM's new OS/2 (the successor to PC-DOS), and some of the newest releases of UNIX.

## Versions of UNIX

For the past few years, there have been two major variations of UNIX: the Berkeley System Distribution (BSD) from the University of California and System V from AT&T (see Figure 1.2). The Berkeley versions, with their emphasis on technical innovation, appeal more to institutions engaged in education, research, and engineering. The AT&T versions, with their emphasis on resource sharing, appeal more to business. One of the main hybrids was XENIX, the version that was developed by Microsoft and marketed by the Santa Cruz Operation for microcomputers.

Various efforts are currently under way to unify the versions into a single UNIX product. AT&T and Microsoft have already announced a new combined UNIX/XENIX product for 386 machines, called UNIX System V/386, Release 3.2. Early in 1988, AT&T and Sun Microsystem began work on a new version featuring a graphical interface called Open Look and conformance with the Institute of Electrical and Electronic Engineers (IEEE) portable operating system environment standard (POSIX). In response, IBM, Digital Equipment Corporation (DEC), Hewlett-Packard, and others formed the Open Software Foundation (OSF) to support a competing standard UNIX version.

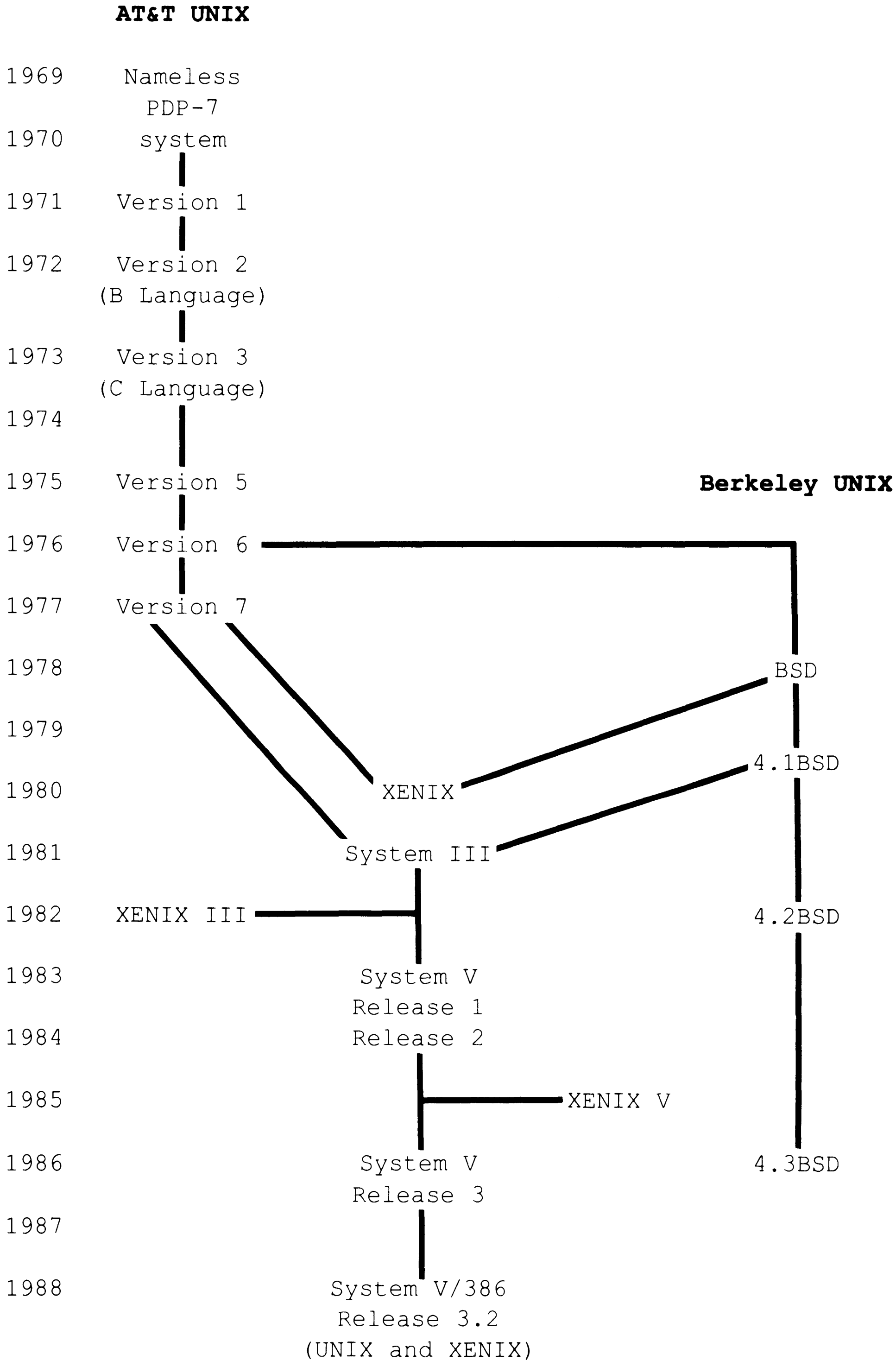
## UNIX System V

System V has been offered in various releases: Release 1.0 (1983), Release 2.0 (1984), Release 2.1 (1985), Release 3.0 (1986), Release 3.1 (1987), and Release 3.2 (1989). (The joint Sun/AT&T product will be known as Release 4.) The main features of Release 3, which relate to networking systems, sharing files, and system efficiency, are described later in this book. Part VII, "Network Administration," provides an introduction to Release 3 features and offers a discussion of Remote File Sharing for system administrators; and Appendix M, "UNIX versus XENIX," summarizes the features of System V, Release 3.

To give programmers a standard environment for developing software for UNIX systems running on computers of any size, AT&T now offers a two-volume *System V Interface Definition.* Copies of these and other UNIX documents are available from

AT&T Customer Information Center (CIC)   (800) 432-6600 (U.S.A.)
P.O. Box 19901   (800) 255-1242 (Canada)
Indianapolis, Indiana 46219   (317) 352-8557 (elsewhere)

FIGURE 1.2. The versions of UNIX and XENIX.



## UNIX DOCUMENTATION AND THIS BOOK

This book concentrates on general purpose programs, text-editing and formatting, one area of program development (shell programming), and administration and maintenance. There is also some basic information on configuring your terminal. C programming and software development are vast subjects, and are beyond the scope of this book. After reading this book, however, you should be well prepared to go on to books that cover these other areas.

For a comprehensive treatment of UNIX commands, you have to turn to one of the detailed reference works, such as the *UNIX User's Manual* from Bell Telephone Laboratories. This is now organized into separate volumes that cover specific areas, such as Program Development, Text Processing, and System Administration.

In printed form, a complete set costs about two hundred dollars. However, at many installations, a copy of the manual is included with UNIX on disk. Whenever we reach the limits of the scope of this book, we refer to this manual—by whatever name it may be known at your installation. Bell Telephone Laboratories is now called AT&T Bell Laboratories, and the manual is now called the *UNIX User's Manual.*

For quick reference, you can buy a compact 50-page command summary for under ten dollars from

Specialized Systems Consultants
P.O. Box 7, Northgate Station
Seattle, WA 98125-0007
(206) FOR-UNIX

or

A System Publications
P.O. Box 8681
Trenton, NJ 08650

# 2

# Getting Started with UNIX

In this chapter you will learn these basic things about UNIX:

- Preliminary set-up procedures
- Logging in and logging out
- Editing a command line
- Running processes (commands)
- Using the calculators
- Aids to learning

## 2.1 Preliminary set-up procedures

If you have been using a single-user operating system like CP/M, MS-DOS, or PC-DOS, then you are accustomed to simply turning on the computer, possibly entering the date and time, and going to work. However, with a multi-user system like UNIX, a number of different people can gain access to the computer at any given time from a number of different terminals. So before you can begin using UNIX, you have to make sure that UNIX recognizes you as a user on the system and that it can communicate with the terminal you are using.

### Identifying yourself to UNIX

You will be able to use UNIX only after your name has been added to a list of users. If you are working in a company or an institution, there is probably a system administrator designated to take care of new users. If so, this administrator will assign you an identifier, a password, and a terminal. The *identifier* assigned to you (also known as an *account name*, *login name*, or *user name*) will probably be either one of your names or your initials. The password is designed to prevent anyone else from using the system under your name. In most cases, the terminal will provide you with a keyboard for entering information and a video screen for receiving information back from the system.

### IDENTIFYING YOUR TERMINAL

If you have a version of UNIX that has been customized for your machine, then you probably won't have to do anything to identify your terminal. If not, you may have to make sure that the various settings for your terminal are compatible with the settings for your computer. Otherwise, your terminal and computer won't be able to communicate with each other. It's like making sure that you and another person both speak the same language before you begin a conversation.

Basically, you have to make sure that the terminal and computer are both sending information to each other at the same speed, that both are using the same convention for sending and receiving, that both are using the same scheme to check for errors, and that both interpret special characters the same way. To find out whether they have been set correctly, just log in and see what happens. If you see double characters on the screen (or no characters), random characters (also known as "garbage"), or no response to the (RETURN) key (Return or Enter), the settings for your terminal may be incorrect—get help from your system administrator. If you don't have any problems, the settings are fine. Later on, after you become more familiar with UNIX, you can learn to use a command called **stty** (set teletype) to make adjustments (see Chapter 33).

## 2.2 Logging in and logging out

### LOGGING IN

Once you have an identifier, a password, and a working terminal, you are ready to log in. Logging in is what gives you access to the UNIX operating system. Let's go through the basic procedures (which may vary from one system to another) shown here.

1. Establish connection between the computer and your terminal.

   □ This is the part that can vary the most. For your system, it may mean dialing a telephone number, flipping a switch on your terminal, or typing something at the keyboard. Find out from your system administrator exactly what is required.

   □ Once you have taken the necessary action, UNIX will first make itself known to you by a short, simple message in the upper left-hand corner of your screen like this:

   ```
   login: _
   ```

2. Identify yourself.

□ In the space after the "login" prompt, type your system identifier (or user name) and press (RETURN).

□ UNIX will probably respond by asking for your password:

```
login: robin
Password: _
```

3. Enter your secret password.

□ In the space after the "Password" prompt, type your password (which will not appear on the screen) and press (RETURN).

□ After a brief pause, you will see something like this:

```
login: robin
Password:

Last login: Tue Jan 15  08:17:26  on tty03

* * * * * * * * * * * * * * * * * * * * * * * * *
*                                               *
*  Welcome to  U N I X  System V  January 1990  *
*                                               *
* * * * * * * * * * * * * * * * * * * * * * * * *

$ _
```

Congratulations! You have just successfully logged in. The dollar sign ($), percent sign (%), or other symbol in front of the cursor is called the UNIX shell prompt. Later in the book, we'll explain why it has that name. We'll also show you how you can change it to something else. For now you can think of the UNIX shell prompt as similar to the prompt you see on the screen when you use CP/M, MS-DOS, or PC-DOS (A>). It tells you that UNIX is ready for you to type a command and go to work.

A system administrator can log in as a *super-user* to perform special tasks that require extraordinary privileges. Such a user will have a special prompt that looks like a pound sign (#).

## Typing commands

Just to verify that you have actually gained access to UNIX, try a few simple UNIX commands.

1. Let's begin with the command that tells you the day of the week, the date, and the time of day.

□ Type ***date*** and press (RETURN).

□ UNIX will respond with a display like this:

```
$ date
Mon Jan 17 09:02:37 EST 1990
$ _
```

2. Next, let's find out who is logged in right now.

□ Type ***who*** and press (RETURN).

□ UNIX will respond with a display like this:

```
$ who
janis     tty03  Jan 17  08:12
alex      tty05  Jan 17  08:39
jkl       tty07  Jan 16  21:16
robin     tty12  Jan 17  09:02
guapo     tty16  Jan 17  08:57
$ _
```

Each line of this display gives a user's identifier (like "`janis`"), a terminal number (like "`tty03`"), and the date and time the user logged in (like "`Jan 17 08:12`"). When UNIX was developed in the late 1960's, it was common to use a Teletype® hardcopy display (printing) device as a terminal. This is why UNIX says "tty" (from *t*ele*ty*pe) to mean "terminal."

3. Finally, let's type a command that UNIX doesn't know and see what happens.

□ Type ***why*** and press (RETURN).

□ UNIX will respond with a message like this:

```
$ why
why: Command not found.
$ _
```

## Logging out

When the time comes to end a session with UNIX, you can't just turn off your terminal. You have to *log out.* If you don't log out, UNIX will still consider you logged in, even though your terminal is disconnected.

For most systems, all you have to do is to type Control-D (hold the (CTRL) key down with one finger and press `D` with another). Since Control-D often means end-of-file or end-of-transmission to UNIX, there may be times when you will have to type Control-D several times to log yourself out. Some systems may require that you also press other keys. If you are using the C shell and the variable called "**ignoreeof**" is set, you have to type ***logout*** and press (RETURN). (You can also use **exit** in either shell.)

```
$                    [Press Control-D—nothing displayed]
login: _
```

When you have successfully logged out, the "login" prompt reappears on the screen. It isn't that UNIX doesn't like to see a user log out; it's just that UNIX doesn't have anything else to say. At this point you can either log in again or stay logged out. Since we have some more things to do, let's log back in and go on to the next section.

## Changing your password

Since your password gives you some measure of security under UNIX, you may want to change it from time to time. Changing it will make it more difficult for others to log in under your system name. For practice, try changing your password right now. Just type ***passwd*** and press (RETURN):

```
$ passwd
Changing password for robin
Old password: _
```

Type your current password in the space provided (you won't see it on the screen). Next, UNIX will ask you to type your new password (up to eight characters), then type the new password a second time to confirm it:

```
$ passwd
Changing password for robin
Old password:
New password:
Retype new password:
$ _
```

To make your password difficult for anyone to guess, you can mix upper and lower case letters with numbers and other characters. The more unusual (and longer) you make it, the harder it will be for another user to stumble across it by trial and error. It's usually best to avoid things like your nickname, your license plate number, your social security number, or your cat's name. Here are some examples of good passwords:

```
eASy-2.C
WheN?NoW
wHy_NoT?
New*4(U)
```

## 2.3 Entering a command line

Earlier in this chapter you used two simple commands, **date** and **who**. To carry out any action under UNIX, you have to execute a command (type a command line and press RETURN). In the case of the **date** command, the four letters **date** make up the entire command line. In the case of the **who** command, the three letters **who** make up an entire command line.

Although these two command lines required nothing more than the name of the command, many other commands require additional information about the command's options and any files to be processed by the command. Such information, if required, follows the command's name on the command line. For example, the **who** command can be executed by itself to list the users currently logged in. But there is also another form of the command:

```
$ who am i
robin     tty05     Jan 17  09:02
$ _
```

If you should forget your system identifier or if you have to log in under more than one identifier, you can use this form of the command to find out which identifier you are currently logged in under. The extra words added to the command (***am i***) form an *argument* to the command, which modifies the way the command works.

Until you press RETURN, it's still possible to make changes to a command line. The rest of this section discusses ways to make such changes.

### Erasing a character CTRL-H

To erase a single character on the command line, use # (number sign) or CTRL-H (hold down the CTRL key with one finger and press H with another). On some keyboards, there may be a key called BACKSPACE that performs this function. Here is how a typical correction might look in steps:

```
$ whoo_            [Extra o typed at the end of who]

$ who_             [Type #, CTRL-H, or BACKSPACE]
```

Now you can press the RETURN key to have the command line executed.

### Erasing an entire command line @

To erase the entire command line and start all over, use the at-sign (@) in UNIX or CTRL-U in XENIX. Again, there may be a different key for this function on your system. Assuming it's @, here is another typical correction:

```
$ fate_            [You typed fate instead of date]

$ fate             [Press @ to erase the line. In most cases, you won't
_                  actually see the line erased; the cursor will simply
                   drop down to the next line]

$ fate
date_              [Retype the line]
```

Now press the RETURN key to have the command line executed. In this simple example, pressing the @ key to erase the line instead of pressing CTRL-H four times to erase four characters separately saved you only three keystrokes. However, on a long command line, it would save you many more keystrokes.

### Restoring the prompt DEL

Sometimes, after you have started a process, the process hangs and the UNIX shell prompt ($) does not return to the screen. If this happens, press DEL to terminate the process and restore the prompt:

```
$ who
_                  [Press DEL—nothing will be displayed]
$ _
```

## 2.4 Process control

Once you have typed a command line correctly, there are several things you can do to control the resulting *process*: you can run the process in the background, request a list of processes currently running, abort a process, or halt screen output from a process.

### Running a process in the background &

Once you execute a command, the shell usually waits for the process to complete, then displays another shell prompt. This is called *foreground processing.* Unless you instruct UNIX otherwise, any process that you initiate will run in the *foreground* by default. That is, the process will tie up your terminal while it is running, making it impossible for you to do anything else with UNIX until the process is complete.

By instructing UNIX to run a process in the *background*, you can free your terminal and proceed to another task immediately. To run a process in the *background*, type an ampersand (**&**) at the end of the command line before pressing RETURN. (The & is SHIFT 7 on most keyboards.) *Background processing* is usually most suitable for commands that take a long time to execute. For example, to suspend processing for an hour (**sleep 3600**), type this:

```
$ sleep 3600 &
2167
$ _
```

UNIX will respond by displaying a process identification number (PID) and then reissuing another shell prompt. With the program running in the *background* and the shell prompt on the screen, you are now free to enter another command line (which may be another *background process*), without having to wait for the execution of **longtime** to be completed. While *background processing* is convenient for certain tasks, there are disadvantages to consider:

- A background process can't accept standard input.
- Any output from a background process to your screen will disrupt whatever you are typing at that moment.
- You have less control over a background process than you have over a foreground process .
- If you try to initiate too many background processes at once, you may run the risk of overloading your system.

## Finding out what processes are running **ps**

If you are running processes in the background, there is another UNIX command to find out which ones are still running at a particular moment and which have been completed. The **ps** (process status) command lists all processes currently running, displaying for each process its assigned process identification number (PID), the terminal on which it was initiated (TTY), the amount of time it has been running (TIME), and something to indicate the command line used to initiate it (COMMAND), as in this example:

```
$ ps
  PID  TTY  TIME COMMAND
 1905  12   1:16 -sh
 2132  12   2:18 ed
 2167  12   4:02 -sh
 2218  12   0:58 ps
$ _
```

### Aborting a process

DEL
**kill**

Sometimes it may be necessary to terminate a process before it has been completed. To abort a *foreground process*, press the DEL key (or CTRL-C). To abort a *background process*, use the **kill** command. Type **kill**, followed by the PID for the process you are terminating. For example, to abort the execution of **sleep 3600** described above, you could use

```
$ kill 2167
2167: terminated [UNIX responds with a message]
$ _
```

As long as you know the PID's, you can abort more than one process with a single command line:

```
$ kill 2132 2167
2132 2167: terminated
$ _
```

Some systems allow you to issue a **kill** to terminate *all* processes initiated from your terminal. However, since such a command can also log you out of the system, it's usually best not to try it—at least not now.

### Halting screen output

CTRL-S

At times you will find that a screen display is scrolling up the screen faster than you can read it. To halt the scrolling temporarily, press CTRL-S to make the display pause. Then, after you've had a chance to read the display, you can press CTRL-Q to resume scrolling.

## 2.5 Using the calculators

UNIX provides a calculator (and a preprocessor for that calculator) that you can use right at your terminal.

### Using the desk calculator

**dc**

Use the **dc** command to call up a simple interactive desk calculator. Here is a typical session with **dc**, with comments to the right of each line:

```
$ dc              Call up the desk calculator
3                 Enter 3
4                 Enter 4
+                 Perform addition
```

```
p                 Display the result
7
3+p               Add 3 and display the result
10
4*p               Multiply by 4 and display the result
40
2/p               Divide by 2 and display the result
20
5-p               Subtract 5 and display the result
15
q                 Exit the desk calculator
$ _
```

Other features include number bases, scaling, functions, subscripts, and logical control. For further details, see the *UNIX User's Manual.*

## Using the high-precision calculator bc

UNIX offers another calculator called **bc** that allows unlimited precision, conversion of numbers from one base to another, a range of 0–99 places after the decimal point, variables, functions, arrays, and comments. Here is a typical session with **bc**:

```
$ bc              Call up the high-precision calculator
14 + 23           Add two numbers together—no equal sign
37
34 - 53           Subtract a larger number from a smaller
-19
8 * 9             Multiply two numbers together
72
72 / 12           Divide a number by another—even quotient
6
sqrt(81)          Take the square root of a number—exact root
9
scale = 10        Request ten places after the decimal point
z = sqrt(15)      Assign a value to the variable z
z                 Request the value of z (to ten places)
3.8729833462
define s(a,b) {   Define a function called s
 auto c           with automatic variable c
 c = a + b        that adds two arguments a and b and
 return(c)        returns the sum as the value of the function
}                 ended with a closing brace
x = sqrt(53)      Assign the square root of 53 to variable x
y = sqrt(31)      Assign the square root of 31 to variable y
```

```
s(x,y)                  Compute the sum of x and y
12.8478742520
quit                    Leave bc and return to the UNIX shell
$ _
```

For those with a little more background in programming and mathematics, here are a few additional features of **bc**:

```
$ bc                    Call up the calculator again
obase = 16              Change the output base to hexadecimal
65536                   Convert 65,536 from decimal to hexadecimal
10000
ibase = 8               Change the input base to octal
377                     Convert 377 from octal to hexadecimal
FF
ibase = A               Change the input base back to decimal
obase = A               Change the output base back to decimal
define f(n) {           Define function f (the factorial function)
 auto i,j               with automatic variables i and j
 j = 1                  with j initially set to 1 and i stepped
 for(i=1;i<=n;i++) \       from 1 to n in increments of 1
     j = j * i          assigning j the product of itself by i
 return(j)              and returning j as the value of the function
}                       ending the function with a closing bracket
f(20)                   Request the value of 20 factorial (20!)
2432902008176640000
quit                    Leave bc and return to the UNIX shell prompt
$ _
```

Note the continued line in the factorial function. This line was broken, using a backslash (\), only to save space. On your own system you would probably type the entire `for` statement on a single line. But as this example shows, you always have the option in UNIX of splitting a long line of input.

By entering **bc -l** at the command line instead of **bc**, you can also invoke a mathematical library, which includes sine (`s`), cosine (`c`), arctangent (`a`), exponential (`e`), natural logarithm (`l`), and Bessel (`j(n,x)`) functions. By entering **bc** *file(s)* instead of **bc**, you can have **bc** read statements from one or more files before accepting keyboard input. This allows you to store longer functions in files instead of having to type them over again every time you want to use them.

## Displaying the Calendar

Another kind of calculator called **cal** allows you to display any month or year from 1–9999 A.D. For example, here is a calendar for 1987:

```
$ cal 1987
                               1987

         Jan                    Feb                    Mar
 S  M Tu  W Th  F  S    S  M Tu  W Th  F  S    S  M Tu  W Th  F  S
             1  2  3    1  2  3  4  5  6  7    1  2  3  4  5  6  7
 4  5  6  7  8  9 10    8  9 10 11 12 13 14    8  9 10 11 12 13 14
11 12 13 14 15 16 17   15 16 17 18 19 20 21   15 16 17 18 19 20 21
18 19 20 21 22 23 24   22 23 24 25 26 27 28   22 23 24 25 26 27 28
25 26 27 28 29 30 31                          29 30 31

         Apr                    May                    Jun
 S  M Tu  W Th  F  S    S  M Tu  W Th  F  S    S  M Tu  W Th  F  S
          1  2  3  4                   1  2       1  2  3  4  5  6
 5  6  7  8  9 10 11    3  4  5  6  7  8  9    7  8  9 10 11 12 13
12 13 14 15 16 17 18   10 11 12 13 14 15 16   14 15 16 17 18 19 20
19 20 21 22 23 24 25   17 18 19 20 21 22 23   21 22 23 24 25 26 27
26 27 28 29 30         24 25 26 27 28 29 30   28 29 30
                       31
         Jul                    Aug                    Sep
 S  M Tu  W Th  F  S    S  M Tu  W Th  F  S    S  M Tu  W Th  F  S
          1  2  3  4                      1          1  2  3  4  5
 5  6  7  8  9 10 11    2  3  4  5  6  7  8    6  7  8  9 10 11 12
12 13 14 15 16 17 18    9 10 11 12 13 14 15   13 14 15 16 17 18 19
19 20 21 22 23 24 25   16 17 18 19 20 21 22   20 21 22 23 24 25 26
26 27 28 29 30 31      23 24 25 26 27 28 29   27 28 29 30
                       30 31
         Oct                    Nov                    Dec
 S  M Tu  W Th  F  S    S  M Tu  W Th  F  S    S  M Tu  W Th  F  S
             1  2  3    1  2  3  4  5  6  7          1  2  3  4  5
 4  5  6  7  8  9 10    8  9 10 11 12 13 14    6  7  8  9 10 11 12
11 12 13 14 15 16 17   15 16 17 18 19 20 21   13 14 15 16 17 18 19
18 19 20 21 22 23 24   22 23 24 25 26 27 28   20 21 22 23 24 25 26
25 26 27 28 29 30 31   29 30                  27 28 29 30 31
```

To display one month, type the corresponding number (1–12) between **cal** and the year. For example, here is a calendar for January 1987:

```
$ cal 1 1987
   January 1987
 S  M Tu  W Th  F  S
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
$ _
```

## 2.6 Other aids

### On-line manual **man**

If this feature is available on your system, you can execute the **man** command to display a section of the *UNIX User's Manual.* Type **man**, then the name of a UNIX command. This will give you a description of the command and all of its options. For example, to see the section on **who**, you could use

```
$ man who
```

### Visual shell (XENIX only) **vsh**

For XENIX users, the *visual shell* offers a series of screen displays to help you get started with the basic features of the system quickly and easily. To see the first display, type **vsh** and press RETURN. Then the following prompt will appear at the lower left-hand corner of the screen:

```
COMMAND: Copy Delete Edit Help Mail Name
         Options Print Quit Run View Window
Select option or type command letter
```

Use the left and right arrow keys (← and →) to select a file; use the space bar, TAB key, or the first letter of the command to select a command; then press RETURN twice to execute the command using the file that you selected.

## 2.7 Summary

In this chapter you learned some basic things about UNIX: preliminary set-up procedures, logging in and logging out, and running processes.

### Preliminary set-up procedures

Before you can begin using UNIX, you have to have an identifier, a password, and a terminal, which will be assigned to you by the system administrator for your system.

### Logging in and logging out

To log in to a UNIX system, establish connection with the computer, type your identifier, and type your password. To log out, type CTRL-D (possibly more than once) or type **logout** and press RETURN. To change your password, use the **passwd** command.

### Editing a command line

To begin a *process* under UNIX, type a command line and press RETURN. The command line may contain either just the command by itself or the command plus modifiers. To erase a single character on a command line, use CTRL-H (or possibly BACKSPACE). To erase an entire command line and start over again, use the at-sign (`@`) for UNIX or CTRL-U for XENIX. To restore the shell prompt after running a process, press DEL (or CTRL-C).

## Process control

To run a process in the background, type an ampersand (`&`) at the end of the command line. To find out what processes are running on UNIX at a particular moment, use the **ps** (process status) command. The system will respond with a list of processes that shows the process identification number, the terminal on which it was initiated, the amount of time it has been running, and the command line itself.

To abort a process running in the foreground, use the DEL key. To abort a process running in the background, use the **kill** command, along with the process identification number. To halt a rapidly moving screen display, press CTRL-S; to resume scrolling, press CTRL-Q.

## Using the calculators

To use the desk calculator, enter the **dc** command and enter "computational lines," with operations following the operands. Use the **q** command to exit. To use the high-precision calculator, enter the **bc** command and enter statements. Use **quit** to exit. To display any year 1–9999 A.D., use the **cal** command followed by the year; to display a month, type a number 1–12 between **cal** and the year.

## Other aids

To display a section of the *UNIX User's Manual* at your terminal, execute the **man** command. Type **man**, followed by the name of the command you wish described, then press RETURN. XENIX users can begin using the system quickly via screen displays and prompts with the aid of the *visual shell.* Type **vsh** and press RETURN to begin. A full-screen display will help you select a command and file to work with.

# 3

# The UNIX File System

## 3.1 What is a file system?

If you have used any kind of library, then you are already familiar with the concept of a *filing system.* A library offers you books kept on shelves, together with a *directory* that you can use to locate a particular book. In a library, the directory is called a card catalogue. Usually placed in an open area that is easy to find, the card catalogue gives you the location of each book in the library, together with brief information about the book.

Every operating system has a *file system* that resembles a library's filing system. In an operating system, the items stored are *files*, not books. Computer files can store text, data, graphics information, programs, and also directories to catalogue the other files. Each item in a typical directory provides a file's name, location, size, type, and possibly information about the file's accessibility to users. (In UNIX, only the file's name is in the directory itself; the rest of the information is kept somewhere else. But this introductory chapter is not the place to discuss that.)

If you have been using one of the common operating systems for microcomputers, then you may have been using an unstructured file system. In an unstructured file system, there is one large directory to catalogue files, with all files named in this one directory. With no structure to relate the files in any particular way, the directory simply lists the files directly.

## 3.2 A structured file system

In the UNIX file system, there are many directories, not just one. Furthermore, there is a clearly defined structure that places some directories within other directories and that may place different kinds of files in different directories. As a user of a UNIX system, you have your own personal directory, whose name is identical to the identifier that you type when you log in. Your directory, like other directories, may contain both files and other directories, allowing you to store related files together. Before we begin to discuss specific directories and files, there is one more point to consider: UNIX regards peripheral devices like terminals, printers, and disk drives as *files*. The procedure for accessing one of these devices is the same as the procedure for accessing any other file in the system.

A directory called usr contains the directories of all users of the UNIX system. (On very large systems, there may be several user directories, with names like usr1, usr2, or some other variation.) Your directory is known as your *home directory*. This is the directory that the system administrator assigns to you, the directory in which you begin working whenever you log in. Within this directory, you can create as many subdirectories as you need to organize your work. You can create one directory for your text files, one directory for your program files, one directory for your messages, and so on. Some of the other directories on a typical UNIX system are as follows:

- bin—contains the system's binary files, also known as executable object code files (most commands reside here)
- dev—contains the files representing the system's devices (terminals, printers, disk drives, and so on)
- tmp—contains the system's temporary storage files
- etc—contains miscellaneous files that are primarily used for system administration

These five major directories all belong to a primal directory called the *root directory* (or simply the *root*), forming the main branches of a tree. It is customary to depict this tree upside down, with the root at the top and the branches pointing downward. Part of a simple UNIX file system is shown in Figure 3.1, with home directories for users Dan, Robin, and Ann.

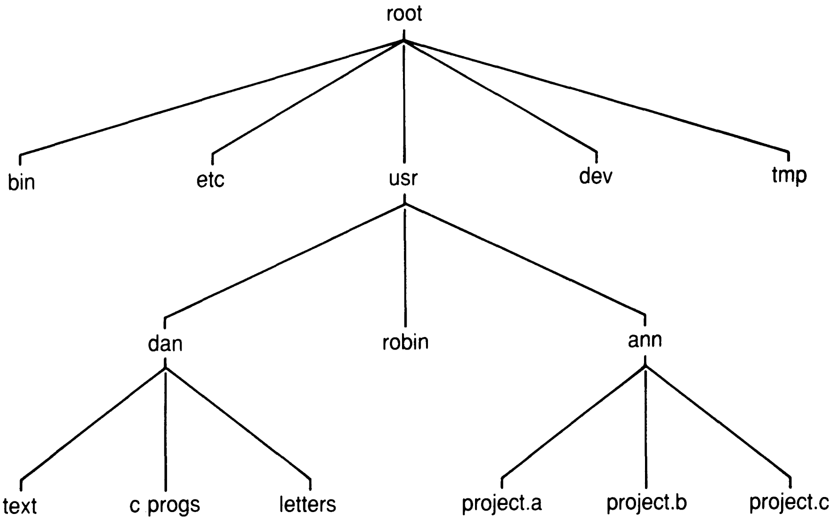
UNIX, then, has a structured file system that contains three kinds of files: *directories*, which store the names of other files (including other directories); *ordinary files*, which store text, source programs, and object code; and *special files*, which correspond to peripheral devices. In the rest of this chapter, we'll discuss how directories and files are named, how to use pattern-matching characters to select files, how to work with files, how to work with directories, and how to grant or deny access to your directories and files.

## Naming directories and files

The *root* directory is identified by a single character: slash (/). To name one of the major directories directly under root, type a slash (/) to represent root, followed by the directory's own name, as in /usr. The slash in front of usr tells you that usr is a subdirectory of root. Referring to Figure 3.1, here is how you would identify root and the major directories:

| | |
|---|---|
| / | root directory |

FIGURE 3.1. Part of a typical UNIX file system.



| | |
|---|---|
| /usr | user directory |
| /bin | binary directory |
| /dev | device directory |
| /etc | miscellaneous directory |
| /tmp | temporary directory |

To identify one of the user home directories, type another slash after /usr, followed by the account name, as in /usr/dan and /usr/ann. In each case here, the first slash refers to the root, while the second indicates that dan and ann are subdirectories of usr. (In UNIX terminology, usr is the *parent directory*, while dan and ann are *subdirectories*.)

To identify the subdirectories under dan and ann, we continue with the same conventions. We can identify Dan's text directory as /usr/dan/text and Ann's project.a directory as /usr/ann/project.a. Referring again to Figure 3.1, here is how you would identify all their directories:

/usr/dan/text
/usr/dan/c_progs
/usr/dan/letters

/usr/ann/project.a
/usr/ann/project.b
/usr/ann/project.c

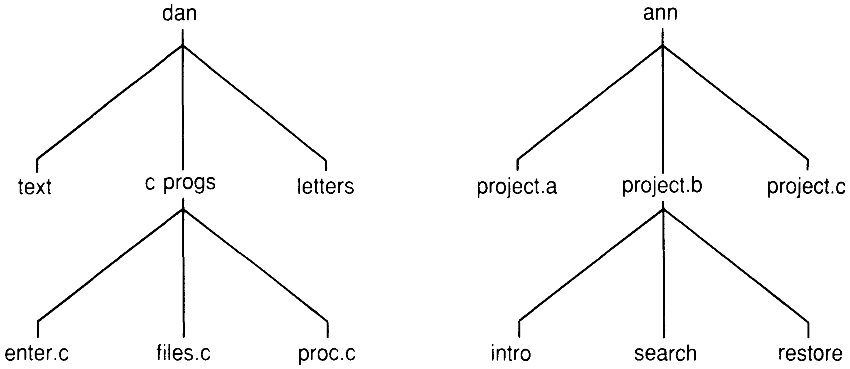
Now suppose Dan has three C programs called enter.c, files.c, and proc.c and Ann has three files under project.b called intro, search, and restore. Then we could depict these files as shown in Figure 3.2.

To identify these six files, you could give their full *pathnames*:

/usr/dan/c_progs/enter.c
/usr/dan/c_progs/files.c
/usr/dan/c_progs/proc.c

/usr/ann/project.b/intro
/usr/ann/project.b/search
/usr/ann/project.b/restore

FIGURE 3.2. User files in a UNIX file system.



Ann (and other users) would have to use the names in the left-hand column to refer to Dan's files; Dan (and other users) would have to use the names in the right-hand column to refer to Ann's files. However, when Dan and Ann refer to their own files in their own directories, they can use these shorter names (*partial pathnames*):

```
c_progs/enter.c   project.b/intro
c_progs/files.c   project.b/search
c_progs/proc.c    project.b/restore
```

Later in this chapter, we'll you show how to move around the file system from one directory to another (change directories). If Dan should move from his own home directory to Ann's, then he could use the shorter names that Ann uses when she is working from her own home directory; if Ann should move from her own home directory to Dan's, then she could use the shorter names that Dan uses when he is working from his own home directory.

Furthermore, if Dan should move from his home directory to his own subdirectory c_progs and Ann should move from her home directory to her own subdirectory project.b, then they can use even shorter names for their files:

```
enter.c   intro
files.c   search
proc.c    restore
```

## Rules for naming and accessing files

The rules for naming and accessing files (including directories) are closely related to the structure of the UNIX file system:

- The root directory is identified by a slash (/).
- A simple filename can be any combination of 1-14 characters *other than* slashes (/), asterisks (*), question marks (?), quotation marks (" or '), square brackets ([ or ]), or control characters.

- A pathname is a sequence of directory names, possibly followed by a simple filename, with each pair of names separated by a slash (/).

To avoid misinterpretation, the safest characters to use for simple filenames are letters of the alphabet, numbers, periods (.), hyphens (-), and underlines (_). Note: In UNIX, upper and lower case letters are *not* the same (e.g., **newfile** is not the same as **NEWFILE**).

The directory permanently assigned to you is called your *home directory*; this is the directory to which you log on. Any directory to which you may move after logging on (including your home directory) will be called your *current directory*, or *working directory*, for as long as you remain in that directory. The directory in which your current directory resides at any moment is called your *parent directory.* UNIX provides shorthand symbols to indicate your current directory (.) and your parent directory (..).

If a pathname used to access a file begins with a slash (/), then the search for the file begins at the *root directory.* Such a pathname is called an *absolute pathname* (or *full pathname*), since it always begins with the *root directory.* If a pathname begins with a simple filename, then the search for the file begins at your current directory. Such a pathname is called a *relative pathname*, since the file is accessed with respect to your current directory.

Later in this chapter we'll discuss procedures for giving other users *permission* to access to your directories and files. It goes without saying that you can access only those directories and files for which you have permission.

## 3.3 Working with directories

This section shows you how to work with UNIX directories using six common commands.

### DISPLAYING THE CONTENTS OF A DIRECTORY **ls**

To sort and display the names of all the directories and files that reside in your current directory, use the **ls** command, as illustrated here:

```
$ ls
file.1
file.2
letters
memos
specs
$ _
```

This doesn't show which names refer to files and which refer to directories. However, there is another form of this command that you can use to distinguish files from directories (and also to display a lot of other information). Just add the **-l** ("hyphen el" or "minus el") option to get a more detailed long listing, as shown here:

```
$ ls -l
total 501
-rw-r-----  1  robin     108  Apr  5  14:33  file.1
-rw-r-----  1  robin     123  Apr  9  09:17  file.2
drwx--x---  2  robin     301  Mar 27  08:04  letters
drwx--x---  1  robin      87  Mar 15  13:42  memos
drwx--x---  2  robin     428  Mar 11  15:31  specs
$ _
```

We'll discuss this listing in greater detail in the next section. Here is a quick summary of the information given for each file or directory:

- Type of file: ordinary (-) or directory (d)
- Permissions, discussed in the next section
- Number of *links* in the file system to other users
- Owner, or creator, of the file
- Size of the file in bytes (characters)
- Date and time of last modification of the file
- Name of the file

## Changing your working directory **cd**

To change your working directory (that is, to move to another directory), use the **cd** (change directory) command, as in

```
$ cd /usr/harold
$ _
```

If you have been given access, you can now operate within Harold's home directory. To return to your home directory from any other directory, use the **cd** command without a name following, as shown here:

```
$ cd
$ _
```

## Determining your working directory **pwd**

To find out the name of your current working directory at any moment, use the **pwd** (print working directory) command, as in

```
$ pwd
/usr/robin
$ _
```

## Creating a new directory **mkdir**

To create a new subdirectory within your current working directory, use the **mkdir** (make directory) command, as in

```
$ mkdir messages
$ _
```

This command will create a new subdirectory called *messages*. In setting up your home directory, try to find the right balance between too few subdirectories and too many. With too few, you fail to take advantage of the structure of the UNIX file system; with too many, you create a maze for yourself. Somewhere between five and fifteen main subdirectories is usually optimal, but there may be exceptions.

## Removing an existing directory **rmdir**

To remove an existing directory from your working directory, move to the target directory, delete all its files, move back to the parent directory, and then use the **rmdir** (remove directory) command. For example, suppose you would like to delete a directory called **useless**. You could follow the procedure illustrated here:

```
$ cd /usr/useless     [Move to directory useless]
$ pwd                 [Make sure you are in the right directory]
/usr/useless
$ rm -i *             [Delete all files in useless]
letter: ?
introd: ?
memo_3: ?
$ cd ..               [Move to the parent directory]
$ rmdir useless       [Delete directory useless—now empty]
$ _                   [The directory and its files are gone]
```

If you try to remove a directory that is not empty, you will see a warning displayed. On many systems you may be able to use one of the following shorter methods instead of the above:

```
$ rm /usr/useless/*        [Delete all files in directory useless]
$ rmdir /usr/useless       [Delete directory useless]
$ _

$ rm -r /usr/useless       [Delete files in useless, then delete directory
$ _                         useless itself                               ]
```

### Renaming a directory **mv**

To change the name of a directory, use the **mv** (move) command. For example, to change the name of a directory from **old.name** to **new.name**, use this command:

```
$ mv old.name new.name
$ _
```

## 3.4 Working with files

Whether you are composing letters, performing calculations, or writing programs, you spend much of your time on a computer dealing with files. It's always helpful to know how to display, combine, copy, and otherwise manipulate files. In this section we discuss such ways of working with UNIX files.

### Displaying the contents of a file **cat**

To display the contents of any file, use the **cat** (concatenate) command. The **cat** command simply displays the contents of a file (or several files) on the screen, as in this example:

```
$ cat file.3                     $ cat file.1 file.2
This is a very short file        This is file.1
that contains two lines.         This is file.2
$ _                              $ _
```

### Combining files **cat**

Another function of the **cat** command is to combine, or concatenate, files with the result usually stored in another file. In this example

```
$ cat file.1 file.2 > file.3
$ _
```

the contents of files **file.1** and **file.2** are concatenated and stored as a single file called **file.3**. In using the **cat** command, avoid storing the result of a concatenation in one of the original files, since this will cause the original file to be overwritten. For example, in the command

```
$ cat file.1 file.2 > file.1  [No good—don't do this!]
$ _
```

**file.1** simply becomes **file.2**, which is not at all the intended result.

## Renaming a file **mv**

You can use the **mv** (move) command either to rename a file or to move it from one directory to another. To change the name of a file, enter a pair of commands like this:

```
$ cat new.name
cat: cannot open new.name
$ mv old.name new.name
$ _
```

The **mv** command will change the file's name from **old.name** to **new.name** (you can then no longer access **old.file**). The **cat** command is used to make sure that **new.name** isn't already the name of another file. If it is, that file will be replaced and lost.

## Moving a file **mv**

To move a file (or several files) from one directory to another (without changing their names), give the name of the new directory last on the command line, as in

```
$ mv file.3 file.4 text
$ _
```

This command, executed from the parent directory, will move **file.3** and **file.4** from your current directory to a subdirectory called **text**. You can verify this with the **ls** command:

```
$ ls text
file.3
file.4
$ _
```

You can also rename a file during a move by using a partial pathname. For example, to move **file.3** to subdirectory **text** and rename it **entries**, you could use this:

```
$ mv file.3 text/entries
$ _
```

## Copying a file **cp**

To make a duplicate copy of a file, use the **cp** (copy) command illustrated here:

```
$ cp file.one FILE.ONE
$ _
```

This command will make a copy of **file.one** called **FILE.ONE**, so that the same file is now accessible by either name. (A reminder to users of other operating systems: In UNIX, **file.one** and **FILE.ONE** are *different* names.)

To copy a file (or several files) from one directory to another, use the **cp** command from the directory that contains the files, as shown here:

```
$ cp letter-a letter-b letter-c correspondence
$ _
```

This command will copy **letter-a**, **letter-b**, and **letter-c** from the current directory to a subdirectory called **correspondence**.

You can also rename a file as you copy it. For example, to copy **letter-c** to **correspondence** and rename it **ltr.04-16-85**, you could use this:

```
$ cp letter-c correspondence/ltr.04-16-85
$ _
```

## Deleting a file **rm**

To delete a file (or several files), use the **rm** (remove) command, as shown here:

```
$ rm intro.1 intro.2 intro.3
$ _
```

This form of the command will delete the files **intro.1**, **intro.2**, and **intro.3** from the current directory immediately. If you would like to confirm before proceeding to delete the files, add the **-i** (interactive) option:

```
$ rm -i intro.1 intro.2 intro.3
intro.1: ?
intro.2: ?
intro.3: ?
$ _
```

This is a convenience worth the extra moment it takes to confirm the deletion. Once the files have been deleted, you can't recover them—unless (1) another user has a *link* to these files or (2) they have been saved on a back-up tape.

## Linking a file **ln**

In UNIX, a given file may be known by more than one name. Furthermore, the different names by which the same file is known may be used by different users in different directories. Each name by which a file is known is called a *link* to the file. Any changes that any user makes to the file will be in effect for each name by which the file may be known.

For example, if Dan should execute the following **ln** (link) command from his subdirectory **letters**, then he will be able to access the file **intro** from his own subdirectory **letters**, using the same name (**intro**):

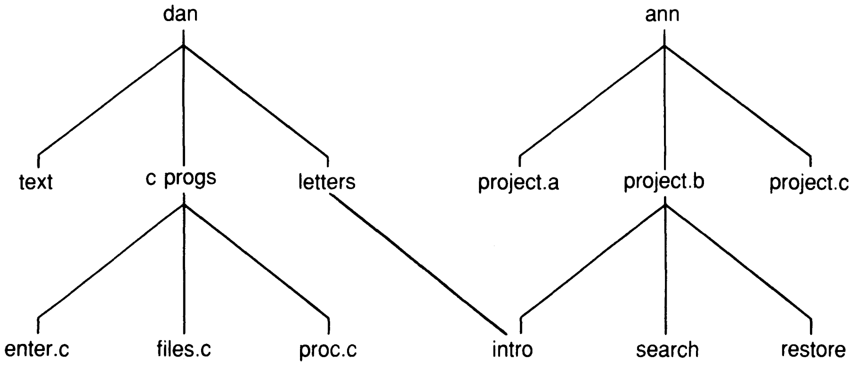
```
$ cd letters
$ ln /usr/ann/project.b/intro intro
$ _
```

If Dan prefers another name for the file, he can give the preferred name in the **ln** command line. In the following example, Dan makes the same link as in the previous example, but this time the file will become known as **discover** to Dan (see Figure 3.3):

```
$ ln /usr/ann/project.b/intro discover
$ _
```

It is not possible to link a directory to another directory or to link a file in a different file system. To remove a link, use the **rm** command.

FIGURE 3.3. Linking file **intro**.



## Matching a character **?**

If you ever have to move, copy, or delete a large number of files, you can save time and keystrokes by using one of the UNIX wild card characters for matching filenames. There are three varieties, and they generally have the same meanings throughout UNIX.

To match any single character in a filename, you can use a question mark (?) in the desired position in the filename. For example, to delete existing files intro.1, intro.2, and intro.3, as in the example above, you could use a command like this:

```
$ rm intro.?
$ _
```

If these are the only existing files with names that begin with intro. and end with a single character, then this command is equivalent to the command shown in the preceding section, "Deleting a File," p. 34).

If letter-a, letter-b, and letter-c are the only existing files with names that begin with letter- and end with a single character, you can use the command

```
$ cp letter-? correspondence
$ _
```

to copy all three of them to directory correspondence.

In each example above, we have spoken of "existing files." You can never use a wild card character, such as "?," to refer to a file that does not yet exist.

## MATCHING SPECIFIC CHARACTERS [ ]

To narrow the selection process to a specific set of characters and then match one of these characters in a filename, enclose the desired characters within square brackets in the appropriate position in the filename. For example, suppose you have files in your working directory with these names:

writer.0
writer.1
writer.2
writer.3
writer.4
writer.5
writer.6
writer.7
writer.8
writer.9
writer.10
writer.11

To delete writer.5, writer.7, and writer.9, you could use the following command:

```
$ rm writer.[579]
$ _
```

The three numbers inside the brackets let you select the three files desired.

You can also give a range of letters or numbers in brackets. For example, to delete writer.1, writer.2, writer.3, and writer.4, you could use this command:

```
$ rm writer.[1-4]
$ _
```

### Matching any number of characters *

To match any number of characters in a filename, you can use the asterisk (*) in the desired position in the filename. (To match the entire name, use the asterisk alone.) For example, suppose you have the following files in your working directory:

```
info.a   info.ab   info.abc   info.test
info.b   info.23   info.new   info.old
```

Then you could change to the parent directory and use the following command to move all of these files from directory **obsolete** to a subdirectory called **information**.

```
$ mv obsolete/info.* information
$ _
```

To use another example, suppose you have four C programs in subdirectory **text**:

```
enter.memo   files.dept   wide_col.c      news.ltr
F_327.c      a_file.text  new_compare.c   get_number.c
```

You could use a command like this to copy all of the files that end in **.c** to a subdirectory called **C_programs**:

```
$ cp text/*.c C_programs
$ _
```

Be very careful when you use the asterisk (*), especially in a **rm** command. If you don't type the command exactly right, you may delete files unintentionally.

## 3.5 File and directory permissions

As noted in Chapter 2, UNIX allows you to access other directories and files in the system, but only if you have permission from the owners of those directories and files. This section deals with the UNIX system of permissions, which apply to individual owners, groups of users, and other users.

### Determining permissions ls -l

To determine the permissions associated with a given file or directory, use the **ls -l** command described in the previous section. The permissions are indicated by the nine characters that follow the first character:

```
$ ls -l
total 501
-rw-r-----  1  robin      108  Apr  5  14:33  file.1
-rw-r-----  1  robin      123  Apr  9  09:17  file.2
drwxr-x---  1  robin       87  Mar 15  13:42  memos
drwxr-xr--  2  robin      301  Mar 27  08:04  letters
drw-rw-rw-  1  robin      216  Mar  3  11:56  proposals
drw-r--r--  2  robin      428  Mar 11  15:31  specs
```

The first character, as noted before, indicates the type of file:

| | |
|---|---|
| - | ordinary file |
| d | directory |

The remaining nine characters represent three sets of three characters: one set for the individual user, one for the user's working group (if any), and one for all other users. If we take the display in the example and spread out the characters to show the groupings, we get something like this:

| **Type** | **User** | **Group** | **Others** | |
|---|---|---|---|---|
| - (file) | r w - | r - - | - - - | file.1 |
| - (file) | r w - | r - - | - - - | file.2 |
| - (file) | r w x | r - x | - - - | memos |
| - (file) | r w - | r - x | r - - | letters |
| d (directory) | r w - | r w - | r w - | proposals |
| d (directory) | r w - | r - - | r - - | specs |

In each of these three groups of characters, there is one permission for reading, one for writing, and one for executing. *Reading*, *writing*, and *executing* have different meanings for ordinary files and directories:

For an *ordinary file*, permissions are defined as follows:

- *read* permission means you may look at the contents of the file
- *write* permission means you may change the contents of the file
- *execute* permission means you may type the name of the file in a command line as if the file were a UNIX command.

For a *directory*, permissions are defined as follows:

- *read* permission means you may see the names of the files in the directory
- *write* permission means you may add files to and remove files from the directory
- *execute* permission means you may change to the directory, search the directory, and copy files from it

The characters used to represent these permissions are as follows:

| | | | |
|---|---|---|---|
| r | read permission | w | write permission |
| x | execute permission | - | permission denied |

Putting this all together, here are the nine characters for directory memos in the display above:

| **Owner** | **Group** | **Others** | **File** |
|---|---|---|---|
| r w x | r - x | - - - | memos |

This shows that the owner of memos has permission to read (r), write (w), and execute (x); members of the owner's group have permission to read (r) and execute (x), but not to write (-); and all other users are denied access of any kind (- - -).

Here are the nine characters for file.1:

| **Owner** | **Group** | **Others** | **File** |
|---|---|---|---|
| r w - | r - - | - - - | file.1 |

This shows that the owner of file.1 has permission to read (r) and write (w), but not to execute (-); members of the file's working group have permission only to read (r), but not to write (-) or execute (-); and all other users are denied access of any kind (- - -).

## Changing permissions **chmod**

You can make changes to permissions by entering a **chmod** (change [access] mode) command. The **chmod** command allows the owner of the file to add to (**+**) or remove from (**-**) existing permissions. It also allows the owner to clear existing permissions and assign all permissions from scratch; this is known as assigning permissions absolutely (**=**). The **chmod** command affects any of the three types of access for any of the three categories of UNIX users, using one-letter symbols in the following order (left to right):

| | | | | | |
|---|---|---|---|---|---|
| **u** | owner (user) | **+** | add permission | **r** | to read |
| **g** | File's group | **-** | remove permission | **w** | to write |
| **o** | all others | **=** | absolute permission | **x** | to execute |
| **a** | all (default) | | | | |

For example, suppose you wanted to grant permission to write for members of your working group and permission to read and write for all other UNIX users for file.1. You could use these expressions in a **chmod** command:

**g+w, o+rw** Add permission (**+**) to write (**w**) to your working group (**g**); and add permission (**+**) to read (**r**) and write (**w**) to all other users (**o**)

Then, using the **ls -l** command to display *before* and *after*, you could incorporate these expressions into a **chmod** command, as shown in the following sequence. Note: You must type the command line for **chmod** exactly as shown, with a comma between **g+w** and **o+rw** and no spaces surrounding the comma.

```
$ ls -l file.1
-rw-r-----  1  robin      108  Apr  5  14:33  file.1
$ chmod g+w,o+rw file.1
$ ls -l file.1
-rw-rw-rw-  1  robin      108  Apr  5  14:33  file.1
$ _
```

To remove permissions currently in effect, simply use a minus sign (**-**) in place of a plus sign (+), and then form a **chmod** command in the same way. For example, suppose you wanted to remove permission to write for members of your working group and permission to read and write for all other UNIX users for file.1 (that is, revoke the permissions granted in the previous example). You could use these expressions in a **chmod** command: **g-w** Remove permission (**-**) to write (**w**) to your working group (**g**) **o-rw** Remove permission (**-**) to read (**r**) and write (**w**) to all other users (**o**)

Then, using the **ls -l** command to display *before* and *after,* you could incorporate these expressions into a **chmod** command, as shown in the following sequence:

```
$ ls -l file.1
-rw-rw-rw-  1  robin      108  Apr  5  14:33  file.1
$ chmod g-w,o-rw file.1
$ ls -l file.1
-rw-r-----  1  robin      108  Apr  5  14:33  file.1
$ _
```

To clear permissions currently in effect and assign permissions from scratch, use an equal sign (**=**) to form the **chmod** command. The command in the following example achieves the same result as the one in the previous example:

```
$ ls -l file.1
-rw-rw-rw-  1  robin      108  Apr  5  14:33  file.1
$ chmod u=rw,g=r file.1
$ ls -l file.1
-rw-r-----  1  robin      108  Apr  5  14:33  file.1
$ _
```

**Caution:** It's possible for you to lock yourself out of one of your own files with **chmod**. Be careful when you type it.

## 3.6 Summary

After a brief introduction to file systems, this chapter discusses the structure of the UNIX file system, followed by basic procedures for working with files, directories, and permissions. A computer file system may be compared to the filing system used in a library, where the card catalogue roughly corresponds to a *directory* and the books roughly correspond to *files*. In UNIX, a directory is also a file itself.

UNIX has a structured file system, with a primal directory called *root* at the top and at least five major directories branching out from the root directory. Five major directories are **usr** (user), **bin** (binary), **dev** (devices), **tmp** (temporary), and **etc** (miscellaneous). Each UNIX user has a *home directory* in the **usr** directory (or in a user directory with another name). The name for this home directory is the same as the identifier with which the user logs on. The user may create as many subdirectories in his or her home directory as necessary.

The full pathname of every file begins with root (/), then includes the name of the major directory, followed by a subordinate directory, and so on, down to the name of the file itself. The slash (/), the symbol that represents the root directory, is also used to separate directory names and the simple filename from each other. A simple filename consists of 1-14 characters other than the following (and control characters): / * ? " ' [ ]

After you log in to your home directory, you can then move from one directory to another, provided you have permission. Any directory from which you are operating at a particular moment is known as your *working directory*, or *current directory*. While a *full pathname* can be given from any directory, you can also use a *relative pathname*—a pathname relative to your current directory.

### Working with directories

To display a sorted list of the names of all directories and files in your current directory, use the **ls** command (with **-l** for more details). To change your working directory (that is, to move from one directory to another), use the **cd** command, followed by the name of the new directory (no name for your home directory). To find out the name of your current working directory, use the **pwd** command.

To create a new subdirectory within your working directory, use the **mkdir** command. To remove an existing directory from your working directory, move to the target directory, delete all its files, move back to your working directory, and then use the **rmdir** command. (You may also be able to use **rmdir -r**.) To change the name of a directory, use the **mv** command with the new name last.

## Working with files

To display the contents of a file, use either **cat** or **more**. To combine, or concatenate, files, use the **cat** command with redirection of output to the target file (>). To rename a file, use the **mv** command, the old name, then the new name. (Any existing file with the new name will be lost.) To move a file (or files) from one directory to another, use the **mv** command, the name(s) of the file(s), then the pathname of the new directory.

To make a copy of a file within the same directory, use the **cp** command, the old name, and the new name. To copy a file (or files) from one directory to another, use the **cp** command, the name(s) of the file(s), and the name of the directory. To delete a file (or files), use the **rm** command, followed by the name(s) of the file(s). To *link* a file (that is, to attach a file to a different directory), use the **ln** command, giving the full pathname of the file being linked and optionally giving the file a different name for use in its new directory.

To match any single character in a filename, use a question mark (?) in the desired position. To match one of a set of characters in a filename, enclose the characters within square brackets ([ ]) at the desired location in the string. To match any number of characters in a filename, use an asterisk (*) in the desired position.

## File and directory permissions

To determine permissions associated with a file or directory, use the **ls -l** command. The first character indicates whether the entry is an ordinary file (-) or a directory (d). The next nine characters indicate whether permission to read (r), write (w), or execute (x) has been granted to the owner, the file's working group, or other users.

To change existing permissions for a directory or file, use the **chmod** command, either adding (**+**) or removing (**-**) permission to read (**r**), write (**w**), or execute (**x**) for the owner (**u**), the file's group (**g**), all other users (**o**), or all users (**a**). To clear existing permissions and assign all permissions from scratch, use **chmod** with the symbol for assigning permissions absolutely (**=**).

# 4

# Using UNIX Commands

Now that you are familiar with the UNIX file system, you are ready to learn more about UNIX commands. This chapter begins with a general discussion of command lines; then covers methods of interacting between UNIX processes; and concludes with descriptions of commands for displaying text on the screen, processing text files, and using lineprinters.

## 4.1 Constructing a command line

### Command lines in general

In general, a command consists of three parts, although not every command requires all three parts:

name of command    options    name(s) of file(s)

There isn't much to say about the command's name, except that most UNIX commands have short names. Command options are usually designated by a hyphen (or minus sign), followed by a single letter (also called a *switch*). Sometimes you can type more than one letter after a single minus sign (to indicate multiple options); sometimes you cannot. In a few instances, command options are designated by plus signs instead of minus signs. Many commands allow one or more input files to be named. (Output files are generally, but not always, designated by an output option switch like **-o**. Another method of designating an output file will be discussed later in this chapter.) The various options and filenames that follow the command are referred to, collectively, as *arguments*.

### An example

As an example, consider the **ls** (list contents of directory) command, discussed in Chapter 3, "The UNIX File System." The *UNIX Programmer's Manual* (or *UNIX User's Manual*) shows 21 possible options for this command, as follows (spread out here with headings added for easier reading):

| **Name** | **Command Options** | **File(s)** |
|---|---|---|
| **ls** | [ **-RadCxmlnogrtucpFbqisf** ] | [ *name* ... ] |

The name of the command is **ls** (**list**). There are 22 different options—21 switches plus no switch—you can use: **-l** (long format), **-t** (time of last modification), **-a** (all entries), **-s** (give size), **-d** (name only for directories), **-r** (reverse order), and so on (some of the others are unsuitable for this discussion). The brackets (which are not to be typed on a command line) indicate that all option switches are optional, never required. The compression of the 21 letters into a single word indicates that more than one letter can be typed after a single minus sign. (Don't bother trying to learn how to pronounce "RadCxmlnogrtucpFbqisf.") In the case of this command, some options turn off other options. Finally, the word *name*, followed by ellipses, indicates that you can type at least one directory name after the options.

Given this information, here are a few of the command lines that can be constructed with the **ls** command:

| | |
|---|---|
| `$ ls` | List the contents of the current directory |
| `$ ls -l ..` | List the contents of the parent directory (long listing) |
| `$ ls -als /usr/paul` | List all entries (long listing), giving file sizes, of the contents of /usr/paul |
| `$ ls -a /etc /bin` | List all entries in /etc, then all entries in /bin |

We won't go into all 21 of the options in detail here. However, here are a few more of interest: **-C** (multi-column, sorted down); **-x** (multi-column, sorted across); **-t** (sort by time of last modification); **-u** (sort by time of last access); **-F** (mark directories with /, executable files with *); **-p** (mark directories with /); **-R** (list subdirectories recursively); **-q** (replace nongraphic characters with ?).

## 4.2 Redirection of input and output

The entire operation of a computer can be summed up in three phases:

- Input—the user supplies a computer program with information to process

- Processing—the computer program performs a set of functions on the information received from the user

- Output—the computer program returns the results of processing to the user

Although most of this takes place electronically, the basic procedure is similar to stepping up to a window in a bank to make a deposit. You hand the teller an endorsed check with a deposit slip (input); the teller makes a record of the deposit and stamps a receipt (processing); then the teller hands you the stamped receipt to keep for your records (output).

When you deal with your computer through UNIX, it is common for you to submit input via your terminal's keyboard and to receive output via your terminal's video screen. In fact, UNIX regards your keyboard as its *standard input* and your screen as its *standard output.* For example, UNIX will ordinarily assume that the command **date** will be typed at your keyboard and that the information requested by **date** (date, day of the week, hour, minute, and second) is be be displayed on your screen.

However, with most UNIX commands, you can at any time instruct UNIX to *redirect* the input or output of a command. For example, you can have UNIX receive input from a file instead of from the keyboard. Or you can have UNIX send output to a line printer instead of to the screen. (Keep in mind that UNIX regards peripheral devices like line printers as *files.*) The symbols used in a command line to request redirection are the *less than* sign (<) and the *greater than* sign (>) ((SHIFT) comma and (SHIFT) period, respectively, on most keyboards). You can think of these symbols as arrowheads pointing in the direction of the flow of information.

## Redirection of Input

As an example, UNIX has a **mail** command that takes the text that you supply as input and places it in the directory of each user you name after **mail** in a command line. One way to send a letter to Mary, John, Sandy, and Paul would be to type the following after the UNIX shell prompt and then type the letter at your keyboard:

```
$ mail mary john sandy paul
```

Suppose you use the screeen editor **vi** (Part II) to write your letter and then store the text in a file named letter_5. You could then send this letter to these same users on your UNIX system using the **mail** command with redirection of input. Then, to send your letter to Mary, John, Sandy, and Paul, you could type the following after the UNIX shell prompt and press (RETURN):

```
$ mail mary john sandy paul < letter_5
```

The direction of the arrowhead tells you that letter_5 is the input file, which is taking the place of your keyboard.

### Redirection of output

Just as you can use the input symbol (<) to redirect *input*, you can also use the output symbol (>) to redirect *output*. For example, the command line

```
$ ls
```

will list all the files in your directory on your screen. UNIX also allows you to modify the command line

```
$ ls > files
```

to redirect the output of **ls** from your screen to a file named **files**. If **files** does not exist when you issue the **ls** command, it will be created by the shell; if **files** does already exist, its contents will be overwritten.

A common use of redirection of output is the joining together, or *concatenation* of several files, using the UNIX **cat** command. To concatenate files **file_1**, **file_2**, and **file_3** and store the resulting text in another file called **append**, type the following after the UNIX shell prompt and press (RETURN):

```
$ cat file_1 file_2 file_3 > append
```

Then, to concatenate three more files and add the resulting text to **append**, rather than overwrite it, use another redirection symbol (>>). For example, to concatenate **file_4**, **file_5**, and **file_6** and *add* the resulting text to **append**, type the following after the UNIX shell prompt and press (RETURN):

```
$ cat file_4 file_5 file_6 >> append
```

If the file doesn't exist, >>, like >, causes the shell to create a new file; but if the file already exists, >> always adds to the end of it, never overwriting it.

## 4.3 Pipelines

As noted in the previous section, UNIX regards the keyboard as the *standard input* and the video screen as the *standard output* for most commands being executed, or *processes*. In addition to *redirection*, there is another way to alter the standard way of dealing with input and output: UNIX can connect two processes with a *pipe* (or *pipeline*), so that the output of one process becomes the input for another. The symbol for a pipe is the vertical bar ( | ), which is usually placed in different locations on different keyboards.

Without pipes, if you wanted to use the output of one process as the input for another, you would have to go through a roundabout procedure. For example, suppose you had three small text files called **part_d**, **part_e**, and **part_f**. You would like to keep the files separate in UNIX, but when you print them on the lineprinter, you would like to see all the text on one page, rather than spread over three different pages. To accomplish this, first concatenate the files and store them in a fourth file:

```
$ cat part_d part_e part_f > temp_file
```

Then use this new file as input for the **lp** command, which sends text to the lineprinter:

```
$ lp temp_file
```

Finally, use the **rm** (remove) command to delete the temporary file that you used to store the combined files:

```
$ rm temp_file
```

With a pipeline, you can connect **cat** and **lp** directly, eliminating the need for an intermediate file, like this:

```
$ cat part_d part_e part_f | lp
```

The pipe symbol ( | ) tells UNIX to take the output from **cat**, which otherwise would have gone either to a file or to the screen, and use it as the input for **lp**. This will accomplish the desired result with one command line instead of three: UNIX will concatenate the three files and send the resulting text to the lineprinter to be printed.

Commands that appear in pipe statements may include all the usual options and file designations. For example, the **pr** (print) command displays text on the screen. (The reason that it's called print instead of display is that the original terminals for UNIX were printing Teletype machines, not video display terminals.) In its modified form, the command **pr -4** displays text in four columns.

You can also set up multiple pipelines. For example, to have your files printed in three columns on a lineprinter, rather than displayed on the screen (assuming you have more than 150 files in your directory), you could enter a command like this:

```
$ ls | pr -3 | lp
```

In this command line, **ls** provides the list of files, **pr -3** formats the list in three columns, and **lp** prints the formatted list on the lineprinter.

Having at your disposal all the commands of UNIX, plus the pipeline feature to connect them in various ways, is like sitting on the floor in front of a box of tinker toys. You can construct all kinds of clever new commands out of the existing commands. The combinations are endless. Here are a few more examples, using just a handful of commands: the **who** command to list all users currently logged on the system, the **sort** command to alphabetize lines of text, the **ls** command to list your files, and the **wc** (word count) command to count the number of lines, words, and characters in a file.

| Command Line | Purpose |
|---|---|
| `$ who | sort` | To see a list of users in alphabetical order |
| `$ who | wc` | To see how many users are currently logged on |
| `$ ls | wc` | To see how many files are in your directory |

Note that, because of the way different commands work, many conceivable pipeline combinations are not possible to construct. For example, neither **who** nor **ls** could ever be on the receiving end of a pipe; these processes gather their information from within the system, never from external input. On the other hand, neither **sort**, **wc**, nor **lp** could ever be on the originating end of a pipe; these processes must receive input to be able to function.

## 4.4 Displaying text on the screen

In Chapter 3 you learned that you can use the **cat** (concatenate) command to display text on your screen. In this section you will learn more about **cat**, and also learn about commands that you can also use to display text: **more**, **pg**, **head**, and **tail**.

### Entering text into a file **cat**

Using the concept of redirecting output, you can use the **cat** command to perform simple text entry. All you have to do is redirect the output of **cat** from the screen to a file. Then type the text, pressing CTRL-D after the last line to indicate to **cat** that you have no more to type. In the following example, we enter the text shown into a file called enter:

```
$ cat > enter
Here is a short message
to show what we can do
with the cat command.
                    [Press CTRL-D—nothing will be displayed]
$ _
```

Now we can use **cat** without redirection to display the text we have just entered:

```
$ cat enter
Here is a short message
to show what we can do
with the cat command.
$ _
```

If you are using a microcomputer with communications software as a terminal to a UNIX system, you can also use the **cat** command in the way just shown to *capture* text and store it on one of your own disk files. (This is also called *downloading* data.) Just start the capture feature in your communications program, log into UNIX, and execute a **cat** command like the one just shown to display text on the screen.

To send text from your microcomputer to UNIX (that is, to *upload* it), log into UNIX, move to the desired directory, and type a **cat** command line that will redirect output to the desired file, like this:

```
$ cat > micro.text
```

Then use the feature in your communications software that sends out the contents of a disk file, which should also be displayed on your screen. After all text has been sent, type (CTRL-D) to tell **cat** that you have no more text.

When you upload text to UNIX in this way, you may end up with undesired double-spacing. If this happens, don't worry. There's a **C** program presented in Chapter 15, "Programming with C," that can help you. With a slight modification, this program will remove the unwanted blank lines for you. (The slight modification is to change `max` from `2` to `1` in line 5.)

## DISPLAYING TEXT A SCREEN AT A TIME

**pg**
**more**

Variations of the **cat** command, **pg** in UNIX and **more** in XENIX, also display text on your screen, but instead of letting the text race past you, pause after each screenful, displaying something like this on the bottom line:

```
--More-- (6%) _
```

Now you have the following four choices:

- Press the (RETURN) key to display one more line
- Press the space bar to see the next screenful

- Type ***/text*** to search for **text**
- Press the DEL key to exit.

### Displaying part of a file

**head**
**tail**

A pair of commands, **head** (XENIX) and **tail** (UNIX and XENIX), allow you to display the beginning (or end) of a file. Unless you request otherwise, the amount displayed will be ten lines. However, you can also request another amount; with **tail** only, you can request an amount in characters (**c**), lines (**l**), or blocks (**b**). (A *block* in System V is 1,024 characters.) For example, to see only the last 17 characters of **enter**, you could use this:

```
$ tail -17c enter
the cat command.
$ _
```

## 4.5 More on working with files

### Determining the type of a file

**file**

Sometimes we lose track of our files. We look at the directory and ask ourselves, "What's **enter**? When did I create a file called **enter**?" Most UNIX systems have a command called **file** that determines (or at least tries to determine) the general type of a file, which may give you some information about it very quickly. Here are three brief examples:

```
$ file enter
enter: English text
$ _
```

```
$ file ch.set_6
ch.set_6: ascii text
$ _
```

```
$ file FOCUS
FOCUS: commands text
$ _
```

This is not an enormous amount of information about your file (and it isn't always correct), but it's a start.

## Sorting a file **sort**

Sorting allows you to put lines of text in order. It is probably the most common form of text processing. For example, suppose you have used **cat** to enter two files named **fruits** and **animals** with the contents shown:

```
$ cat > fruits                  $ cat > animals
bananas                         horses
oranges                         cats
apples                          dogs
cherries                        birds
pears                           lizards
CTRL-D                              CTRL-D
$ _                             $ _
```

To sort these files and display the results, use the **sort** command:

```
$ sort fruits                   $ sort animals
apples                          birds
bananas                         cats
cherries                        dogs
oranges                         horses
pears                           lizards
$ _                             $ _
```

As mentioned earlier, you can also sort the list of system users and display it on the screen this way:

```
$ who | sort
dave       tty11   Feb  2 15:56
elaine     tty04   Feb  3 09:58
manny      tty07   Feb  3 08:05
$ _
```

The **sort** command will be described in detail in Chapter 13, "Searching and Sorting."

## Dealing with repeated lines **uniq**

Let's take an *extremely* simple example of a common task that is often necessary after you've sorted a list of items. Suppose that, for some strange reason, you decide to keep a list of the different animals that show up in your yard over a period of time. Of course, you keep this list in a file on your UNIX system.

At the end of one eventful day, you rush to your terminal and add the following to your **animals** file:

```
$ cat >> animals    [Remember, two symbols (>>) to add text]
birds
raccoons
dogs
birds
```

```
cats
                         [Press CTRL-D—nothing will be displayed]
$ _
```

Now comes the moment you've been waiting for: it's time to sort your list. Use the **sort** command, and get the list on the left. If you don't want all those repeated items, you can pipe your list to a command called **uniq** to get the new list on the right.

```
$ sort animals
birds
birds                          $ sort animals | uniq
birds                          birds
cats                           cats
cats                           dogs
dogs                           horses
dogs                           lizards
horses                         raccoons
lizards                        $ _
raccoons
$ _
```

As you can see, the **uniq** command in its plainest form condenses a *sorted list* by listing each item only once, no matter how many times it appears in the original list. (You could also use **sort -u**.)

The **uniq** command also has several options, one of which is to precede each item in the final list with the number of times it appeared in the original list (**-c**). Here is an example of **uniq** with the **-c** option:

```
$ sort animals | uniq -c
   3 birds
   2 cats
   2 dogs
   1 horses
   1 lizards
   1 raccoons
$ _
```

## Preparing text for processing **prep**

If you have a file with ordinary text in sentences and paragraphs and you would like to study the individual words in the file, you may be able to use the XENIX **prep** command to list each word in a separate line. Here is an example, using a new file called **words**:

```
$ cat > words
The more you study, the more you learn.
                  CTRL-D
$ prep words
the
more
```

```
you
study
the
more
you
learn
$ _
```

You could also sort this list and use **uniq** to determine how many times each word occurs in the file. First place each word on a separate line with **prep**; sort the list with **sort**; then get a count by word using **uniq** using the **-c** (count) option:

```
$ prep words | sort | uniq -c
1 learn
2 more
1 study
2 the
2 you
$ _
```

## Counting lines, words, and characters in a file **wc**

Sometimes you need to have a few statistics on a file. If you're writing a program, you may want to know how many lines of code you have. If you're writing an article, you may want to know how many words you've written. The **wc** (word count) command tells you how many lines, words, and characters there are in a file (in that order). Words are assumed to be separated by either punctuation marks, spaces, tabs, or newlines. To obtain a complete count for **enter**, enter the following:

```
$ wc enter
    3      15      69  enter
$ _
```

If you would rather have one of the three numbers given by itself, you can use one of these options with the **wc** command:

**-l** (lines only)
**-w** (words only)
**-c** (character only)

Here are examples of commands to obtain each of the separate counts:

```
$ wc -l enter
       3 enter     [There are 3 lines in enter]
$ wc -w enter
      15 enter     [There are 15 words in enter]
$ wc -c enter
```

```
     69 enter     [There are 69 character in enter]
$ _
```

## SEARCHING FOR A PATTERN IN A FILE grep

To find a key word or phrase in a file, you can use the **grep** command. In UNIX, any sequence of characters that you are looking for is called either a *search pattern* or a *regular expression.* (If you're curious about such things, **grep** is an acronym for "globally find regular expressions and print"—more or less. They didn't like **gfreap,** so they called it **grep.** Now you know.) For example, to display all lines in **enter** that contained the letters `cat`, you could use a command like this:

```
$ grep cat enter
with the cat command.          [There was only one line.]
$ _
```

If you wanted to find out who was logged in on terminal **tty15**, you could use a command like this:

```
$ who | grep tty15
manny     tty15   Feb  3 10:05
$ _
```

This command illustrates piping the result from a command to another instance of the same command. The **grep** command will be discussed in greater detail in Chapter 13, "Searching and Sorting."

## FINDING MISSPELLED WORDS spell

With the UNIX **spell** command, you can check spelling by comparing words in a file against entries in a large on-line dictionary. For example, suppose you have a file named **lines** that contains the following text:

```
$ cat > lines
Now is the tyme for all good men to
come to the ade of there country.
                  [CTRL-D]
$ _
```

To check this file, use this command line:

```
$ spell lines
tyme
ade
$ _
```

Note that "there" isn't listed, even though it's incorrect. The **spell** program can determine whether a word is misspelled, but it can't determine whether or not the word is used correctly.

With **spell -v**, you can also list words that may be derived from words listed in the dictionary, such as `carefully` (from `careful`), `people's` (from `people`), `listening` (from `listen`), and so on.

## Comparing files line by line **diff**

There are many times when you find yourself with one copy of a file and also a modified version in the same directory. "Which one is which? Are they the same?" you ask yourself. One UNIX command that can help you is **diff**. To show how it works, let's create a simple file called **wildlife.1**, then modify it to create a second file called **wildlife.2**:

```
$ cat > wildlife.1              $ cat > wildlife.2
1 antelope                      1 antelope
2 bear                          2 buffalo
3 coyote                        3 coyote
4 deer                          4 elk
5 elk                           5 fox
CTRL-D                          CTRL-D
```

Comparing these two files with **diff**, we get the output shown below, which indicates how the second file differs from the first:

```
$ diff wildlife.1 wildlife.2
2c2                [Line 2 has been changed:
< 2 bear           • The first file (<) contains 2 bear.
---
> 2 buffalo        • The second file (>) contains 2 buffalo.]
4,5c4,5            [Lines 4 and 5 have been changed:
< 4 deer           • Line 4 of the first file is 4 deer.
< 5 elk            • Line 5 of the first file is 5 elk.
---
> 4 elk            • Line 4 of the second file is 4 elk.
> 5 fox            • Line 5 of the second file is 5 fox.]
$ _
```

The **diff** program uses `c` to indicate a change, `d` to indicate a deletion, and `a` to indicate an addition, along with < to indicate a line in the first file and > to indicate a line in the second file. The hyphens (–) are inserted when a line from the first file is compared with a line from the second.

The **diff** command also has several options, such as ignoring blank spaces in the two files (**-b**) and providing a list of the **ed** commands required to change the first file to the second (**-e**). But we'll have to move on now to another command for comparing files.

## Displaying lines common to two files **comm**

To obtain a different view of the differences between two files, you can use the **comm** (common) command. This command displays three columns on the screen, one for lines unique to the first file, a second for lines unique to the second, and a third for lines common to the two files. Here is how the output for **comm** would look for wildlife.1 and wildlife.2:

```
$ comm wildlife.1 wildlife.2
                 1 antelope     [Common to both files]
2 bear                          [First file only]
        2 buffalo               [Second file only]
                 3 coyote       [Common to both files]
4 deer                          [First file only]
        4 elk                   [Second file only]
5 elk                           [First file only]
        5 fox                   [Second file only]
$ _
```

As you can see, the three columns give you a graphic display that is easy to read. (If you look closely, though, you see that **comm** made a mistake: it first listed `elk` as being unique to wildlife.2, then as being unique to wildlife.1.) The **comm** command also allows you to suppress a column (or a pair of columns), using **-1** to suppress the first column, **-2** to suppress the second column, **-3** to suppress the third column, and combinations like **-13** to suppress the first and third columns. Here is how you could select lines common to both files (third column) and store them in a file called common:

```
$ comm -12 wildlife.1 wildlife.2 > common
$ cat common
1 antelope                      [Common to both files]
3 coyote                        [Common to both files]
                                [We missed elk again]
$ _
```

The option **-12** in the above example told **comm** to suppress the first and second columns (that is, to output only the third column).

For comparison of program files, you may also be interested in the **cmp** command, which compares two files *character by character* (*byte by byte*).

## Translating characters **tr**

There may be times when you will want to change certain characters every place they occur in a file. For example, suppose you have used numbers to indicate the steps of a procedure you have written. Then you decide (or possibly you receive a request) to use letters of the alphabet instead. With the **tr** command, you can make the change fairly easily. Here's a file:

```
$ cat > five.steps
1. Turn on the machine.
2. Start the program.
3. Request the P option.
4. End the program.
5. Turn off the machine.
                  [CTRL-D]
$ _
```

To make the translation described, type the command (**tr**), a space, the set of characters to be translated, another space, then the set of characters into which to translate the original characters (in the order desired), using redirection to accept the input from **five.steps**:

```
$ tr 12345 abcde < five.steps
a. Turn on the machine.
b. Start the program.
c. Request the P option.
d. End the program.
e. Turn off the machine.
$ _
```

This **tr** command translated 1 into a, 2 into b, 3 into c, and so on. You can also use range notation to enter the command this way to accomplish the same thing:

```
$ tr 1-5 a-e < five.steps
a. Turn on the machine.
b. Start the program.
c. Request the P option.
d. End the program.
e. Turn off the machine.
$ _
```

For larger sets of characters, place the ranges within pairs of brackets (and then place the brackets within double quotation marks to keep them from being interpreted as members of the sets of characters), as shown here:

```
$ tr "[a-z]" "[A-Z]" < five.steps
1. TURN ON THE MACHINE.
2. START THE PROGRAM.
3. REQUEST THE P OPTION.
4. END THE PROGRAM.
5. TURN OFF THE MACHINE.
$ _
```

We can also use the **tr** command to delete characters by including the **-d** (delete) option. For example, we could use the following command to delete the step numbers:

```
$ tr -d 12345 < five.steps
. Turn on the machine.
. Start the program.
. Request the P option.
. End the program.
. Turn off the machine.
$ _
```

On the other hand, if we wanted to delete the explanations and just leave the step numbers, we could also include the **-c** (complement) option to delete everything *except* the numbers:

```
$ tr -cd 12345 < five.steps
12345$ _
```

The reason that the numbers (and the prompt) appear together on the same line is that the characters that separate lines from each other (called *newlines* in UNIX) have been deleted along with the visible characters. Here's a similar command line. See if you can explain what is happening here:

```
12345$ tr -cd "[A-Z]" < five.steps
TSRPET$ _
```

In the two examples above, the two options **c** and **d** were placed together behind the same minus sign. In UNIX, this is referred to as *bundling* options. Some UNIX commands allow bundling of options, some do not.

## 4.6 Using printers

You can use the UNIX command **lp** to print the contents of a file on the system lineprinter. (Actually, the file is placed in a queue, and, depending on system demand for printing, won't necessarily be printed immediately.) For example, to print the contents of a file called **section_4**, you could use

```
$ lp section_4
request id is mx80-217 (1 file)
$ _
```

The message from **lp** tells you that this is the 217th request for printing on printer mx80. If you decide for some reason to cancel printing, you can use the job number after a **cancel** command, like this:

```
$ cancel mx80-217
$ _
```

## Variations of the command **lp**

Using some of the techniques you learned earlier in this chapter, you can do more than just print one file on the lineprinter. You can also print a series of files like this:

```
$ lp section_1 section_2 section_3
request id is lx1000-346 (3 files)
$ _
```

By piping text from **pr** to **lp**, you can take advantage of some the formatting options of **pr** before starting to print. For example, to paginate text and arrange it in two columns before printing, you could use

```
$ pr -2 text_5 | lp
request id is fx86e-297 (standard input)
$ _
```

Using two pipes, one from **sort** to **pr** and another from **pr** to **lp**, you could sort and paginate lines of text in a file before printing:

```
$ sort data_15 | pr | lp
request id is lx1000-408 (standard input)
$ _
```

## Options for the command **lp**

Two of the options available for the **lp** command can be helpful. The **-c** (copy) option makes a copy of the text to be printed, as a precaution against loss of the text in the queue like this:

```
$ lp -c section_5
request id is mx80-329 (1 file)
$ _
```

The **-m** (mail) option reports to you by mail when your printing job has completed like this:

```
$ lp -m section_5
request id is lx1000-453 (1 file)
$ _
```

### Finding out what is queued for printing **lpstat**

In a multi-user system like UNIX, everyone has to share the system's lineprinter. So any time you initiate the **lp** command to print a file, **lp** places the name of the file in a queue. It's just like waiting in line at a bank. The names of the files move through the queue as the files are printed; then they are removed one by one as each file leaves the queue.

The queue is kept in a UNIX directory, whose name is often **/usr/spool/lpd**. To find out which files are currently in the queue for printing, all you have to do is to display the contents of this directory on your screen, using the **lpstat** (line printer statistics) command:

```
$ lpstat
total 28
mx80-217       robin       17462    Apr 6 09:09 on mx80
lx1000-346     robin        3685    Apr 6 09:12
fx86e-297      robin        8931    Apr 6 09:13
lx1000-408     robin        2366    Apr 6 09:17
mx80-329       robin        6328    Apr 6 09:19
lx1000-453     robin       23697    Apr 6 09:21
$ _
```

## 4.7 Summary

In this chapter you have learned about command lines, redirection and pipelines, and commands for displaying text, working with files, and using lineprinters. To begin a *process* under UNIX, type a command line and press RETURN. The command may contain either just the command by itself or the command plus modifiers.

### Redirection and pipelines

Unless you instruct UNIX otherwise, UNIX regards your keyboard as its *standard input* and your video screen as its *standard output*. But you can *redirect* either input or output elsewhere. To have a UNIX command take its input from a file, instead of from your keyboard, use the *less than* sign (<) in front of the name of the file.

To have a UNIX command send its output to a file, instead of to your video screen, use the *greater than* sign (>) in front of the name of the file. If the file does not exist, it will be created; if it does exist, it will be overwritten. To *append* the contents of the output file, rather than overwrite the contents of the file, use a pair of *greater than* signs (>>) in front of the name of the file.

To connect two processes, so that the output of one becomes the input of the other, use the vertical bar ( | ) between the names of the commands (with or without surrounding spaces).

## Displaying text on the screen

You can use the **cat** command for entering text into a file, downloading to your microcomputer, or uploading to UNIX. You can use **pg** (UNIX) or **more** (XENIX) to display text one screenful at a time, **head** (XENIX) to display only the opening lines of the text, or **tail** (UNIX and XENIX) to display only the closing lines.

## More on working with files

To determine the type of text in a file, you can use the **file** command. To sort a file by lines, use the **sort** command, followed by the name of the file. As with many other commands, the results of the **sort** command can also be redirected to a new file. In addition, **sort** can receive its input from the standard input (the keyboard) or another command via a pipe. To eliminate repeated lines in a sorted list, use the **uniq** command. One option of this command (**-c**) also gives you an item count.

To have each word of text in a file placed on a separate line for further processing, use the **prep** command (XENIX only). To obtain the total number of lines, words, and characters in a file, use the **wc** command either by itself to display all three statistics or with one of three options (**-l**, **-w**, **-c**) to display one. (Combinations such as **-lw** and **-wc** are also allowed.)

To search for a pattern in a file, use the **grep** command, followed by the pattern and then the name of the file. You can also pipe input from another command (in which case the filename becomes unnecessary). To see a list of misspelled words in a file, use the **spell** command, followed by the name of the file.

To compare the contents of two files line by line, use the **diff** command. (There is also a program called **diff3** that compares three files at a time and another called **cmp** that compares two files character by character.) To determine which lines are common and which are unique to each of two files, use the **comm** (common) command. To translate one set of characters into another set, use the **tr** command. You can also use this command to delete characters.

## Using lineprinters

To have a file sent to the system's lineprinter to be printed, use the **lp** command, followed by the name of the file(s). You can also use pipes with **lp** to format the output before printing it. The **-c** (copy) option informs **lp** to make a copy before printing; the **-m** (mail) option informs **lp** to notify you by mail when printing has completed. To find out which files have been queued for printing on the lineprinter, use the **lpstat** command. To cancel a printing request, use the **cancel** command with the printing request number.

# 5

# Communication in UNIX

In this chapter you will learn how to communicate within and outside of UNIX. We'll start by communicating with other users on your own system, then discuss communicating with other UNIX systems.

## 5.1 Communicating with other users

To allow various forms of communication between users on the same system, UNIX provides three facilities: electronic mail, an automatic reminder service, and direct messages.

### SENDING MAIL **mail**

To send electronic mail to any user on your UNIX system, use the **mail** command. Just type **mail**, a space, the other user's identifier, and press RETURN. For example, to send a message to Janice, you could use

```
$ mail janice
```

After typing what you want to say, press CTRL-D to conclude your message. You can also address a message to several different users at the same time. Here is an example:

```
$ mail ralph laura peter sara

There will be a meeting at 2:00 pm today to review
the design of the Rattlesnake.  Bring the specifications
you received from Engineering last week.
CTRL-D
$ _
```

Each recipient will be notified of the existence of this message when he or she logs on.

## RECEIVING MAIL **mail**

If another user has sent mail to you, the system will store it in a file with your login name in directory **/usr/mail**, and the next time you log on you will see the following message:

```
You have mail
```

This is known as the "You have mail" message. To find out what has arrived, type the **mail** command without any arguments. The system will display the most recent message on the screen, give you a question mark prompt (?), and wait for you to indicate what you want to do with the message.

```
$ mail
From paul Tue Jan 23 09:24:17 1989
Yes, I got your message.  Let's meet
for lunch.  What time can you make it?
? _
```

You now have the option of saving or deleting this message, going on the next next message, or returning to the shell prompt. Suppose you decide to return to the shell prompt and reply to this message. Here are the steps:

```
? q
$ mail paul
We should be able to leave at 11:30
CTRL-D
$ _
```

If, on the other hand, you wanted to save this message in another file (**lunch**), you could type this instead:

```
? s lunch
From gina Tue Jan 23 09:13:42 1989
I need to see you sometime today.
Let me know when you're free.
? _
```

When you saved the first message, the system automatically displayed (printed) the next one. As long as you stay in the **mail** session, the system will continue to display additional messages. At any time, you can type one of the following to perform the action indicated:

| | |
|---|---|
| *** (or ?)** | List all the **mail** commands |
| RETURN | Display the next message |
| **p** | Redisplay (print) the current message |
| **d** | Delete the current message |

| | |
|---|---|
| **m** *user* | Forward the current message to *user* |
| **s** [*file*] | Save the current message (with header) in *file* (file **mbox** in the current directory if you omit *file*) |
| **w** [*file*] | Save the current message (without header) in *file* (same default) |
| **!** *command* | Execute *command* without leaving **mail** |
| **x** | Exit **mail**, leaving all messages intact |
| **q** | Quit **mail**, leaving only unexamined messages intact |

One way to set up a reminder system is to use the **mail** command to send messages to yourself. Here is an example:

```
$ mail robin
Avoid falling asleep when Wally starts talking about his
pet frog, avoid unnecessary shouting, and avoid loud,
senseless arguments.
CTRL-D
$ _
```

To view mail in a file other than **/usr/mail/robin**, use the **-f** option with the name of the alternate file, like this:

```
$ mail -f lunch
From paul Tue Jan 23 09:24:17 1989
Yes, I got your message.  Let's meet
for lunch.  What time can you make it?
? _
```

To reverse the order in which messages are displayed (first received is first displayed), use the **-r** option:

```
$ mail -r
From will Tue Jan 23 08:31:25 1989
I hope you still have the used VCR
for sale.  I'd like to take a look.
? _
```

## SENDING MAIL **mailx**

If your installation has the **mailx** command, a more sophisticated tool for handling mail, you can use it instead of **mail**. The basic operation is about the same, but there are many more options available in **mailx**. For example, to send a message to several people, you could type this:

```
$ mailx ralph laura peter sara
Subject: _
```

If your system is so configured, you will now see a "Subject" prompt. Type a subject heading, then press RETURN. Then the rest will be the actual message (up to the final CTRL-D).

```
$ mailx ralph laura peter sara
Subject: Rattlesnake Design
There will be a meeting at 2:00 pm today to review
the design of the Rattlesnake.  Bring the specifications
you received from Engineering last week.
CTRL-D
$ _
```

Unlike **mail**, **mailx** allows you to interrupt entry of your message with an ESC to perform other functions. For example, if you suddenly think of other people who should be receiving the message (say Len and Jane), you can add their names to the recipient list by typing this:

```
~t len jane
```

Each escape command consists of a tilde (~), followed by another character (in this case, t). Here are some of the other commands you can use:

| | |
|---|---|
| **~?** | List all the escape commands |
| **~r** *file* | Read text into your message from another file |
| **~w** *file* | Write your message to another file |
| **~s** *subject* | Set the subject heading |
| **~t** *users* | Add *users* to the "To" list |
| **~c** *users* | Add *users* to the "Copy" list |
| **~h** | Prompt yourself for "To" list, subject heading, and "Copy" list |
| **~v** | Invoke the visual editor (described in Part II) to modify your message |
| **~p** | Print (display) the current message |
| **~m** *messages* | Read in other messages, indented to the first tab stop |
| **~f** *messages* | Read in other messages, without indentation |
| **~!** *command* | Run a UNIX command without leaving **mailx** |
| **~\|** *command* | Pipe the message through a UNIX command |
| **~q** | Quit; save message in file **dead.letter** in your home directory |
| **~x** | Exit without saving the message |

## RECEIVING MAIL **mailx**

To review mail sent to you, type **mailx** without any names. Unlike **mail**, **mailx** will display a summary of the mail that has been sent to you, with a one-line entry for each individual message (called a *header*) and a pointer

(>) to the current message. The first letter indicates whether the message is new (N), read (R), or unread (U). Again, the system gives you a question mark prompt for your response, as shown here:

```
$ mailx
"/usr/mail/robin": 3 messages 2 new 1 unread
 U  1 will     Tue Jan 23 08:31   2/69  VCR for sale
 N  2 gina     Tue Jan 23 09:13   2/62
>N  3 paul     Tue Jan 23 09:24   2/72  Lunch today
? _
```

The choices for **mailx** are similar to those for **mail**. The command list shows the command names spelled out, but you need only enter the first letter. By default, *msglist* in the list that follows is simply the current message. But you can redefine *msglist* to specify messages by number, sender, subject, or type. Here are some of the commands:

| | |
|---|---|
| **?** | List all the commands with explanations |
| **list** | List all the commands without explanations |
| **type** [*msglist*] | Display the message(s) |
| **next** | Display the next message |
| **top** [*msglist*] | Display the first five lines of messages |
| **from** [*msglist*] | Display header(s) for message(s) |
| **header** | Display active message headers |
| **z [-]** | Display the next [or last] page of headers |
| **save** [*msglist*] *file* | Save (append) the message(s) to *file* |
| **delete** [*msglist*] | Delete the message(s) |
| **undelete** [*msglist*] | Restore deleted message(s) |
| **preserve** [*msglist*] | Preserve message(s) in **mbox** |
| **Reply** [*msglist*] | Reply to the sender(s) of the message(s) |
| **reply** [*msglist*] | Reply to the sender(s) of the message(s) and also to other recipients |
| **edit** [*msglist*] | Edit message(s) |
| **cd** [*directory*] | Change to *directory* (home if none named) |
| ! *command* | Execute UNIX command |
| **quit** | Quit (preserving only unread messages in mbox) |
| **xit** | Exit (preserving all messages in mbox) |

The **mailx** command also allows you to view mail in a file other than /usr/mail/robin, using the same **-f** option:

```
$ mailx -f lunch Read mail in file /usr/robin/lunch

$ mailx -f       Read mail in file /usr/robin/mbox
```

## Automatic reminder service **calendar**

Another way to remind yourself of events is to use the automatic reminder service that UNIX provides with the **calendar** command. Every day or so, UNIX uses the **calendar** command to examine each user's home directory for a file named **calendar**, whose contents may look something like this:

```
Mar 21    Planning meeting at 9:30 in conference room
Apr 17    Jennifer's wedding in Hendersonville
Apr 28    Awards dinner at the Blackjack Inn
May 3     Presidential primary
```

Other forms of dates, such as `March 21` and `3/21`, are also allowed. The **calendar** command extracts from this file each line that contains either today's date or tomorrow's date and mails it to you. You can also call up the **calendar** manually like this:

```
$ calendar
```

UNIX will search your home directory for a file named **calendar** to look for any pertinent items to display.

## Writing directly to a user **write**

The **write** command allows you to send a message directly to another user's terminal, where it will immediately appear on the screen. Here is an example of initiation of a **write** command, which is similar to sending mail:

```
$ write paul
How are you doing on your project?  o
                [CTRL-D—not displayed on the screen]
$ _
```

Here is what Paul will see on his screen immediately after this command line is initiated:

```
Message from robin (tty07) [Tue Nov 10 15:21:59]...
How are you doing on your project?  o
<EOT>
```

At this point, Paul can respond with a **write** command of his own, and begin a terminal-to-terminal dialogue:

```
$ write robin
It's about two-thirds completed.  How about yours?  o
                [CTRL-D—not displayed on the screen]
$ _
```

To avoid confusion during a dialogue (that is, wondering if the other party is about to say more), you can set up a simple protocol to let the other party know when you have completed your current message. For example, each user could type ***o*** for *over* at the end of each message and ***oo*** for *over and out* at the conclusion of the dialogue.

If you're the kind of fun-loving person who can't resist pulling pranks on other users, then **write** is one command you'll want to add to your repertoire immediately. Unfortunately, however, there is also a command called **mesg** that allows people who are less fun-loving to shut out **write** messages. All they have to do is to add the **n** (no) option, and the fun is over:

```
$ mesg n
$ _
```

To allow **write** messages again, add the **y** (yes) option, and you're back in business:

```
$ mesg y
$ _
```

Finally, to find out whether **write** messages are allowed or prohibited on your terminal at a given moment, type ***mesg*** without an argument:

```
$ mesg
⎧is yes⎫
⎨is no ⎬          [Three responses possible]
⎩error ⎭
$ _
```

## 5.2 Communicating outside your system

UNIX allows you to communicate with someone outside your UNIX system with two different commands: **cu** (call up) and **uucp** (UNIX-to-UNIX copy). We'll take them one at a time.

### Calling outside your UNIX system **cu**

The **cu** (call up) command allows you to dial a telephone number and call up one of the following:

- another UNIX system
- a terminal
- a non-UNIX system

Once a connection is established, you can carry on an interactive conversation (as with **write** on your own system) and possibly transfer files back and forth. If the telephone number of the other system is 345-6000 and both sides are set up to communicate at a speed of 1200 bit/s, depending on the type of connection, you could type something like this:

```
$ cu 3456000 -s 1200
```

With connection established, you will probably see a login message:

```
login: _
```

You can now log into the other system as you would into your own. Having logged in, you can use **cu** to send a file called message to the other side with a command like this:

```
~<message
```

If you are connected to another UNIX system, the **stty** (set teletype [terminal]) command on both systems specifies the same characters for *erase* and *kill*, and the **cat** command is active on both systems, another way to send message to the other side is to type a command like this:

```
~%put message
```

If you are connected to another UNIX system, the **echo** and **tee** commands are active on both systems, and directory permissions allow it, you can copy a file on the other to your working directory with a command like this:

```
~%take reply
```

To conclude your conversation, type the following:

## UNIX-TO-UNIX COPYING **uucp**

In Chapter 3, "The UNIX File System," you learned about the **cp** (copy) command that you can use to make a copy of a file either in your working directory or in another directory. There is a similar command called **uucp** (UNIX-to-UNIX copy) that you can use to copy a file to or from another UNIX system. For example, if two XENIX microcomputers called ucb/cat/fish and ucb/moon/dog both belong to a common network, then a user can execute a command like this from ucb/cat/fish:

```
$ uucp latest ucb/moon/dog!/usr/robin/news
$ _
```

This command will take the contents of a file called /usr/leslie/latest on ucb/cat/fish (assuming that /usr/leslie is the current directory) and make a copy called /usr/robin/news on ucb/moon/dog. An exclamation point ( ! ) separates the name of the microcomputer from the name of the file. Otherwise, **uucp** is quite similar to **cp**. Note that if you're using the C shell, you have to type a backslash ( \ ) in front of each exclamation point, as shown here:

```
% uucp latest ucb/moon/dog\!/usr/robin/news
% _
```

If you don't know the exact name of the recipient's home directory, you can use a tilde (~) in front of the recipient's user name to have **uucp** search for the directory, as shown in another example typical of XENIX usage. With UNIX, it's more customary to use a public directory (/usr/spool/uucppublic).

```
$ uucp latest ucb/moon/dog!~robin/news
$ _
```

To have **uucp** mail the originator a message after the copy has been made, include the **-m** option in the command line, as shown here:

```
$ uucp -m latest ucb/moon/dog!~robin/news
$ _
```

To have **uucp** also mail the recipient a message after the copy has been made, add the **-n** option (with the recipient's user name appended), as shown here:

```
$ uucp -m -nrobin latest ucb/moon/dog!~robin/news
$ _
```

In the following example, a user logged into ucb/moon/dog transfers all files whose names end in the suffix .c to a directory named /usr/leslie on ucb/cat/fish. (Since source programs in the **c** language have names like list.c, post.c, enter.c, and so on, this user is sending all the C programs in the user's working directory.)

```
$ uucp *.c ucb/cat/fish!/usr/leslie
$ _
```

Since the user who executed this command was logged into **ucb/moon/dog** when the command was executed, it wasn't necessary for the user to type ***ucb/moon/dog!***. However, **ucb/moon/dog!*.c** also would have been correct in place of ***.c**.

Actually, **uucp** is just one of a family of UNIX commands. Another member of the family is **uux** (UNIX-to-UNIX execute), which can be used to execute commands on another computer in the network. For example, suppose the system administrator has set up files to allow the **lp** command to execute. Then to format the contents of a file called **raw.text** and then have it printed on **ucb/moon/dog**'s printer, you could execute a command like this from another computer in the network:

```
$ pr -2 raw.text | uux - ucb/moon/dog!lp
$ _
```

Except for the **uux** command, the hyphen, the name of the other computer, and the exclamation point (**uux - ucb/moon/dog!**), this command is quite similar to a command for performing the same function on your own system.

Although it is quite simple to execute commands like these once a network has already been set up, the task of setting up the original network is much more complex. This is discussed for Part VII. For now, at least you are now aware of these programs and some of their possible uses. Note that the **uucp** network, with over 3,000 sites, is tied into other networks, such as ARPANET, and allows transfers to Ethernet installations.

Note that the rules for naming nodes on a network are similar to the rules for naming files in a file system. Unless your recipient's node is "below" yours in the network, you must use the recipient's full address.

## ENCRYPTING INFORMATION **crypt**

If you have sensitive information that you would like to keep secret, you may want to consider encrypting certain files with the UNIX **crypt** command. In general, an encryption program takes the original text that you provide (the *cleartext*), transforms it with a sequence of characters (the *key* or *password*), and produces an encrypted version of the original (the *cyphertext*). You will have to remember the key (or write it down on a slip of paper) to be able to retrieve the cleartext at a later time.

## ENCRYPTING A FILE

First we need a file that contains text. Let's use the **cat** command to enter the following. (I know, you're going to say that this text has already been encrypted.)

```
$ cat > remark
```

```
"I would not like to make a value judgment on that
other than to say that I have no comment."
   [Four tabs]                           ---Alexander Haig
   [(CTRL-D) to terminate text]
$ _
```

Now use the **crypt** command to perform the actual encryption, to which UNIX will respond immediately with a request for the key. We'll use *convolution.*

```
$ crypt < remark > remark.crypt
Enter key: convolution [You won't see the key on the screen]
```

Redirection is used on this command line for both input and output. The *crypt* command takes its input from the file you just typed (**remark**), then sends its output to a file called **remark.crypt**.

The next logical step is to remove the original file (**remark**); otherwise, it doesn't make much sense to have it encrypted:

```
$ rm remark
$ _
```

Don't let curiosity get the better of you by taking a peek at **remark.crypt**. You'll just cause trouble for your terminal, which will probably beep and go blank trying to read the various things in the file.

## Viewing the Cleartext

To view the cleartext on the screen at a later time, use the **crypt** command again with the same key:

```
$ crypt < remark.crypt
Enter key: convolution [The key won't appear on the screen]
"I would not like to make a value judgment on that
other than to say that I have no comment."
                                         ---Alexander Haig
$ _
```

## Printing the Cleartext

To print the cleartext on the system's lineprinter at a later time, use the **crypt** command with the same key, sending the output through **pr** to **lp** via pipelines:

```
$ crypt < remark.crypt | pr | lp
Enter key: convolution [The key won't appear on the screen]
$ _
```

### Comments on encrypting files

The security of an encrypted file depends to a large degree on the invulnerability of the key that you select. For our simple example here, we have not really made a very good choice. If you really want your encrypted files to be secure, you should select a complex string of characters that cannot be readily determined. As in composing a good password, you can use your imagination to produce something easy to remember, like these:

```
Why_4GET:it?

U2:can|B,1st!
```

It goes without saying, but we'll say it anyway: If you're concerned about your encrypted files, don't store your key anywhere on the UNIX system. Either memorize it or write it on an unmarked slip of paper.

Both of the major UNIX text editors (**ed** and **vi**) have features for handling encrypted files.

## 5.3 Some basics of communication

If you drive a car, it's very difficult to avoid terms like *disc brakes*, *rack-and-pinion steering*, *turbo-charger*, and so on. Likewise, as soon as you start using a computer, you are confronted with new terms. For many people, the most perplexing terms seem to be related to communication. For those interested, here are a few basic concepts.

### Units of information

The smallest amount of information that a computer handles is a binary digit (or *bit*), which can be one of two things: 1 or 0, in the language of the software engineers who write programs; high or low, in the language of the hardware engineers who design the machinery. Eight bits form a *byte*, which is the unit by which information is usually stored in a computer. A byte corresponds to one character (of text, of data, or of program code).

Inside the computer, information is sent in groups of bits, depending on the machine's design capacity. For most microcomputers, information is sent in groups of 8, 16, or 32 bits at a time. Outside the computer, information may be sent either in groups of bits (*parallel* transmission) or one bit at a time (*serial* transmission).

### Communicating with local devices

Every computer spends a considerable percentage of its time sending messages back and forth to various pieces of equipment. From the computer's point of view, the disk drives, printers, and terminals connected to it are external, or *peripheral*, devices. There must be something to connect them, and there must be a common method for exchanging messages.

Any device in the same building can usually be connected to a computer with a cable (either parallel or serial). To make things a little easier for everyone, most computers and peripheral devices have plugs for widely accepted types of cable connectors. One type of serial connector commonly used is called an *RS-232C* connector. The RS stands for "recommended standard," and *232C* is the designator that some committee came up with. Your printer, terminal, and modems may all be attached with these cable connectors.

A newer *RS-422/423* standard is beginning to overtake *RS-232C* in serial transmission as we approach the late 1980s. RS-422/423 offers more connectors (37 instead of 25) and better control.

## COMMUNICATING WITH REMOTE DEVICES

A computer can also be connected with another computer or a terminal at another location. However, since most people don't have miles of cables lying around (and since the cable wouldn't be able to carry signals far enough anyway), something else has to be used. That something is the vast network of cables used to handle telephone service.

There's one slight problem, however. Computers and telephones don't use the same kind of signals. Computers use *digital* signals, while telephones use *analog* signals. To see what this means, compare a watch with number displays to a watch with moving hands. The watch with number displays uses *digital* signals to tell us the time; the watch with moving hands uses *analog* signals.

We have a solution to the the problem of differences between computers and telephones: an electronic device that converts digital signals to analog (modulator) and also converts analog signals back to digital (demodulator). This modulator/demodulator is usually called a *modem* for short (or a *data set*). According to some predictions, around the beginning of the 21st century, a large part of the telephone system will have been converted to digital operation, and modems will be obsolete.

## COMPATIBILITY

Whether a cable or a telephone is used to connect them, a computer and another device must be in agreement about a number of things before they can start sending messages back and forth to each other: they have to be sending and receiving by the same timing method, at the same speed, with the same protocol, in the same duplex mode, and in the same coding system. Let's briefly consider each of these.

## Timing method

One problem a computer must solve when it sends information back and forth is how to determine where a character (or byte) in transit begins and ends. One solution, usually used by larger computers, is to have both sides send messages back and forth to synchronize the transmission, and then release a continuous stream of information. This is called *synchronous* communication.

Another method, usually used by small computers and printers, is to bracket each individual byte between a pair of bits (a *start bit* and a *stop bit*). This method, used where characters are usually sent sporadically and at irregular intervals, is called *asynchronous* (or *start-stop*) communication.

Another term used here relates to error-checking during transmission. An extra bit is often added to the character, start, and stop bits (the *parity bit*). The object is to make the total number of bits either even (*even parity*) or odd (*odd parity*).

## Data rate

*Data rate* (often referred to as *baud rate*) is the speed at which a computer sends or receives information. This speed is measured in bits per second (bit/s), and is sometimes classed as *slow* (110, 150, 300, or 600 bit/s), *medium* (1200, 1800, 2400, 3600, or 4800 bit/s), or *high speed* (9600 and 19,200 bit/s). A data rate of 2400 is typical for today's microcomputers.

## Protocol

Communication requires a set of rules to determine which side is supposed to send information and when. On larger computers, such a set of rules is called a *protocol* (or *line discipline*), and is based on synchronous transmission. Such protocols are classed as either *byte-synchronous* or *bit-synchronous.* One byte-synchronous protocol, IBM's BSC (binary synchronous communication), uses timing signals from the sending and receiving sides to synchronize groups of characters being transmitted. One bit-synchronous protocol, IBM's SDLC (synchronous data link control), relies on standard data formats for synchronization.

On smaller, asynchronous computers, the term *handshaking* is usually used instead of *protocol.* One of the most common handshaking methods is called *XON/XOFF.* By this method, the sending side continues to send information until a temporary storage area (or *buffer*) on the receiving side approaches capacity. The recipient sends an XOFF signal to the sending side to halt transmission momentarily. Then, when the recipient's buffer gets low, it sends an XON signal to the sending side asking for more.

System V, Release 3 implements features that standardize communication to and from a UNIX system. Consequently, programs like **cu** and

**uucp**, which operate between networked UNIX systems, are now independent of protocols and communication media. Beginning with Release 3, applications, protocols, and media are separated from each other in different *layers.* This makes it possible for UNIX systems to be connected to a larger number of other systems, both UNIX and non-UNIX. It also makes it possible for users to access files outside their own UNIX system. For more information, see Part VII and Appendix M, "UNIX versus XENIX".

## Duplex mode

The directional capability of the line connecting two commmunicating parties can be classified as follows:

*simplex* transmission is possible in one direction only (similar to a one-way street)

*half-duplex* the two sides may take turns sending to each other, but that transmission may take place in only one direction at a time (similar to a street that is one-way south in the mornings and one-way north in the afternoons)

*full-duplex* transmission may take place in both directions at the same time (similar to a two-way street)

In discussing terminals, duplex mode raises another issue: What happens to the characters you type at the keyboard? There are only two choices for a terminal: Process the characters locally at the terminal or send them to the host computer for processing. When a terminal processes only, it is said to be in *block mode*; when it transmits to the host only, it is said to be in *full-duplex mode.* When it does both, it is said to be in *half-duplex mode*; when it does neither, it is said to be *locked.* On the UNIX system, communication with terminals is typically carried out in *full-duplex mode.*

## Coding system

The final topic relates to the way information is coded by a computer. You may have learned Morse code at some time. If so, you know that it is a coding system that assigns one code for each letter of the alphabet. The entire operation of a computer and its related devices (including communication between them) is carried out through codes. Every letter of the alphabet, every number, and every instruction is known to a computer by a code.

Large computers use IBM's EBCDIC (extended binary-coded decimal interchange code), an 8-bit system with a total of 256 codes. Smaller computers use ASCII (American Standard Code for Information Interchange), a 7-bit system with a total of 128 codes. The entire coding system includes all the display characters found on a typewriter-style keyboard (letters, numbers, and symbols), plus a collection of control characters (see "Character Codes," Appendix N). Here is how a typical ASCII code looks when we represent it in binary digits (bits):

1101001

Since this isn't very easy to read, people usually represent numbers like this in a different number system. Our decimal system (base 10) is a little difficult to translate to binary (base 2), so it's customary to use either octal (base 8) or hexadecimal (base 16) to represent ASCII codes. Here's how all four systems look side-by-side for the letter `i`.

| **Binary** | **Octal** | **Hexadecimal** | **Decimal** | **Symbol** |
|---|---|---|---|---|
| 1101001 | 151 | 69 | 105 | `i` |

The five columns above show five different ways to interpret the same character. If you type ***i*** at your keyboard, the computer sees only `1101001` transmitted, although we could write this code in any of the number systems shown.

## THE ASCII TABLE

Individual characters like `i` are customarily arranged in four columns of 32 characters each (as in Table N.1). So far we have focused on one character (`i`). To gain a little more perspective, let's look at a complete row of ASCII characters from a table—the row that contains `i`. We'll assume that this is a four-column ASCII table.

```
HT  011          )  051          I  111          i  151
```

Here we see four entries, each followed by its octal representation:

| | | | |
|---|---|---|---|
| `HT` | (horizontal tab) | octal code 011 | (commonly known as TAB) |
| `)` | (right parenthesis) | octal code 051 | |
| `I` | (uppercase I) | octal code 111 | |
| `i` | (lowercase i) | octal code 151 | |

Note the following about this arrangement of characters:

- Upper case `I` and lower case `i` are listed side-by-side on the same row

- `I`, `i`, and `)` are *display characters* (characters that appear on the screen when you type them)

- `HT` (TAB) is a *control character* (a character that causes action).

To relate this discussion to your keyboard, note two more things about the arrangement of these characters in the ASCII table:

- Pressing the key labeled `I` alone produces lower case `i`
- Pressing `I` and the SHIFT key together produces upper case `I`
- Pressing `I` and the CTRL key together produces `HT` (TAB), which causes the cursor to move across the screen to the next tab stop

HT (TAB) is called a *control character* because it causes an action that controls the way your terminal works. You could say that it's also called by this name because it's the code that results when you press the CTRL key and the `I` key at the same time. Pressing CTRL and `i` together (sometimes referred to as *pressing* CTRL-I) is equivalent to pressing the TAB key.

To view the ASCII table on the UNIX system, you can display the contents of the file /usr/pub/ascii with the **cat** command, like this:

```
$ cat /usr/pub/ascii
```

Unfortunately, the display will be *sideways*, with the control characters at the top of the table and the display characters on the bottom. The display will show 16 rows of eight characters each. It is more common to show an ASCII table with control characters on the left and display characters on the right, as shown in Table N.1.

## Control characters and display characters

Let's conclude this chapter with a few more words about control characters and display characters. Suppose you see this text file displayed on your screen:

```
Odd no.:        7
Even no.:       8
```

Now here's a question: Assuming that you pressed the TAB key before typing ***7*** and ***8*** and you pressed the RETURN key at the end of each line, how many characters are there in this file?

- If you are completely new to computers, you will probably say 17.
- If you know a little about computers, you may say 19.
- The correct answer is 23. What?

Explanation: There are only 17 *visible* characters. But there are also two blank spaces (which bring the total to 19), and there are four control characters, two TAB characters and two *newline* characters (which bring the total to 23). The TAB character, also known as CTRL-I, is HT (hexadecimal code 9); the newline character, also known as CTRL-J, is NL (hex code A).

This is how this file looks to UNIX (using hexadecimal representation):

```
4F  64  64  20  6E  6F  2E  3A   9  37   A  45  76  65
6E  20  6E  6F  2E  3A   9  38   A
```

Translated into recognizable symbols:

```
O   d   d [sp]  n   o   .   :  HT   7  NL   E   v   e
n [sp]  n   o   .   :  HT   8  NL
```

From our point of view, display characters appear on the screen and control characters do not. But to UNIX they are all just characters. It's something like the characters who bring you a television show. There are the characters you see on your screen (the actors) and the others you never see (those who take care of the cameras, costumes, make-up, props, and so on).

## 5.4 Summary

In this chapter you have learned about communication: first communicating with other users on your own system, then communicating with other UNIX systems. The chapter closes with a brief description of communication concepts.

### Communicating with other users

To send electronic mail to another user on your system, use the **mail** command, followed by the desired user identifier(s). Type your message, then type CTRL-D by itself (or a period followed immediately by RETURN) on the line following the last line of your message. If another user has sent you a message via electronic mail, you will see the message "You have mail" the next time you log on. To find out exactly what you have received, use the **mail** command by itself, then press RETURN to look at each individual message. You can also type ***d*** to delete the current message or ***p*** to repeat it.

The **mailx** command is similar to **mail**, but includes a wider variety of options for sending and receiving mail.

To remind yourself of something, you can either mail an electronic message to yourself with **mail** or post messages after dates in a special reminder file called **calendar**, which will be automatically checked by the system. To send a message directly to another user's terminal, use the **write** command followed by your message starting on the next line and then (CTRL-D) to exit the **write** command.

## Communicating outside your system

To dial up another UNIX system, a terminal, or a non-UNIX system, use the **cu** (call UNIX) command, including the telephone number of the other party and the speed at which both of you are sending data to each other. Once you have established a connection, you can type messages to each other at the keyboard and send files back and forth.

To copy files from one UNIX system in a network to another, use the **uucp** (UNIX-to-UNIX copy) command, including the names of the two UNIX systems with the filename(s). To execute UNIX commands on another system in the network, use the **uux** (UNIX-to-UNIX execute) command, receiving the commands via a pipeline. All of this assumes, of course, that someone has already set up such a network in which to use these commands.

To encrypt a file, use the **crypt** command with a key, redirecting input from the original file and redirecting output to the encrypted file. To view (or print) the original text at a later time, use the **crypt** command again with the same key, redirecting input from the encrypted file. To ensure the security of your encrypted files, select a long, complex key, and don't store it anywhere in the UNIX system.

## Basics of communication

Computers communicate with other computers and other equipment through wires (sometimes with the help of modems), using one of several timing methods, speeds, protocols, and duplex modes. Micromputers communicate (both internally and externally) with ASCII codes, which include control characters and display characters.

## For further reading

If you would like to learn more about communications, refer to the following:

Friend, George E., John L. Fike, H. Charles Baker, and John C. Bellamy, *Understanding Data Communications*, Dallas: Texas Instruments, 1984.

# Part II

# Text Editing

In Part II you will learn how to enter and edit text with **vi** and **ex**, which will allow you use all of your screen as a work area. These programs are more convenient, but also slower, than the line editor **ed**. After an introduction to the features, you will learn how to change and delete text, how to find and replace text, how to move and copy text from one place in a file to another, how to move and copy text from one file to another, how to mark certain lines in a file for easy access, how to modify **vi** operating options, and how to use abbreviations and key definitions. Note that under UNIX, formatting is separate from editing. Formatting text for printing is discussed in Part IV.

# 6

# Introduction to **vi**

## 6.1 Background

### Text and computers

As you can see by looking at the Table of Contents, three of the seven parts of this book relate to text. One of the main uses of computers is to work with text: enter it into a file, process it, format it, or print it on paper. You are working with text every time you write a memo, a letter, a computer program, an article, or a book.

The programs that help you work with text are becoming more and more sophisticated every year, making the use of computers more and more convenient. Twenty years ago, the program that allowed you to enter and modify text (a *text editor*) was separate from the program that let you control the appearance of the text on a printed page (a *text formatter*). In the early 1970s, Wang and others began to introduce a new kind of program that combined these separate functions in a single program (a *word processing* program). Word processing programs played a significant role in promoting the popularity of personal computers.

In the 1980s, we are witnessing the start of a major revolution in the publishing industry as new programs venture beyond basic editing and formatting to laying out pages, merging text and graphics, and designing books. We are also seeing more programs that handle related tasks, such as checking for correct spelling, grammar, and style.

### Text-editing and UNIX

The first text editor for UNIX was a fairly primitive program called **ed** that was line oriented. Then, around 1976, an enhanced version of **ed** called **ex** was developed at the University of California by William Joy and others. A major feature of **ex** is that it can be run in a visual mode called **vi** (vee-eye, the visual interpreter). Most people prefer **vi** over **ed** because of its ease of use, and some even prefer **vi** over word processing programs because of its many useful features. (For those who may prefer to use **ed**, see Appendix B, "Summary of **ed** Commands").

Since **vi** uses the entire screen, like most popular word processing programs, you will find it very convenient. However, you can't just invoke the program and start using it, as you can with **ed**. You first have to make sure that UNIX has detailed information about your terminal. (This corresponds to the *installation* procedure for some word-processing programs).

To determine whether or not your UNIX system has been prepared for your terminal, proceed to the next section and try invoking **vi**. If the correct display appears on your screen as described, you can complete the rest of this chapter right now. However, if the display is distorted or doesn't appear at all, turn to Chapter 33, "Terminals and Printers", for the information you will need. Then return to this chapter and continue.

## 6.2 Typing a letter

Once someone has identified your terminal to the UNIX shell, as described in Appendix L, "Summary of **termcap** and **terminfo**", you can call up **vi** and start typing a letter.

### Starting with a new directory

As you learned earlier, one way to take full advantage of the UNIX file system is to organize your home directory into subdirectories for different kinds of files. Let's start this chapter by creating a new subdirectory in your home directory to contain your **vi** files, then move to this directory for the exercises in this chapter.

1. Create a new subdirectory:

   □ Create a directory called text with the **mkdir** command:

   ```
   $ mkdir text
   $ _
   ```

   □ The directory has been created, but you are still in your home directory.

2. Move from your home directory to text:

   □ Move to subdirectory text with the **cd** command:

   ```
   $ cd text
   $ _
   ```

   □ Now text is your working directory. Any files you create will be stored in this directory.

3. Call up the visual editor **vi**:

- □ After the UNIX shell prompt, type ***vi letter*** and press (RETURN):

```
$ vi letter
```

- □ After a few moments, the screen will clear and the following display will appear:

```
_
~
~
~
~
~
~
~
~
~
~
~
~
"letter" [New file]
```

The number of lines that appear in the *display window* varies from terminal to terminal.

## Typing the First Draft

With **vi** running and a file named, you are ready to start entering the text of your letter:

1. Select *appending*:

   - □ Type **a** (without (RETURN)) to append text in *text entry mode.*
   - □ The **a** will *not* appear on the screen, but once you have typed it, you can begin entering text.

2. Type a few lines of text:

   - □ Type the following, pressing (RETURN) at the end of each line:

```
Dear Mr. Fenton: (RETURN)
(RETURN)
I came to your office straight from (RETURN)
the tennis courts.  There wasn't (RETURN)
enough time to take a shower before (RETURN)
the interview. (RETURN)
(ESC)
```

- □ The text will appear on the screen as you type it, with each new line replacing one of the tildes in the left-hand column.

3. Return to *vi command mode*:

- □ Press the ESC (*Escape*) key to leave *text entry mode.*
- □ The screen display will not change, but you will return to *vi command mode*, from which you can enter more commands.

4. Try the *repeat command*:

- □ Type a period ( . ) (without RETURN) to repeat the insertion.
- □ Now the screen will look like this:

```
Dear Mr. Fenton:

I came to your office straight from
the tennis courts.  There wasn't
enough time to take a shower before
the interview.

Dear Mr. Fenton:

I came to your office straight from
the tennis courts.  There wasn't
enough time to take a shower before
the interview.

_
~
~
```

Once you typed the period ( . ), **vi** repeated the insertion of five lines of text (including two blank lines)—beginning at the cursor's location on the screen. The unused lines below the text still begin with tildes.

5. Try the *undo command*:

- □ Type ***u*** to undo the insertion.
- □ The screen will again look like this again:

```
Dear Mr. Fenton:

I came to your office straight from
the tennis courts.  There wasn't
enough time to take a shower before
the interview.

_
~
~
```

□ Type ***u*** again to restore the insertion.

□ Type ***u*** once more to remove the insertion.

You can use the *undo command* to recover from an error. It will undo the results of the most recent change performed, no matter how extensive. Note that when used twice in succession, the **u** command undoes the action of the previous *undo command.*

6. Write the text to the file:

□ Now type ***:w*** and press RETURN to write this text to **letter**.

□ First you will see `:w` at the bottom of the screen; then you will see this display on the status line:

```
"letter" [New file] 7 lines, 141 characters
```

A colon ( : ) always means a temporary change to *ex command mode*, from which you can use an **ex** command—in this case, the **w** (write) command.

## Inserting the date

Now we'll begin adding things to the letter, starting with the date. To do this, we'll have to move the cursor to the top of the screen and make room for another line of text.

1. Move the cursor to the top of the screen:

□ Hold down the SHIFT key and type capital **H** (not **h**) to move the cursor to the upper left-hand corner of the screen.

□ The display should look like this:

```
Dear Mr. Fenton:

I came to your office straight from
the tennis courts.  There wasn't
enough time to take a shower before
the interview.

~
~
```

2. Open a new line above:

□ Type capital **O** (not **o**) to open a new line above the first line.

□ Now there will be a blank line above the first line, with the cursor resting at the left margin.

3. Type the date:

- □ Type the date as shown on the blank line and press RETURN.
- □ Now the display should look like this:

```
March 17, 1987
_
Dear Mr. Fenton:

I came to your office straight from
the tennis courts.  There wasn't
enough time to take a shower before
the interview.

~
~
```

4. Return to *vi command mode*:

- □ Press the ESC key to leave "line-opening" in *text entry mode.*
- □ You have to do this every time you finish typing new text. Otherwise, **vi** will interpret your next keystroke as another character of text instead of a command.

## Inserting a new paragraph

Having inserted the date and selected *vi command mode* again, you are now ready to insert a new paragraph. This involves moving the cursor down and selecting *insert mode.*

1. Move the cursor down:

- □ Type **j** (unshifted) twice to move the cursor down to the blank line after the salutation.
- □ If your terminal has arrow keys (and your **termcap** entry permits), you can use the down arrow key (↓) instead of **j**.

2. Prepare to insert a new paragraph:

- □ Type **i** to select inserting in *text entry mode.*
- □ You won't see **i** on the screen and the display won't change:

```
March 17, 1987

Dear Mr. Fenton:
_
I came to your office straight from
the tennis courts.  There wasn't
enough time to take a shower before
the interview.

~
```

3. Insert a new paragraph:

   □ Press RETURN to leave a blank line after the salutation, type the following, press RETURN after each line, then ESC at the end:

   ***I'm sorry you fainted during my*** RETURN
   ***interview on Friday.  I was*** RETURN
   ***raising my arms to give the*** RETURN
   ***victory signal.*** RETURN
   ESC

   □ Your display should look like this:

```
March 17, 1987

Dear Mr. Fenton:

I'm sorry you fainted during my
interview on Friday.  I was
raising my arms to give the
victory signal.
_
I came to your office straight from
the tennis courts.  There wasn't
enough time to take a shower before
the interview.

~
```

4. Return to *vi command mode*:

   □ Press the ESC key to leave *text entry mode.*

   □ Always return to *vi command mode* after completing text entry.

5. Repeat the paragraph:

   □ Type a period ( . ) to repeat the paragraph you just typed.

   □ Your display will look like this:

```
March 17, 1987

Dear Mr. Fenton:

I'm sorry you fainted during my
interview on Friday.  I was
raising my arms to give the
victory signal.

I'm sorry you fainted during my
interview on Friday.  I was
```

```
raising my arms to give the
victory signal.

I came to your office straight from
the tennis courts.  There wasn't
enough time to take a shower before
the interview.

~
~
~
```

6. Now delete the second copy of the first paragraph:

   □ Type ***u*** (not **U**) to undo the last command.

   □ The repeated paragraph will now disappear.

## Adding a closing

This letter needs a closing. This involves moving the cursor down and adding more text.

1. Move the cursor to the last line of the display:

   □ Type capital **L** (not **l**) to move the cursor to the blank line below the second paragraph.

   □ The cursor should now be in the lower left-hand corner of the display window.

2. Select appending in *text entry mode* again:

   □ Type ***a*** to append more text.

   □ Nothing will happen, but **vi** is now ready to receive more text.

3. Type the closing:

   □ Type the following, pressing (RETURN) after each line of text:

   (RETURN)
   ***I hope you will give very careful*** (RETURN)
   ***consideration to my qualifications.*** (RETURN)
   (RETURN)
   ***Bob ("Ace") Sanders*** (RETURN)

   □ Your display should look like this:

```
March 17, 1987

Dear Mr. Fenton:
```

```
I'm sorry you fainted during my
interview on Friday.  I was
raising my arms to give the
victory signal.


I came to your office straight from
the tennis courts.  There wasn't
enough time to take a shower before
the interview.

I hope you will give very careful
consideration to my qualifications.

Bob ("Ace") Sanders

_
~
~
~
```

4. Return to *vi command mode*:

   □ Press the ESC key to leave *text entry mode.*

   □ Always return to *vi command mode* after completing entry of text.

5. Write the text to the file:

   □ Type ***:w*** and press RETURN to write this text to **letter**.

   □ You will see `:w`, then this display at the bottom of the screen:

   ```
   "letter" 19 lines, 363 characters
   ```

## 6.3 Making changes to the letter

Now that we have a complete letter, let's make some changes to it. This will help us try various features of **vi**.

### Changing the date

Let's start with the date at the top of the letter. We'll start by moving the cursor, then make a change.

1. Move the cursor up to the first line:

   □ Hold down the SHIFT key and type ***H*** to move the cursor to the top of the display.

- □ The cursor should now be positioned over the `M` in `March.`

2. Change `March` to `September`:

- □ Type ***cw*** to change a word. The display will look like this:

```
Marc$ 17, 1987
```

- □ Immediately type ***September*** and press ESC.
- □ The first line should now look like this:

```
September 17, 1987
```

The **c** (change) command can be used either alone or in conjunction with another character (in this case, **w** for ***word***). The character that follows **c** specifies exactly how much text is to be changed. You'll learn how to use **c** and other commands with characters like **w** in the chapters that follow.

## Changing a name

Next we'll change the name on the salutation line. This involves moving the cursor into position and making a double change.

1. Move the cursor to the start of the name on the salutation line:

- □ Type a slash (**/**) to request a search, followed immediately by ***Mr***. At the bottom of the screen, you will see

```
/Mr_
```

- □ Now press RETURN to begin the search.
- □ The cursor will jump to the `M` in `Mr.` on the third line.

2. Change `Mr. Fenton` to `Mrs. Benson`:

- □ Type ***2cW*** to change two words. The line will look like this:

```
Dear Mr. Fenton$
```

- □ Type ***Mrs. Benson:*** immediately after ***2cW***, then press the ESC key.
- □ The salutation line will now look like this:

```
Dear Mrs. Benson:_
```

## Running a UNIX command

Sometimes you work so hard at your terminal that you lose track of the time. However, **vi** lets you take a look at the clock (using the **date** command) without having to interrupt your editing session. You can also run any other UNIX command from **vi**.

1. Find out what time it is:

   □ Type ***:!date*** and press RETURN to find out the date and time.

   □ You should see a display like this appear:

```
:!date
[No write since last change]
Mon Apr  3 13:52:27 PST 1987
[Hit return to continue]_
```

   □ As the prompt says, press RETURN to return to **vi**.

2. Find out who else is on the system:

   □ Type ***:!who*** and press RETURN to obtain a list of active users.

   □ You should see a display like this appear:

```
:!who
john     tty07   Apr 3 08:43
janice   tty16   Apr 3 09:17
billy    tty03   Apr 3 08:24
[Hit return to continue]_
```

You can type any shell command in this way. The main thing to remember is that you can run any UNIX command from within **vi** by preceding the command line by the two characters **:!**. The colon gives you **ex**; the exclamation mark gives you the shell. (For an extensive session with the shell, you can also use **:sh**, execute your commands, then type CTRL-D to return to **vi**.)

## Getting information

Before making any more changes, let's ask **vi** for some information.

1. Move to a specific line:

   □ Type ***9G*** to have the cursor "go to line 9."

   □ Now hold down the CTRL key and press **G** (unshifted )to ask **vi** for status information.

□ You will see a display like this at the bottom of the screen:

```
"letter" [Modified] line 9 of 19 --47%--
```

This line tells you the name of your file (**letter**), the line number where the cursor is resting (`9`), the total number of lines (`19`), how far the cursor is from the beginning of the file (`47%` of the file), and whether you have made changes to the file (`[Modified]`) that haven't been written.

2. Move to the first and last lines:

□ Type ***L*** to move the cursor to the last line on the screen, and type CTRL-G again. At the bottom of the screen you will see

```
"letter" [Modified] line 19 of 19 --100%--
```

□ Now type ***H*** to move back to the first line and type CTRL-G a third time. At the bottom of the screen you will see

```
"letter" [Modified] line 1 of 19 --5%--
```

Each time you type CTRL-G, **vi** gives you status information based on the cursor's location in the text.

## Deleting words

Now we'll delete a couple of words just to show you how different **vi** commands have many things in common.

1. Move the cursor to `very` in the twelfth line:

□ Type ***/very*** and press RETURN.

□ Now the cursor is resting on the `v` in `very careful`.

2. Delete the words `very careful`:

□ Type ***d2W*** to delete these two words.

□ the line should look like this:

```
I hope you will give_
```

3. Undo the deletion:

□ Let's restore the original wording by typing ***u***.

□ Now the wording is back the way it was before:

```
I hope you will give very careful
consideration to my qualifications.
```

### Capitalizing words

Before concluding this brief sampling of **vi** commands, let's try one more thing: capitalizing a word. Any word will do; let's pick `interview` in line 13. Since the cursor is currently resting on line 15, this involves a backwards search (unless you have selected the "wrapscan" feature—see Appendix D, "Summary of **vi** Options").

1. Move the cursor into position:

   □ Type ***?interview*** and press (RETURN) to move the cursor up to line 13.

   □ The cursor should now be resting on the first `i` of `interview`.

2. Capitalize the word:

   □ Type the tilde ( ) to make the change.

   □ Line 13 should now look like this:

   ```
   the Interview.
   ```

   □ Since it doesn't make any sense to have this word capitalized, restore it by typing the undo command **u**.

## 6.4 Ending the session

Now it's time to write the updated text to disk file **letter** and return to the UNIX shell.

1. Write the text to file **letter**:

   □ Type ***:w*** and press (RETURN) to write to the file.

   □ You will see a display like this at the bottom of the screen:

   ```
   "letter" [Modified] 19 lines, 363 characters
   ```

2. End this session with **vi**:

   □ Type ***:q*** and press (RETURN) to end the session.

   □ The following display will appear, indicating that you have left **vi** and returned to the UNIX shell:

   ```
   :q
   $ _
   ```

Note that **vi** also allows you to combine the write and quit commands into a single command, like this:

```
                    :wq      or   :x       or   ZZ
                    $ _           $ _           $ _
```

## 6.5 Summary

In this chapter you learned a number of common **vi** procedures, such as beginning a session, entering text, inserting new text, making changes, and ending a session. Before you can use **vi**, you have to make sure that someone has placed an entry for your terminal in **/etc/termcap** (or **/usr/lib/terminfo**) and that you have assigned one of the names for your terminal to the shell variable **TERM** and exported **TERM** to the environment. If necessary, see Chapter 33, "Terminals and Printers," for details.

Start **vi** by typing **vi**, a space, and the name of a file. For a new file, the screen will be cleared, the cursor will move to the upper left-hand corner of the screen, and *vi command mode* will be in effect. For an existing file, its contents will be displayed on the screen. To enter text, type ***a*** (append), ***i*** (insert), or one of several other letters without pressing (RETURN), enter the text, then press (ESC) to return to *vi command mode*. Two of the most useful commands in **vi** are the *repeat command* (.), which repeats the most recent command, and the *undo command* (**u**), which undoes the most recent change performed.

To open a new blank line for inserting new text, use either the **o** command (to open *below*) or the **O** command (to open *above* the current line). Enter the text, then press (ESC) to return to *vi command mode*. To insert new text, use either **i** (to insert in front of the cursor) or **I** (to begin inserting in the first column of the current line). Enter the text, then press (ESC) to return to *vi command mode*.

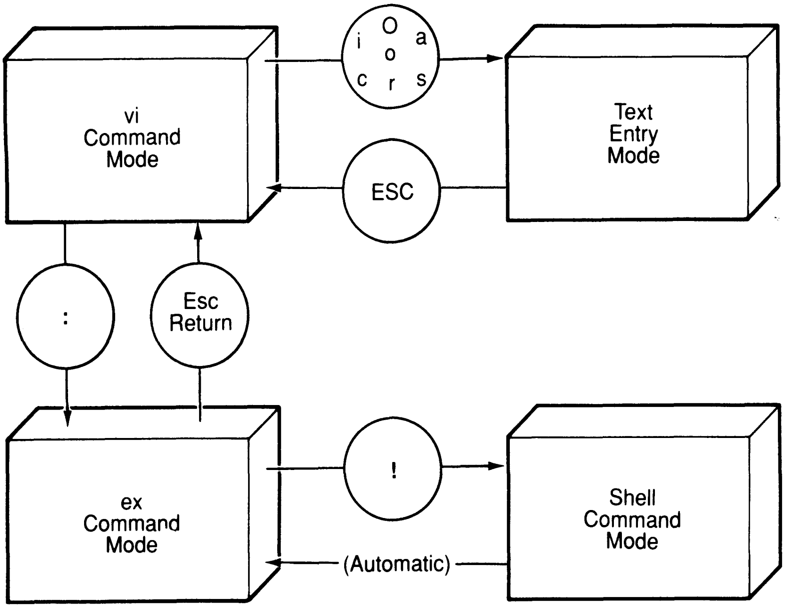
To change a word, move the cursor to the first letter of the word, then type ***cw*** or ***cW*** (change word), followed by the new word (without a space) and (ESC). To change two words, use ***2cw*** or ***2cW***. To search for a string, type either ***/*** (to search forward) or ***?*** (to search backward), followed immediately by the desired string, and press (RETURN).

To execute a UNIX command without having to end your session with **vi**, type ***:!***, followed immediately by the name of the command, and press (RETURN). (Use ***:sh*** to spawn a subshell.) To determine your relative position in the text, press (CTRL-G) to see a display.

To delete a word, move the cursor to the first letter of the word, then type ***dw*** or ***dW*** (delete word). To delete two words, use ***d2w*** or ***d2W***; to delete three, use ***d3w*** or ***d3W***. To change a lower case letter to upper case or an upper case letter to lower case, position the cursor over the letter in *vi command mode* and type the tilde ( ). Non-alphabetic characters are not affected.

To end a session with **vi**, type ***:w*** to write the text to the file, then type ***:q*** to end the session (quit) and return to the UNIX shell. You can also combine these two in a single command.

FIGURE 6.1. The modes at a glance.



# 7

# Making Some Changes

Now that you've had a chance to sample some of the most common features of **vi**, let's examine some of those features in more detail. In this chapter we'll look at ways to insert, change, and delete text. Let's start by beginning a new session and getting acquainted with the various ways of moving the cursor around the display window.

## 7.1 Beginning a new session

Things may have changed a little since you completed the previous chapter on **vi**. If so, here are a few words about returning to **vi** with the same file you were working with in that chapter:

1. If necessary, move from your home directory to **text**:

   □ Use the **cd** (change directory) command like this:

   ```
   $ cd text
   $ _
   ```

   □ Use the **pwd** (print working directory) command to verify:

   ```
   $ pwd
   /usr/robin/text
   $ _
   ```

2. Call up **vi** with the same file (**letter**):

   □ At the shell prompt, type ***vi letter*** and press (RETURN):

   ```
   $ vi letter
   ```

   □ After a few moments, the screen will clear, and the text of your letter will appear, with the cursor on the first line.

   ```
   September 17, 1987

   Dear Mrs. Benson:
   ```

```
I'm sorry you fainted during my
interview on Friday.  I was
raising my arms to give the
victory signal.

I came to your office straight from
the tennis courts.  There wasn't
enough time to take a shower before
the interview.

I hope you will give very careful
consideration to my qualifications.

Bob ("Ace") Sanders

~
~
~
"letter" 19 lines, 363 characters
```

## 7.2 Moving the cursor

To use **vi** effectively, you must first learn how to move the cursor from one location to another. This will not only enable you to move the cursor to where you need it; it will also help you see how **vi** works. Many functions in **vi** begin working at the position of the cursor. In this section, you will learn how to use certain characters to position the cursor on the screen. Later in this chapter you will learn to use many of these same characters with **vi**'s operators to perform a variety of functions.

### Moving a character at a time

Your keyboard may have a separate cursor pad with arrow keys like this: If so, then you may want to use these keys to move the cursor one space at a time. If not, you will have to use the keys ***h***, ***j***, ***k***, and ***l***. Some people prefer to use these keys, even when arrow keys are available, like this:

Press the *character down* key (↓ or j) five times to move the cursor to the fifth line (`I'm sorry...`). Now try all four keys to become familiar with them. When you are finished practicing, leave the cursor at the beginning of the fifth line.

Yet another set of keys, which will work for any version of **vi**, is the following (don't worry—the space bar won't erase your text):

FIGURE 7.1. The cursor arrow keys.

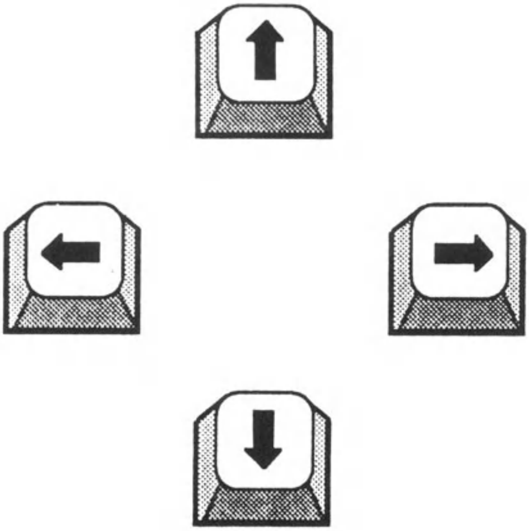


FIGURE 7.2. The new cursor motion keys.

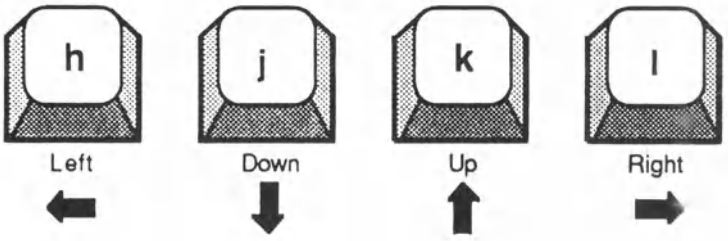
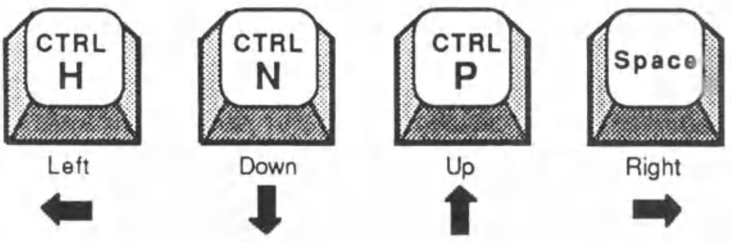
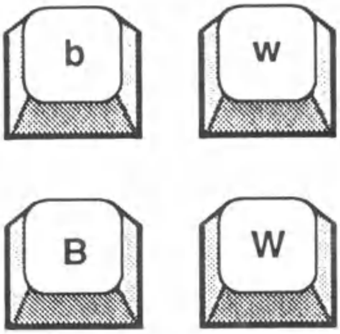


FIGURE 7.3. The old cursor motion keys.



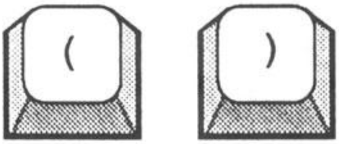
## Moving a word at a time



You also have keys for moving the cursor either forward (**w**) or back (**b**) one entire word at a time. Press **w** about ten times to move forward; press **b** a few times to move back again. As you can see, motion continues from the end of one line to the beginning of the next, and vice versa. When capitalized (shifted), these two keys perform nearly the same functions. However, **W** and **B** disregard punctuation. If you use either **w** or **b**, the cursor will stop on punctuation marks. Two related commands, **e** and **E**, move the cursor to the *last* character of a word rather than the *first.*

Note that you can type a *multiplier* in front of the letter to multiply the number of jumps, as in `2w`, `3b`, and so on. Try commands like these also.

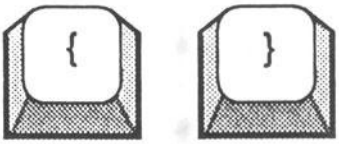
## Moving a sentence at a time



The left and right parenthesis keys (shifted 9 and 0) provide cursor motion to either the beginning (**(**) or the end (**)**) of the current sentence. (Note that **vi** expects a period, followed by two spaces—or a blank line—to mark the end of a sentence.) Try these two keys a few times.

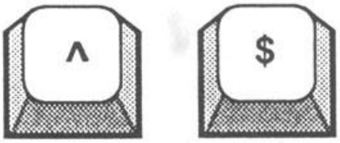
Once again, you can use *multipliers*: **3(**, **4)**, and so on.

## Moving a paragraph at a time



The left and right brace keys provide cursor motion to either the beginning ({) or the end (}) of the current paragraph. (Ordinarily, **vi** regards blank lines as separators of paragraphs. However, if you are writing programs in C, you can change the separators to braces, which delimit blocks of code. For details, see Chapter 12, "Customizing **vi**.") Try these two keys a few times. Again, you can also use *multipliers*: **3**{, **5**}, and so on.

## Moving to either end of a line

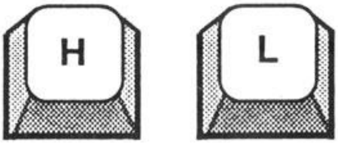


You have a pair of keys for moving the cursor either to the beginning (**^**) or the end (**$**) of the current line .(the same two symbols used by **ed**). The character (**0**) is similar to **^**. It also moves the cursor to the beginning of the line. However, **0** always moves the cursor all the way to the left-hand margin, whereas **^** moves the cursor only to the first *visible* character. Try a few jumps with these keys.

FIGURE 7.4. Basic cursor motion.

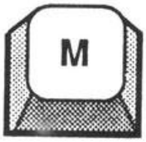
```
{
   Xxx xxxx xxx xx Uxxx xxx xxxxxx xx xxxxx
xxxxxx xxx.  (xxxxx xxx x xxxxxxx xxxxx
^xxxxxxx xxx bx xxxxx wxx xxxx.  )Xxxxx xxxx$
xx xxxxx xxxxxxxx. X xxx xxxxx xxxx xxx.
}
   Yyy yyyy yyy yyyyy yyyy yyyy. Yyyyy yyy
yyyyyyy yyy yyyyyy yyy yyyyy.
```

## Moving to the Top and Bottom of the Screen



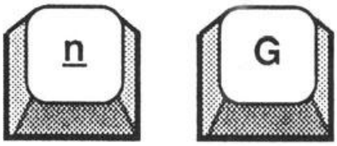
The **H** and **L** keys allow you to move the cursor either to the top (**H**) or the bottom (**L**) of the screen. You can remember the letters as home and lower left. Try your hand with these keys a few times.

## Moving to the Middle of the Screen



The **M** key allows you to move the cursor to the middle (**M**) of the screen. **M,** of course, stands for **middle**. Try this one in conjunction with the **H** and **L** commands.

## Moving to a Given Line



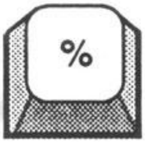
The letter **G** (Go to) provides cursor motion to any line in the editing buffer, regardless whether the line is currently displayed on the screen. By this convention, ***2G*** would take you to the second line; ***5G*** would take you to the fifth line; and so on. One limitation of this key is that you usually have to know the number of the desired line. (If you would like to have **vi** display line numbers on the screen, there is a way to do this. For details, see Chapter 12, "Customizing **vi**.")

Note that ***G*** alone designates the last line of the file and that CTRL-G answers the question, "What is the number of the current line?" Try all these different variations.

### Moving to the matching bracket

If the cursor is currently resting on any of the general bracketing symbols, namely

| | |
|---|---|
| ( ) | parentheses |
| [ ] | square brackets |
| { } | braces |

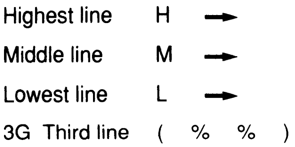


you can use the percent sign (**%**) to move the cursor to the matching symbol. To see how this command works, move the cursor to the last line of text (***L***, then cursor up), and experiment with **%** on the parenthetical expression "Ace").

### A note on cursor motion keys

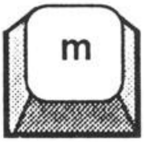
You must be in *vi command mode* to use any of the cursor motion keys just described. (Press (ESC) to return to *vi command mode.* If you hear a beep, your terminal is already in this mode.) None of the cursor motion keys will work after you have entered one of the text insertion modes. The five cursor motion commands just discussed are illustrated in Figure 7.5.

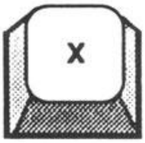
FIGURE 7.5. More cursor motion.



## 7.3 Using markers

### Marking a place





You can set a place marker anywhere in the work area with the **m** command. Just move the cursor to the desired location and type ***m***, followed by any lower case letter of the alphabet. The letter then becomes the name of the place marker.

### Moving to a marker

Once a place marker has been set in the work area with the **m** command, you can move the cursor to the marker with either of the two types of single quotation mark. Type the back quotation mark (or grave accent) `, followed by the letter name of the mark, to move the cursor to the exact location of the mark. Type an ordinary quotation mark ′, followed by the letter, to move the cursor to the beginning of the line that contains the mark.

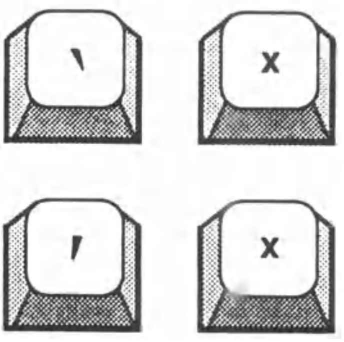
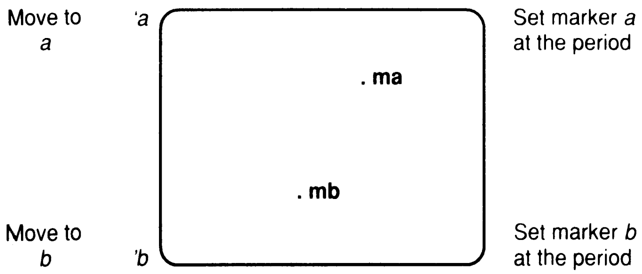


FIGURE 7.6. Using markers.



## 7.4 Controlling the screen display

### Scrolling

You have a pair of keys that produce either a scroll up ((CTRL-U)) or a scroll down ((CTRL-D)). If the cursor is in the upper half of the screen, a scroll up may push the cursor down; if the cursor is in the lower half of the screen, a scroll down may push the cursor up.

To practice scrolling, move the cursor to any line in the letter. Now you can try (CTRL-U) and and (CTRL-D).

Scrolling is illustrated in Figure 7.7.

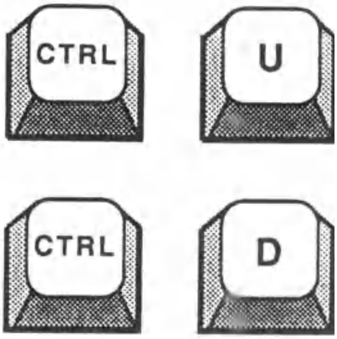
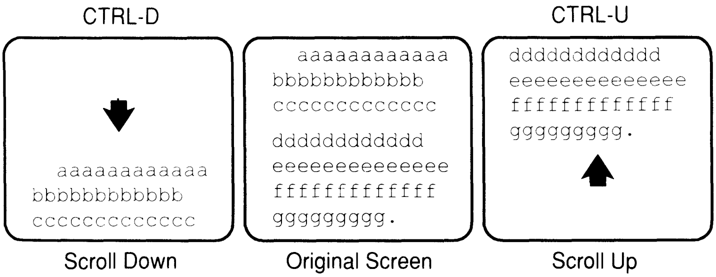


FIGURE 7.7. Scrolling up and down.



## Paging

**vi** also provides one key for paging back ((CTRL-B)) and one for paging forward ((CTRL-F)). ("Paging" means advancing the text one screenful at a time.) Later on, when you are working with long documents, you will find these features very handy.

After practicing with these keys, you can re-move the extra blank lines by abandoning the text (***:q!***) then starting again (***vi letter***). Paging is illustrated in Figure 7.8.

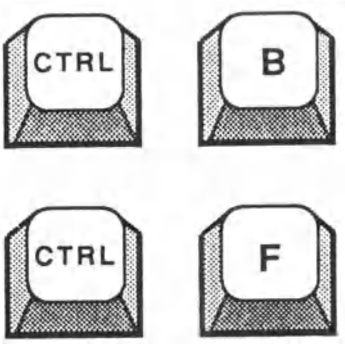
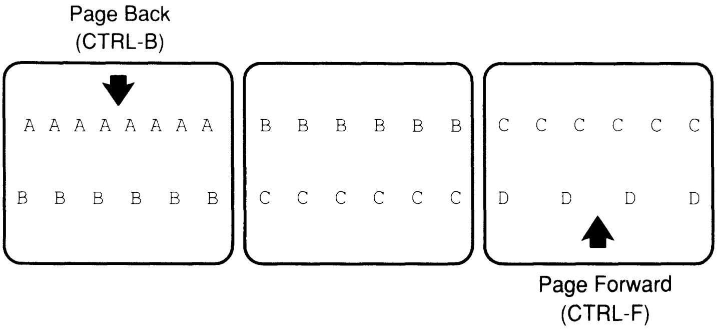
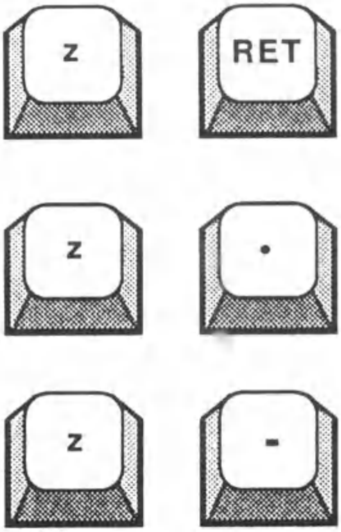


FIGURE 7.8. Paging up and down.

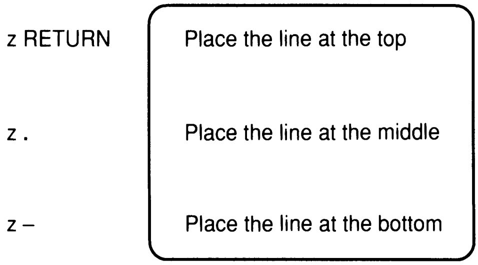


### Repositioning the current line



You have a command that you can use to re-position the current line in the display window—**z**, the "zero screen" command. To place the current line at the *top* of the screen, type ***z*** and press (RETURN); to place the current line in the *middle* of the screen, type ***z.***; to place the current line at the *bottom* of the screen, type ***z-***.
This command also has several variations: If you type a line number *in front of* ***z***, that line will be placed at the top of the screen; if you type a number *after* ***z***, your display window will be reduced (or increased) to that many lines. For example, type ***15z*** to place line 15 at the top of the screen. Type ***z12*** to reduce your display window to 12 lines. Repositioning the current line is illustrated in Figure 7.9.

FIGURE 7.9. Repositioning the current line.

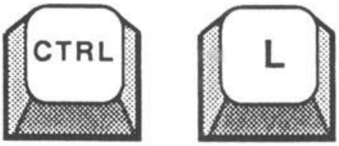


z RETURN Place the line at the top

z . Place the line at the middle

z – Place the line at the bottom

### Clearing system messages



You have a command for clearing system messages from your screen: (CTRL-L). This will clear messages from, but leave the text in your work area unchanged.

## 7.5 Adding new text

As you saw in Chapter 6, you have three choices when you want to add new text in **vi** without altering existing text:

- You can *append* text after existing text.

- You can *insert* text in front of existing text.
- You can *open* up space for a new line of text.

In this chapter we'll explore each of these in greater detail.

## APPENDING TEXT

**a**
**A**

To append text after existing text on the screen, you can use one of two commands: One that lets you add the new text immediately after the cursor (**a**) or one that lets you add the next text at the end of the current line (**A**). Let's take a look at each of them, using the first line of the first paragraph (line 5) of **letter**.

1. Move the cursor into position:

   □ In *vi command mode*, type ***5G*** to place the cursor on line 5.

   □ Type ***4E*** to move the cursor across the line to `fainted`.

   □ The line should now look like this:

   ```
   I'm sorry you fainted during my
   ```

2. Append `away` (with a preceding space) after `fainted`:

   □ In *vi command mode*, type ***a*** to append text after the cursor.

   □ Type ***away*** and press ESC to return to *vi command mode.*

3. Append `first` (with a preceding space) to the end of the line:

   □ In *vi command mode*, type ***A*** to append text at the end of the line.

   □ Type ***first*** and press ESC to return to *vi command mode.*

   □ Now the line should look like this:

   ```
   I'm sorry you fainted away during my first
   ```

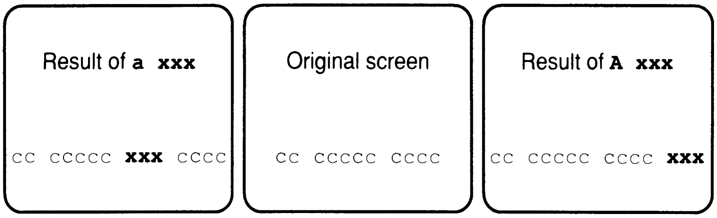
As you can see from these examples, **vi** commands have one meaning in lower case (**a**) and another in upper case (**A**). See Figure 7.10.

## INSERTING TEXT

**i**
**I**

To insert text in front of existing text on the screen, you can use one of two commands: One that lets you add the new text immediately before the cursor (**i**) or one that lets you add the new text at the beginning of the current line (**I**). (If the text is indented, you will want to use **0I**.) Let's take a look at **i** and **I** using line 7 of **letter**.

FIGURE 7.10. Appending text.



1. Move the cursor into position:

   □ In *vi command mode*, type ***/the*** to move the cursor to `the` on line 7.

   □ The line should now look like this:

   ```
   raising my arms to give the
   ```

2. Insert `you` (with a trailing space) in front of `the`:

   □ In *vi command mode*, type ***i*** to insert text before the cursor.

   □ Type ***you*** and press (ESC) to return to *vi command mode.*

3. Insert `just` to the beginning of the line:

   □ In *vi command mode*, type ***I*** to insert text at the beginning of the line.

   □ Type ***just*** (with a trailing space) and press (ESC) to return to *vi command mode.*

4. Capitalize the `v` in `victory`:

   □ Type ***8G*** to move the cursor to the `v` in `victory`.

   □ Type  to change the `v` to a `V`, so that the line looks like this:

   ```
   Victory signal.
   ```

   □ Type ***u*** to undo the capitalization.

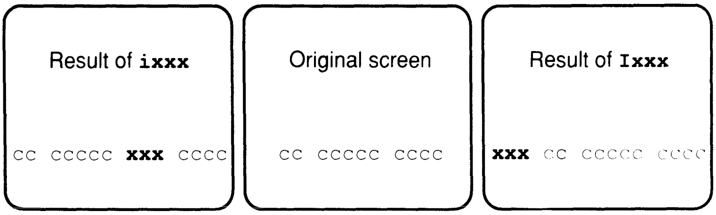
Once again, **i** and **I** are two different commands (see Figure 7.11).

## Opening a new line

**O**
**o**

You can open space for a new line either *above* the current line (**O**) or *below* the current line (**o**). Once again, we'll try one example of each, one near the beginning of the letter and one near the end.

FIGURE 7.11. Inserting text.



1. Move the cursor into position:

   □ Type ***G*** then ***k*** to move the cursor to the last line of text.

   □ The cursor will be at the beginning of the line (but it doesn't really matter where on the line the cursor is for these commands):

   ```
   Bob ("Ace") Sanders
   ```

2. Add `Sincerely,` and three blank lines above.

   □ Type ***O*** to open a new line above the current line.

   □ Type ***Sincerely,***, press (RETURN) three times, and press (ESC) to return to *vi command mode*, so that the closing lines look like this:

   ```
   I hope you will give very careful
   consideration to my qualifications.

   Sincerely,



   Bob ("Ace") Sanders
   ```

3. Add a name and address on new lines below the date:

   □ Type ***1G*** to move the cursor to the first line (the date).

   □ Type ***o*** to open a new line below the current line.

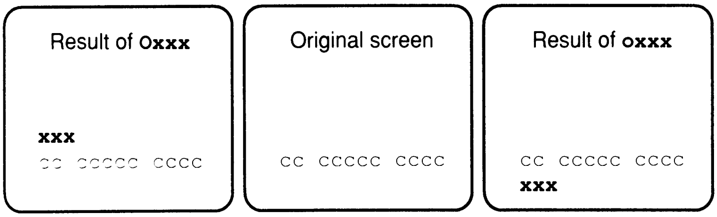
   □ Type the three lines shown (each followed by (RETURN)) and press (ESC) to return to *vi command mode*, so that the lines look like this:

```
September 17, 1987

Agatha R. Benson
Vice President
Fifth National Bank
_
Dear Mrs. Benson:
```

This session will be continued in Chapter 8, "Changing and Deleting Text" (see Figure 7.12).

FIGURE 7.12. Opening a new line.



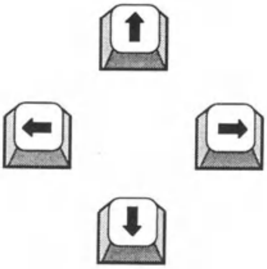
## 7.6 Summary

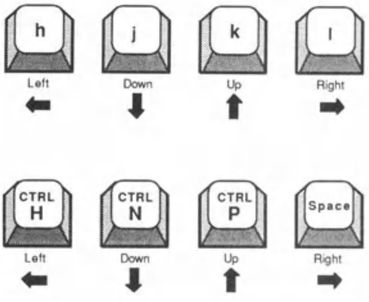
In this chapter you learned how to move the cursor and add new text.

### Moving the cursor

**`Moving a space at a time`**—You can move the cursor one space at a time with a choice of 1) arrow keys on a cursor pad, 2) typewriter keys, or 3) control keys.

```
                         ←         ↑
         ↑         (2)   h    j    k     l
(3)  ←       →     (3)   ^H   ^N   ^P^   space
         ↓                    ↓          →
```





| **Moving** | Back | Ahead | To end |
|---|---|---|---|
| Word(s) | b | w | e |
| | 5b | 5w | 5e |
| | B | W | E |
| | 5B | 5W | 5E |
| Line | ^ | | $ |
| | First | 1G | |
| | Line *n* | *n*G | |
| | Last | G | |
| Sentence(s) | ( | ) | |
| | 5( | 5) | |
| Paragraph(s) | { | } | |
| | 5{ | 5} | |
| Screen | Top: | H | |
| | Middle: | M | |
| | Bottom: | L | |
| Matching Bracket | % | % | |

## Controlling the screen display

| **Scrolling** | | | **Paging** | | |
|---|---|---|---|---|---|
| Up: | CTRL-U | (^U) | Back: | CTRL-B | (^B) |
| Down: | CTRL-D | (^D) | Forward: | CTRL-F | (^F) |

**Placing current line**

| | |
|---|---|
| Top: | z RETURN |
| Middle: | z. |
| Bottom: | z- |

**Clearing messages**

CTRL-L

## Adding new text

| | | | |
|---|---|---|---|
| **Appending text** | | | |
| After cursor: | a | At end of line: | A |
| **Inserting text** | | | |
| Before cursor: | i | At beginning of line: | I |
| **Opening a new line** | | | |
| Below current line: | o | Above current line: | O |

# 8

# Changing and Deleting Text

In Chapter 7 you learned how to move the cursor and adjust the display in **vi**. You then learned how to add new text. In this chapter you will learn how to change and delete text. These two functions are very similar in **vi**.

## 8.1 Changing text

So far, we have only added new text to existing text. We can also make changes to existing text, using the cursor motion characters in conjunction with the change command (**c**) to produce a number of variations. Later in this section we'll talk about shifting text in either direction.

### Changing a word

**cw**
**cW**

The change command (**c**) can be used by itself to change a single letter. But we'll save another command for that, and go on to the next possibility: changing a word. To show how this works, let's move to the first line of the second paragraph.

1. Move the cursor into position:

   □ In **vi** command mode, type ***/I'm*** and press (RETURN).

   □ The cursor should now be be over the `I` in line 9:

   ```
   I'm sorry you fainted away during my first
   ```

2. Change `I'm` to `I am`:

   □ In **vi** command mode, type ***cW*** to change a word.

   □ A dollar sign will appear in place of the `m`:

   ```
   I'$ sorry you fainted away during my first
   ```

   □ Type ***I am*** and press (ESC) to return to **vi** command mode.

   □ The line should now look like this:

```
I am sorry you fainted away during my first
```

We used **cW** in this exercise instead of **cw** because `I'm` has an apostrophe in the middle of it. With **cw**, this would have happened:

```
I am'm sorry you fainted away during my first
```

## Cursor motion and **vi** commands

The previous example looks very simple, but let's review what happened anyway. The example illustrates some things that you will be seeing again and again in **vi**. You may recall from Chapter 7 that **W** by itself makes the cursor jump to the next word (ignoring punctuation), like this:

```
I'm sorry you fainted away during my first
                 W
I'm sorry you fainted away during my first
```

When you combine **W** with **c**, you are telling **vi** to change (**c**) the text (not counting the blank space) that **W** would make the cursor jump over:

```
Change text from here
↓
I'm sorry you fainted away during my first
                 cW
I'm sorry you fainted away during my first
    ↑
    to here.
```

In the examples that follow, we'll be using different characters, but the rule is always the same: the change starts where the cursor is now and stops just before the place where the cursor would land after a jump.

## Changing more than one word

**c*n*w**
**c*n*W**

Changing more than one word is similar to changing one word. The only difference is that you place a number in front of **w** to tell **vi** how many words to change. Let's use the same line of text to illustrate this.

1. Move the cursor into position:

   □ In **vi** command mode, type **3w** to advance the cursor to the `f` in `fainted`

   □ The line should now look like this:

   ```
   I am sorry you fainted away during my first
   ```

2. Change `fainted away` to `passed out`:

   □ Type **c2w** to change two words.

   □ You will see a dollar sign appear in front of `during`:

   ```
   I am sorry you fainted awa$ during my first
   ```

   □ Type ***passed out*** and press ESC to return to **vi** command mode.

   □ The line should now look like this:

   ```
   I am sorry you passed out during my first
   ```

As you can see, the rule for cursor motion applied here: the change started at the cursor's original position and stopped before the `d` in `during` (which is where **2w** by itself would have moved the cursor).

## Changing to the Beginning of the Line **c^**

To change text from somewhere in the middle of a line to the beginning of the line, use **c** followed by ^, the command to move the cursor to the beginning of a line. To illustrate this, we'll use the third line of the first paragraph.

1. Move the cursor into position:

   □ In **vi** command mode, type ***/straight*** and press RETURN.

   □ The cursor will now be on the `s` in `straight`:

   ```
   I came to your office straight from
   ```

2. Change the first five words to `I had to rush` :

   □ Type **c^** to request a change from the beginning of the line to the cursor's current position:

   ```
   I came to your office$straight from
   ```

   □ Immediately type ***I had to rush*** (with a space following `rush`) immediately after **c^** and press ESC to return to **vi** command mode.

   □ The line should now look like this:

   ```
   I had to rush_straight from
   ```

## CHANGING TO THE END OF THE LINE **c$**

To change text from anywhere in a line to the end of the line, use **c** followed by **$**, the command to move the cursor to the end of a line. To illustrate this, we'll use the third line of the same paragraph.

1. Move the cursor into position:

   □ In **vi** command mode, type ***/take*** and press (RETURN).

   □ Now the cursor will be on the `t` in `take`:

   ```
   enough time to take a shower before
   ```

2. Change `take a shower before` to several words:

   □ Type **c$** to request a change from the cursor's position to the end of the line:

   ```
   enough time to take a shower before$
   ```

   □ Type ***get ready for*** immediately after **c$** and press (ESC) to return to **vi** command mode.

   □ The line should now look like this:

   ```
   enough time to get ready for
   ```

## CHANGING AN ENTIRE LINE **cc**

One of the rules of **vi** for most commands is this: To make a command apply to an entire line, type the command twice in succession. Using this rule, the command to change an entire line is **cc**. To illustrate this, we'll use the last line.

1. Move the cursor into position:

   □ In **vi** command mode, type ***/Bob*** and press (RETURN).

   □ The cursor will now be on the capital `B` in `Bob`:

   ```
   Bob ("Ace") Sanders
   ```

2. Change `Bob ("Ace") Sanders` to `Robert G. Sanders III`:

   □ Type **cc** to request a change to the entire line. (The line will vanish.)

   □ Type ***Robert G. Sanders III*** immediately after **cc** and press (ESC) to return to **vi** command mode.

□ The line should now look like this:

```
Robert G. Sanders III
```

After all these changes, the entire letter should look like this:

```
September 17, 1987

Agatha R. Benson
Vice President
Fifth National Bank

Dear Mrs. Benson:

I am sorry you passed out during my first
interview on Friday.  I was
just raising my arms to give you the
victory signal.

I had to rush straight from
the tennis courts.  There wasn't
enough time to get ready for
the interview.

I hope you will give very careful
consideration to my qualifications.

Sincerely,



Robert G. Sanders III
~
~
~
```

## Changing to the Beginning of the Sentence **c(**

Changing text from anywhere in a sentence to the beginning is just like changing to the beginning of a line, described above. The only difference is that you use **c(** instead of **c0** or **c^**. To illustrate this, let's use the first line of the first paragraph (`I am sorry...`).

1. Move the cursor into position:

□ In **vi** command mode, type ***?during*** and press (RETURN).

□ Now the cursor will be on the `d` in `during`:

```
I am sorry you passed out during my first
```

2. Change the first half of the sentence:

   □ Type **c(** to request a change from the beginning of the line to the cursor's position:

   ```
   I am sorry you passed out during my first
   ```

   □ Immediately type ***I'm sorry you fainted*** (with a space following) and press ESC to return to **vi** command mode.

   □ The line should look now like this:

   ```
   I'm sorry you fainted_during my first
   ```

3. Undo the change:

   □ Type **u** (undo) to restore the sentence.

   □ The sentence should now look like this again:

   ```
   I am sorry you passed out$during my first
   ```

## Changing to the end of the sentence **c)**

Changing text from anywhere in a sentence to the end is just like changing to the end of a line, described above. The only difference is that you use **c)** instead of **c$**. To illustrate this, let's use the same sentence.

1. Change the second half of the sentence:

   □ Advance the cursor to the `d` in `during` with **6W**.

   □ With the cursor resting on the `d` in `during`, type **c)** to request a change from the cursor's position to the end of the sentence: I am sorry you passed out I was

   □ Immediately type ***at the end of*** RETURN ***last Friday's interview.*** and press ESC to return to **vi** command mode.

   □ The sentence should now look like this:

   ```
   I am sorry you passed out at the end of
   last Friday's interview.  I was
   ```

2. Undo the change:

   □ Type **u** to restore the sentence.

   □ The sentence should now look like this again:

   ```
   I am sorry you passed out during my first
   interview on Friday.  I was
   ```

## CHANGING TO THE BEGINNING OF THE PARAGRAPH **c{**

Changing text from anywhere in a paragraph to the beginning is just like changing to the beginning of a line. The only difference is that you use **c{** instead of **c(**. To illustrate this, let's use the same paragraph.

1. Move the cursor into position:

   □ In **vi** command mode, type ***/I was*** and press (RETURN).

   □ The cursor will now be on the `I` in `I was`:

   ```
   I am sorry you passed out during my first
   interview on Friday.  I was
   ```

2. Change the first part of the paragraph:

   □ Type **c{** to request a change from the beginning of the paragraph to the cursor's position:

   ```
   I was
   ```

   □ Press (RETURN) and type ***I'm sorry you fainted during last*** (RETURN) ***Friday's interview. You see,*** (with a space following) and press (ESC) to return to **vi** command mode.

   □ Now the paragraph should look like this:

   ```
   I'm sorry you fainted during last
   Friday's interview.  You see,_I was
   ```

## CHANGING TO THE END OF THE PARAGRAPH **c}**

Changing text from anywhere in a paragraph to the end is just like changing to the end of a line, described above. The only difference is that you use **c}** instead of **c$**. To illustrate this, let's use the same line.

1. Change the last part of the paragraph:

   □ Advance the cursor to the `I` in `I was`,

   □ Type **c}** to request a change from the cursor's position to the end of the paragraph (the text will vanish from the screen):

   ```
   Friday's interview.  You see, _
   ```

   □ Immediately complete the paragraph as shown, then press (ESC) to return to **vi** command mode.

```
Friday's interview.  You see, my arms RETURN
just went up into a spontaneous RETURN
victory sign. ESC
```

2. Undo the change:

□ Type **u** to restore the paragraph.

□ It should now look like this again:

```
Friday's interview.  You see, I was
just raising my arms to give you the
victory signal.
```

## 8.2 Deleting text

To *delete* text, use the cursor motion characters in conjunction with the delete command (**d**) to produce variations like those for *changing* text. Since we'll be making a lot of deletions, we'll have to undo most of them with the **u** (undo) command; otherwise, we'll run out of text to delete.

### Deleting a character **x**

The command for deleting a single character (**x**) doesn't follow the pattern for deleting other segments of text. To delete a character, move the cursor to the character and type **x** (not **d**).

### Deleting more than one character **nx**

To delete more than one character (say two characters), place a multiplier (2) in front of the **x** command. Let's try this now.

1. Move the cursor into position:

□ In **vi** command mode, type ***?al.*** and press RETURN.

□ Now the cursor will be on the a in `signal`:

```
victory signal.
```

2. Change `signal` to `sign` by deleting two characters:

□ Type **2x** to make the deletion, so that the line looks like this:

```
victory sign.
```

## DELETING A WORD (OR WORDS)

**dw**
**dW**

To delete a word, move the cursor to the first letter of the word and type **dw**. This works just like **cw**, except that you don't have to enter any replacement text with **dw**. Let's try deleting several words in the text.

1. Move the cursor into position:

   □ Type ***/give very*** and press (RETURN).

   □ The cursor will drop down to the g in give in the third paragraph:

   ```
   I hope you will give very careful
   ```

2. Delete one word:

   □ Type **dw** to delete give:

   □ The line should now look like this:

   ```
   I hope you will very careful
   ```

   □ Restore the word with **u**:

   ```
   I hope you will give very careful
   ```

3. Delete several words:

   □ Type **3dw** (or **d3w**) to delete give very careful:

   □ The line should now look like this:

   ```
   I hope you will _
   ```

   □ Restore the words with **u**:

   ```
   I hope you will give very careful
   ```

If the word (or words) you are deleting are next to punctuation or contain punctuation (isn't, won't, $5.67), use **W** instead of **w**). Here's an example:

4. Move the cursor up to the first paragraph:

   □ Type ***?I'm*** and press (RETURN).

   □ The cursor will move up to the I in I'm:

```
I'm sorry you fainted during last
```

5. Delete two words along with punctuation:

   □ Type **2dW** (or **d2W**), so that the line looks like this:

   ```
   you fainted during last
   ```

   □ Undo the deletion with **u**:

   ```
   I'm sorry you fainted during last
   ```

To delete *part* of a word, move the cursor to the first letter to be deleted and follow the same procedure. For example, to delete `ment` from `establishment,` move the cursor to the `m` and type **dw**.

## Deleting to the Beginning of a Line **d^**

To delete text from anywhere in a line to the beginning of the line, use **d^**, which is analogous to **c^**. Move the cursor to the first character to be retained and type **d^**. All text on the line to the left of the cursor will be deleted. Type **4W** to advance the cursor to the `d` in `during` and type **d^** on the current line. All text to the left of the cursor will vanish:

```
during last
```

Undo this deletion with **u**:

```
I'm sorry you fainted during last
```

## Deleting to the End of a Line **d$**

To delete text from anywhere in a line to the end of the line, use **d$**, which is analogous to **c$**. Move the cursor to the first character to be deleted and type **d$**. All text on the line to the right of the cursor will be deleted. Let's try this.

1. Move the cursor down to the third paragraph:

   □ Type ***`/straight`*** and press RETURN.

   □ The cursor will land on the `s` in `straight`:

   ```
   I had to rush straight from
   ```

2. Delete the rest of the line to the right:

□ Type **d$** to make the deletion, so that the line looks like this:

```
I had to rush_
```

□ Undo the deletion with **u**:

```
I had to rush straight from
```

## Deleting an entire line **dd**

To delete an entire line, use the double rule (**dd**). Move the cursor to any location on the line and press **dd**. For this example, leave the cursor where it is and type **dd**. The line will vanish, leaving the cursor on the following line like this:

```
victory sign.

the tennis courts.  There wasn't
```

To restore the deleted line, use **u**:

```
I had to rush straight from
the tennis courts.  There wasn't
```

You can precede the command with a multiplier to delete more than one line. For example, to delete three lines of text (the current line and the next two), type **3dd**. Let's try that now. Three lines will vanish and the screen will look like this:

```
victory sign.

the interview.
```

Restore the missing text with **u**:

```
I had to rush straight from
the tennis courts.  There wasn't
enough time to get ready for
the interview.
```

## Deleting to the beginning of a sentence **d(**

To delete text from anywhere in a sentence to the beginning of the sentence, use **d(**, which is analogous to **c(**. Move the cursor to the first character to be retained and type **d(**. All text in the sentence that precedes the cursor will be deleted. Let's try this.

1. Drop the cursor down a few lines:

   □ Type ***/get*** and press RETURN.

   □ The cursor will land on the g in `get ready`:

```
the tennis courts.  There wasn't
enough time to get ready for
the interview.
```

2. Delete the first part of the sentence:

   □ Type **d(** to delete all words in front of the cursor.

   □ The display should now look like this:

```
the tennis courts.  get ready for
the interview.
```

   □ Restore the deleted words with **u**:

```
the tennis courts.  There wasn't
enough time to get ready for
the interview.
```

## DELETING TO THE END OF A SENTENCE **d)**

To delete text from anywhere in a sentence to the end of the sentence, use **d)**, which is analogous to **c)**. Move the cursor to the first character to be deleted and type **d)**. All text in the sentence that follows the cursor will be deleted.

1. Leave the cursor where it is now:

```
the tennis courts.  There wasn't
enough time to get ready for
the interview.
```

2. Delete the last part of the sentence:

   □ Type **d)** to delete all words that follow the cursor.

   □ The display should now look like this:

```
the tennis courts.  There wasn't
enough time to_
```

   □ Restore the deleted words with **u**:

```
the tennis courts.  There wasn't
enough time to get ready for
the interview.
```

To delete an entire sentence, simply move the cursor to the beginning of the sentence and type **d)**. To delete several sentences, include a multiplier and follow the same procedure.

## Deleting to the beginning of a paragraph **d{**

To delete text from anywhere in a paragraph to the beginning of the paragraph, use **d**{, which is analogous to **c**{. Move the cursor to the first character to be retained and type **d**{. All text in the paragraph that precedes the cursor will be deleted. Without moving the cursor, try this from your present location in the first paragraph. After you type **d**{, the display will look like this:

```
victory sign.
get ready for
the interview.
```

Restore the missing words with **u**:

```
victory sign.

I had to rush straight from
the tennis courts.  There wasn't
enough time to get ready for
the interview.
```

## Deleting to the end of a paragraph **d}**

To delete text from anywhere in a paragraph to the end of the paragraph, use **d**}, which is analogous to **c**}. Move the cursor to the first character to be deleted and type **d**}. All text in the paragraph that follows the cursor will be deleted. Let's move the cursor and try this.

1. Move the cursor into position:

   □ Type ***?rush*** and press (RETURN) to move the cursor up.

   □ The cursor will land on the r in rush:

   ```
   I had to rush straight from
   the tennis courts.  There wasn't
   enough time to get ready for
   the interview.
   ```

2. Delete the rest of the paragraph:

   □ Type **d**} to delete the rest of the paragraph:

   ```
   I had to_
   ```

   □ Now restore the paragraph with **u**:

   ```
   I had to rush straight from
   the tennis courts.  There wasn't
   enough time to get ready for
   the interview.
   ```

To delete an entire paragraph, simply move the cursor to the beginning of the paragraph and type **d**}. To delete several paragraphs, include a multiplier and follow the same procedure.

## 8.3 Shifting text

### Shifting a line

<<
>>

If you ever need to push text to a different location on a line, **vi** provides one command for shifting a line to the left (<<) and one for shifting to the right (>>). Just move the cursor to the desired line and type either << or >>. (The exact amount of the shift will be eight spaces by default. However, you can change this amount by changing the shift-width variable—see Chapter 12, "Customizing **vi**.") To see how shifting works, we'll use the line with the date.

1. Move the cursor into position:

   □ In **vi** command mode, type ***?17*** and press RETURN.

   □ The cursor will now be on 17:

   ```
   September 17, 1987
   ```

2. Shift the date back and forth across the line:

   □ Type >>, then the period (.) a few times to push the date to the right.

   ```
                   September 17, 1987
   ```

   □ Type <<, then the period (.) to push the date back to the left-hand margin.

   ```
   September 17, 1987
   ```

### Shifting more than one line

*n*<<
*n*>>

To shift more than one line, use a multiplier in front of << or >>. To see how this works, we'll indent lines 3, 4, and 5.

1. Move the cursor into position:

   □ In **vi** command mode, type **/R.** and press RETURN.

   □ Now the cursor will be on the `R` in `R.`:

   ```
   Agatha R. Benson
   Vice President
   Fifth National Bank
   ```

2. Shift the three lines back and forth across the line:

   □ Type **3**>> then the period (`.`) several times to push the lines across the screen:

   ```
                                 Agatha R. Benson
                                 Vice President
                                 Fifth National Bank
   ```

   □ Now type **3**<< then the period (`.`) to push the lines back to the left-hand margin:

   ```
   Agatha R. Benson
   Vice President
   Fifth National Bank
   ```

## 8.4 Ending the session

Now it's time to write the updated text to **letter** and return to the UNIX shell.

1. Write the text to file **letter**:

   □ Type **:w** and press RETURN to write to the file.

   □ You will see a display like this at the bottom of the screen:

   ```
   "letter" [Modified] 27 lines, 442 characters
   ```

2. End this session with **vi**:

   □ Type **:q** and press RETURN to end the session.

□ The shell prompt will now appear, indicating that you have left **vi**:

```
$ _
```

Note that **vi** also allows you to combine the write and quit commands into a single command, like this:

```
:wq
$ _
"letter" 27 lines, 442 characters
```

## 8.5 Summary

In this chapter you learned how to change and delete text. You also learned how to shift lines of text. (Each command for changing or deleting text shown here must be terminated with (ESC).)

### CHANGING TEXT

| | Original text | Command | Resulting Text |
|---|---|---|---|
| Word(s) | can't start | **cw**won | won't start |
| | can't start | **cW**cannot | cannotstart |
| | can't start | **c2W**won't begin | won't begin_ |
| Line(s) | This is it. | **c0**That | Thatis it. |
| | This is it. | **c$**was fine. | This was fine. |
| | This is it. | **cc**That was all. | That was all. |
| | This was all<br>we had. | **2cc**There will<br>be more. | There will<br>be more. |
| Paragraph | At the end of<br>the day he left.<br>So we left, too. | **c{**When they<br>left, | When they left,<br>we left, too. |
| | At the end of<br>the day he left.<br>So we left, too. | **c}**everyone<br>left. | At the end of<br>the day he left.<br>So everyone<br>left. |

## Deleting text

| | Original text | Command | Resulting Text |
|---|---|---|---|
| Word(s) | `can't start` | **dw** | `'t start` |
| | `can't start` | **dW** | `start` |
| | `can't start` | **d2W** | _ |
| Line(s) | `This is it.` | **d0** | `is it.` |
| | `This is it.` | **d$** | `This _` |
| | `This is it.` | **dd** | _ |
| | `This was all`<br>`we had.` | **2dd** | _ |
| Paragraph | `At the end of`<br>`the day he left.`<br>`So we left, too.` | **d{** | `we left, too.` |
| | `At the end of`<br>`the day he left.`<br>`So we left, too.` | **d}** | `At the end of`<br>`the day he left.`<br>`So _` |

## Shifting text

| | Original text | Command | Resulting Text |
|---|---|---|---|
| One Line | `At the end of`<br>`the day he left.`<br>`So we left, too.` | >> | `At the end of`<br>`the day he left.`<br>`So we left, too.` |
| Two Lines | `At the end of`<br>`the day he left.`<br>`So we left, too.` | **2**>> | `At the end of`<br>`the day he left.`<br>`So we left, too.` |

# 9

# Finding and Replacing Text

In Chapter 8 you learned different ways to change or delete text with **vi**. In this chapter, you will learn various ways to search for text and make replacements.

## 9.1 Beginning a new session

Here is another brief reminder about how to get started with this session of **vi**: If necessary, move from your home directory to **text**, then start an editing session with the same file (**letter**):

```
$ cd text
$ pwd
/usr/robin/text
$ vi letter
```

Now you should see something like this:

```
September 17, 1987

Agatha R. Benson
Vice President
Fifth National Bank

Dear Mrs. Benson:

I'm sorry you fainted during last
Friday's interview.  You see, I was
just raising my arms to give you the
victory sign.

I had to rush straight from
the tennis courts.  There wasn't
enough time to get ready for
the interview.

I hope you will give very careful
consideration to my qualifications.

Sincerely,
Robert G. Sanders III
~
~
"letter" 27 lines, 442 characters
```

## 9.2 Searching on a line

There are two **vi** commands that allow you to search in either direction for a character on the current line and make changes.

### Searching to the right **f***x*

To search from left to right for a single character on the current line, use the ***f****x* command (where *x* represents the character you are looking for). Then, if you need to continue searching for the same character, you can use one of two commands to repeat:

; to repeat the search in the *same* direction
, to repeat the search in the *opposite* direction

### Searching to the left **F***x*

To search from right to left for a single character on the current line, use the ***F****x* command (where *x* represents the character you are looking for). Again, if you need to continue searching for the same character, you can use one of two commands to repeat:

; to repeat the search in the *same* direction
, to repeat the search in the *opposite* direction

### Moving to a character **T** **t**

To move the cursor to a character on the current line, use the ***T*** command to move to the left or the **t** command to move to the right. Just position the cursor on the line, type ***T****x* or **t***x* (where *x* is the character you are looking for), and the cursor will jump to the space next to the first occurrence of the character.

## Search and change **ct**

By preceding the search command ***t*** with the change command ***c***, you can change a single character on the current line very quickly. For example, suppose you need to change `Dear Mrs. Benson` in the salutation to `Hello Mrs. Benson.`

1. Move the cursor into position:

   □ Move the cursor to the salutation line (***7G*** or ***/Dear***),

   □ The cursor will move to the `D` in `Dear`:

   ```
   Dear Mrs. Benson:
   ```

2. Change `Dear` to `Hello`:

   □ Type ***ctMHello*** (change text from `Dear` to `M` to `Hello` ), and press ESC.

   □ Now the line should look like this:

   ```
   Hello_Mrs. Benson:
   ```

   □ Type ***u*** to restore the line:

   ```
   Dear Mrs. Benson:
   ```

## Search and delete **dt**

Similarly, you can precede ***t*** with the delete command ***d*** to delete text. For example, suppose you need to change `Dear Mrs. Benson` to `Mrs. Benson`. Since the cursor is already on the same line, just type ***dtM*** (delete all text from here to the first `M` to the right) and press ESC. Now the line should look like this:

```
Mr. Benson:
```

Restore the line with ***u***.

## Search to the left

In both the examples above we were searching to the right (***t***). By using ***T*** instead we could also search to the left. In a search to the left, we could use commands like ***cTMMrs.*** (change all text to the first `M` to the left to `Mrs.` ) and ***dTr*** (delete all text to the first `r` to the left). Try each of these commands; then undo the results in each case with ***u***.

## 9.3 Searching in a file

To help you search for a string anywhere in the work area, **vi** provides a pair of commands analogous to ***f*** and ***F***: one for searching forward and one for searching backwards in the editing buffer.

### Searching forward * /string*

To search forward for a word in a file, type */string* and press RETURN. If you need to continue searching for the word, you can use one of two keys:

***n*** Repeat the search in the same direction (forward)
***N*** Repeat the search in the opposite direction (backwards)

As an example, suppose you need to search for `you` in the letter.

1. Move the cursor into position:

   □ Position the cursor at the beginning of the file (***1G***).

   □ The first line should look like this:

   ```
   September 17, 1986
   ```

2. Begin the search:

   □ Type ***/you*** and press RETURN.

   □ The cursor will jump to the `y` in `you` in line 9:

   ```
   I'm sorry you fainted during last
   ```

3. Continue the search:

   □ Type ***n*** to continue the search forward.

   □ The cursor will jump to the `you` in line 11:

   ```
   just raising my arms to give you the
   ```

4. Continue the search again:

   □ Type ***n*** again.

   □ The cursor will move to `you` in line 19:

   ```
   I hope you will give very careful
   ```

You can type ***n*** again to continue searching forward or ***N*** to search back toward the beginning of the letter. If you don't want the cursor to stop at `your`, `yourself`, or other words that contain `you`, leave a space after the ***u*** when you begin the search, like this: ***/you*** .

## SEARCHING BACKWARD *?string*

To search backwards for a word in a file, type *?string* and press (RETURN). Again, if you need to continue searching for the word, use one of these two keys:

***n*** Repeat the search in the same direction (backwards)
***N*** Repeat the search in the opposite direction (forward)

For example, suppose you need to search for `you` from the end of the letter.

1. Move the cursor into position:

   □ Position the cursor at the end of the file (***G***).

   □ The line should look like this:

   ```
   Robert G. Sanders III
   _
   ```

2. Begin the search:

   □ Type ***?you*** and press (RETURN) to begin searching in reverse.

   □ The cursor will jump to the `y` in `you` in line 19:

   ```
   I hope you will give very careful
   ```

3. Continue the search:

   □ Type ***n*** to continue searching in reverse.

   □ The cursor will jump to the `you` in line 11:

   ```
   just raising my arms to give you the
   ```

4. Continue the search again:

   □ Type ***n*** again to continue searching in reverse.

   □ The cursor will move to `your` in line 9:

   ```
   I'm sorry you fainted during last
   ```

Type ***n*** again to continue searching backwards or ***N*** to search forward. Again, if you don't want the cursor to stop at `your`, `yourself`, or other words that contain `you`, leave a space after ***you*** when you begin the search, like this: ***?you*** . Since all searches wrap from one end of the editing buffer to the other, the two commands (*/string* and *?string*) are practically equivalent.

## 9.4 Making replacements

Each time the cursor stops during a search, you can replace text (or append or insert text, for that matter). Since appending and inserting text have already been discussed in detail, we'll concentrate on replacements here. To replace text in a file, **vi** offers the following basic commands (in addition to the change command discussed in the previous chapter): one that replaces a single character with another, one that replaces a sequence of characters one at a time, one that replaces a single character with more than one, and one that replaces an entire line.

### Replacing one character with another **r**

To replace one character with another, **vi** provides a simple command—***r***. Move the cursor to the character, type ***r***, and then type the new character. For example, suppose you want to change the date from `September 17` to `September 27`.

1. Move the cursor into position:

   □ Type ***?17*** to move the cursor to the `17`.

   □ Now the line should look like this:

   ```
   September 17, 1986
   ```

2. Make the change:

   □ Type ***r2*** *without* RETURN or ESC to change `1` to `2`.

   □ Now the line should look like this:

   ```
   September 27, 1986
   ```

### Replacing characters one at a time **R**

To replace several consecutive characters one at a time, use the ***R*** command. Move the cursor to the first of the characters, type ***R***, and then type the new characters, followed by RETURN. For example, suppose you want to change the date from `September 27` to `August 27`.

1. Move the cursor into position:

   □ Type ***b*** to move the cursor to the `S` in `September`.

   □ Now the line should look like this:

   ```
   September 27, 1986
   ```

2. Make the change:

   □ Type ***R***, then ***August*** without a space, followed by (ESC), to type `August` over `September` (`Augustber`).

   □ After you type ***dw*** to delete `ber`, the line should look like this:

```
August 27, 1986
```

**Caution:** If the replacement text is shorter than the text being replaced, there will be leftover characters; if the replacement text is longer than the text being replaced, you will type beyond the original text over other text.

## Replacing one character with several **s**

To replace one character with any number of characters, **vi** provides the ***s*** command. Move the cursor to the character, type ***s***, and then type the new characters. For example, suppose you wanted to change the name on the next line from `Benson` to `Bertson`.

1. Move the cursor into position:

   □ Type ***/nson*** to advance the cursor to the first `n` in `Benson`.

   □ The line should look like this:

```
Dear Mrs. Benson:
```

2. Make the change:

   □ Type ***srt***, followed by (ESC), to change `n` to `rt`.

   □ Now the line should look like this:

```
Dear Mrs. Bertson:
```

## Replacing an entire line **S**

To replace an entire line of text, use the ***S*** command (which is equivalent to ***cc***). Move the cursor to any location on the line, type ***S***, and then type the new line (or lines), followed by (RETURN). For example, suppose you want to change `Vice President` to `Senior Manager`.

1. Move the cursor into position:

   □ Type ***/Vice*** to move the cursor to the line.

   □ The line should look like this:

```
Vice President
```

2. Make the change:

   □ Type ***S*** to request a change to the line (which will vanish).

   □ Without leaving a space, type ***Senior Manager***, followed by ESC, to type the new line.

   □ The new line should look like this:

   ```
   Senior Manager
   ```

### Ending the session

Now it's time to abandon this text and return to the shell. Type ***:q!*** and press RETURN to abandon the text and quit **vi**.

## 9.5 Making substitutions

The **ex** editor has a substitution command called ***s*** that allows you to replace one string with another.

### Starting a new session with vi

First we'll enter a program called **metric.c** in your **text** directory; then we can make changes later with **vi**. First, let's start a new editing session:

```
$ vi metric.c
$ _
```

Now type ***a*** to append and type the following:

```
/* Convert gallons to liters   */
main()
{
   int low, high, step;
   float gals, ltrs;

   low = 10; high = 20; step = 2;

   printf("%4s \t %6s \n\n", "gals", "liters");

   gals = low;
   while (gals <= high)
   {
      ltrs = (gals * 3.785);
      printf("%4.0f \t %6.2f\n", gals, ltrs);
      gals += step;
   }
}
```

## Changing some settings

Since this is a program, not a letter, let's makes some changes to the basic settings in **vi**. To make it simple, we'll change only two of these:

**number (nu)** The option that produces line-numbering

**shift-width (sw)** The option that determines how far across the screen lines will be shifted by the *shift* commands **<<** and **>>**

1. Turn on line numbering:

   □ Type ***:set nu*** and press (RETURN).

   □ Before you press (RETURN), you will see `:set nu` at the bottom of the screen.

   □ After you press (RETURN), you will see line numbers appear at the lefthand margins, with the text indented.

2. Set the shift-width to 5:

   □ Type ***:set sw=5*** and press (RETURN).

   □ Before you press (RETURN), you will see `:set sw=5` at the bottom of the screen.

   □ The width for all shifts will be changed from 8 (the default) to 5.

At this point you have **vi** active with file **metric.c** and you have line-numbering turned on and option `shift-width` set to five columns. Now you are ready to begin trying out some substitutions.

## Making substitutions **:s**

One reason for selecting this file to make substitutions is that it contains so many repetitions of the same words. For example, the variable `gals` occurs seven different times in this short program. You could probably make seven replacements manually without much bother, but the substitution command (***s***) makes these changes automatically. In a long program, this would be an enormous convenience.

1. Change `gals` to `gallons`:

   □ Type ***:1,$s*** to request a substitution for all lines in the file (line 1 to the last line (`$`)).

   □ Without leaving a space, type ***/gals/gallons/g*** to request a substitution of `gallons` in place of `gal` for all occurrences on each line (`g`).

- □ If the display at the bottom of the screen looks like this, press (RETURN) to begin the substitution:

```
:1,$s/gals/gallons/g
```

- □ If you see mistakes, back up the cursor and correct them before you press (RETURN).

2. Change `ltrs` to `liters`:

- □ Type ***:1,$s*** to request a substitution for all lines in the file (line 1 to the last line ($)).
- □ Without leaving a space, type ***/ltrs/liters/g*** to request a substitution of `liters` in place of `ltrs` for all occurrences on each line (g).
- □ If the display at the bottom of the screen looks like this, press (RETURN) to begin the substitution:

```
:1,$s/ltrs/liters/g
```

- □ If you see mistakes, back up the cursor and correct them before you press (RETURN).

If you ever enter a substitution command incorrectly and then press (RETURN) by accident, you can always undo the results with the ***u*** (undo) command.

3. Look at the results on the screen:

```
/* Convert gallons to liters   */
main()
{
   int low, high, step;
   float gallons, liters;

   low = 10; high = 20; step = 2

   printf("%4s \t %6s \n\n", "gallons", "liters");

   gallons = low;
   while (gallons <= high)
   {
      liters = (gallons * 3.785);
      printf("%4.0f \t %6.2f\n", gallons, liters);
      gallons += step;
   }
}
~
~
```

```
~
3 substitutions on 3 lines
```

The substitution command uses many of the conventions for searching that are used throughout UNIX. Here are a few words about using ***s*** in **vi**:

- The ***s*** command always begins with a colon (`:`) to escape to **ex**, followed by a range of line numbers.
- The line numbers may be actual integers, search strings, or symbols, as illustrated here:

| | |
|---|---|
| ***:1,20*** | Line 1 to line 20 |
| ***:15,/place/*** | Line 15 to the first line that contains `place` |
| ***:5,.*** | Line 5 to the current line |
| ***:.,$*** | The current line to the last line of the file |
| ***:.-20,?end?*** | Twenty lines ahead of the current line to the first line that contains `end` |
| ***:?WPP?,$-2*** | The first line that contains `WPP` (above the current line) to two lines before the last line of the file |

- Use slashes to search forward and question marks to search backwards.
- Use the ***g*** command at the end of the command line to substitute all occurrences in each line, rather than just the first occurrence.

## 9.6 Shifting text

### Shifting lines

<< <br>
>>

Try the new setting for shiftwidth by moving to various lines and typing << or >>. Each shift will be five columns now instead of eight.

### Shifting paragraphs

<} <br>
>}

Try shifting paragraphs (in this case, groups of lines separated by blank lines) with the <} and >} operators.

1. Shift lines 4 and 5:

   □ Type **4G** to move the cursor to line 4.

□ Type >} to shift the two lines (a paragraph to **vi**) to the right:

```
        int low, high, step;
        float gallons, liters;
```

□ Type <} to shift the lines back again:

```
int low, high, step;
float gallons, liters;
```

2. Shift lines 11–17:

□ Type **11G** to move the cursor to line 11.

□ Type >} to shift the lines to the right:

```
        gallons = low;
        while (gallons <= high)
        {
           liters = (gallons * 3.785);
           printf("%4.0f \t %6.2f\n", gallons, liters);
           gallons += step;
        }
    }
```

□ Type <} to shift the lines back again:

```
    gallons = low;
    while (gallons <= high)
    {
       liters = (gallons * 3.785);
       printf("%4.0f \t %6.2f\n", gallons, liters);
       gallons += step;
    }
}
```

You can also shift sentences with **vi**. The commands to use are <**)** and >**)**. Now end this session with **vi** by typing **:wq** and pressing RETURN.

## 9.7 Summary

In this chapter you learned how to search for text, how to make replacements, and how to shift text.

### Searching on a line

| | | | |
|---|---|---|---|
| **F**$x$ | Search left | **f**$x$ | Search right |
| **T**$x$ | Move left to $x$ | **t**$x$ | Move right to $x$ |

## Searching for a string

| | |
|---|---|
| /*string* | Searching forward |
| **?***string* | Searching backward |

## Making replacements

| | |
|---|---|
| **r** | Replacing one character with another |
| **R** | Replacing any number of characters |
| **s** | Replacing one character with any number of characters |
| **S** | Replacing an entire line |

## Making substitutions

| | |
|---|---|
| **:***line n*,*line N***s/***old string***/***new string***/g** | (Search forward) |
| **:***line n*,*line N***s?***old string***?***new string***?g** | (Search backward) |

## Shifting text

| Shift Left | Text | Shift Right |
|---|---|---|
| << | Line | >> |
| *n*<< | *n* Lines | *n*>> |
| <} | Paragraph | >} |

# 10

# Moving and Copying within a File

In Chapter 9 you learned how to search for text and make replacements. In this chapter you will learn how to move and copy text within a file. But first, here are a few words about ways to exit from **vi**.

## 10.1 Exiting **vi**

So far you have used only the **w** (write) and **q** (quit) commands to exit a **vi** file. This includes the combined command **wq** (write and quit):

```
:wq (RETURN)
```

### CONDITIONAL WRITING

To perform a write before quitting *only* if you have made changes (that is, to request a *conditional write*, you can use ***:x*** (RETURN) or ***ZZ***

### ABANDONING A FILE

To abandon a file (and not save any changes you might have made to it), use the *forced quit*, like this:

```
:q! (RETURN)
```

## 10.2 Moving text within a file

In Chapter 8 you learned how to use the change (**c**) and delete (**d**) commands, which operate in similar ways. In this chapter you will learn two new commands called *yank* (**y**) and *put* (**p** or **P**). You will learn that delete and yank are also similar, and that you can use either one of them with put to make text disappear, reappear, or multiply.

If you think of yourself as a magician and the words, sentences, and paragraphs you work with as handkerchiefs, rabbits, and assistants, then you won't have any trouble seeing how *delete*, *yank*, and *put* work. Using *delete* is like putting a rabbit into a hat and then showing the audience an empty hat. Using *yank* is like putting a handkerchief into your pocket while still keeping another in your hand. Using *delete* and *put* in succession is like having your assistant step into a wooden box, and then reappear at the other end of the stage. Using *yank* and *put* in succession is like having your assistant step into a wooden box, and then emerge from one box while a twin emerges from another.

## Creating a new file

Before discussing how to move text from one location to another, let's create a new file in directory **text** called **wall**. Use **vi** to edit a file named **wall**:

```
$ vi wall
```

Type **a** and enter the following text (leaving a blank space and pressing RETURN at the end of each line, and pressing ESC at the conclusion of the text):

```
Request for Wall
                                                   [Blank line]
Beijing (Peking).  The Mayor of West Berlin
stood yesterday on top of the Great Wall
of China next to Deng Xiaoping, leader
of the nation of one billion people.
                                                   [Blank line]
Begun during the Ch'in dynasty (about
the time Rome fought the first Punic
War), the wall was not completed until
the Ming dynasty (about the time the
Mayflower arrived at Plymouth Rock).
                                                   [Blank line]
Some 25 feet high, 15 to 30 feet wide at
the base, and 12 feet wide at the top,
the wall is over 1,500 miles (2,400 km)
long, greater than the distance between
New York and Dallas.
                                                   [Blank line]
The purpose of the wall was to protect
China against invaders from the north.
The mayor told his host, "This wall was
here to keep people out.  We have a
wall that is there to keep people in."
ESC
```

Now write the text to file **wall** (**:w**). You will see this message at the bottom of the screen:

```
"wall" [New file] 25 lines, 746 characters
```

## Transposing characters **xp**

To move text from one location to another, you can make it disappear from one location (with *delete*), then reappear in another (with *put*).

After writing to **wall**, move the cursor to the `o` in `Plymouth` on line 12 (***?outh*** twice). Now, with the cursor resting on the `o`, type **xp** (no RETURN is necessary). The word will now be spelled `Plymuoth`—the `o` and the `u` have been switched. This is the simplest kind of move.

Now type **h** (cursor left), followed by **xp** again, to undo the switch, and try it again. Here's what happens in slow motion:

1. The cursor is on the `o` (keep your eye on the cursor):

```
Mayflower arrived at Plymouth Rock).
```

2. Now delete the `o` with the **x** command:

```
Mayflower arrived at Plymuth Rock).
```

3. Then put the `o` back with the **p** command:

```
Mayflower arrived at Plymuoth Rock).
```

4. Undo the change by typing **U** (not **u**)—**U** is for the entire line.

Did you keep your eye on the cursor? As soon as the `o` vanishes, the cursor moves forward to the `u`. Since the `p` command always puts the text *ahead* of the cursor, the `o` ends up on the other side of the `u`. (Remember, the command to delete a single character is **x**, not **d**.)

## Moving words **dw and p**

Moving words is like moving characters. The main difference is that the command to delete a word is **dw** (or **dW**), not **x**. We plan to move the cursor to two words, delete the words, then move to the new location and put the words there. As an example, let's move `Beijing (Peking)` from line 3 to line 18.

1. Move the cursor into position:

   □ Type ***?Bei*** and press RETURN to move the cursor to `Beijing`.

   □ The display should now look like this:

```
Beijing (Peking).  The Mayor of West Berlin
stood yesterday on top of the Great Wall
of China next to Deng Xiaoping, leader
of the nation of one billion people.
```

2. Delete the two words at their current location:

   □ Type **d2W** to delete `Beijing (Peking).`

   □ The line should now look like this:

```
_ The Mayor of West Berlin
```

3. Move the cursor to the new location:

   □ Type ***/Dal*** and press RETURN to move the cursor to `Dallas` in line 18.

   □ The line should now look like this:

```
New York and Dallas.
```

4. Put the word at the new location:

   □ Type capital **P** (not **p**) to put `Beijing (Peking)` in front of `Dallas.`

   □ The paragraph should now look like this:

```
Some 25 feet high, 15 to 30 feet wide at
the base, and 12 feet wide at the top,
the wall is over 1,500 miles (2,400 km)
long, greater than the distance between
New York and Beijing (Peking). _Dallas.
```

In this example, you used a capital **P** instead of **p** because capital **P** puts text *before* the cursor (*above* the current line in a line operation). Use **p** when you want to put text *after* the cursor (or *below* the current line).

## Moving a sentence

**d) and P**
**d) and p**

Moving a sentence is like moving a word. The main difference is that we use **d)** to delete a sentence, not **dw**. We plan to move the cursor to the beginning of a sentence, delete the sentence, then move to the new location and put the sentence there. As an example, let's move the first sentence of the last paragraph to the end of the paragraph.

1. Move the cursor into position:

   - □ Type ***/The*** and press (RETURN) to move the cursor to `The purpose`.
   - □ The display should now look like this:

```
The purpose of the wall was to protect
China against invaders from the north.
The mayor told his host, "This wall was
here to keep people out.  We have a
wall that is there to keep people in."
```

2. Delete the sentence at its current location:

   - □ Type **d)** to delete the entire sentence.
   - □ The paragraph should now look like this:

```
The mayor told his host, "This wall was
here to keep people out.  We have a
wall that is there to keep people in."
```

3. Move the cursor to the new location:

   - □ Type **L** to move to the blank line at the end.
   - □ The screen should now look like this:

```
wall that is there to keep people in."
_
```

4. Put the sentence at the new location:

   - □ Type lowercase **p** to put the first sentence after the second.
   - □ The paragraph should now look like this:

```
The mayor told his host, "This wall was
here to keep people out.  We have a
wall that is there to keep people in."

The purpose of the wall was to protect
China against invaders from the north.
```

## MOVING A PARAGRAPH

**d**} and **P**
**d**} and **p**

Moving a paragraph is like moving a sentence. The main difference is that the command to delete a paragraph is **d**}, not **d)**. We plan to move the cursor to the start of a paragraph, delete the paragraph, then move to the new location and put the paragraph there. As an example, let's move the second paragraph to where the third paragraph is now (switch paragraphs).

1. Move the cursor into position:

   □ Type **4**{ to move the cursor to the start of the second paragraph.

   □ The display should now look like this:

```
of China next to Deng Xiaoping, leader
of the nation of one billion people.
_
Begun during the Ch'in dynasty (about
the time Rome fought the first Punic
```

2. Delete the paragraph at its current location:

   □ Type **d**} to delete the entire second paragraph.

   □ The display should now look like this:

```
of China next to Deng Xiaoping, leader
of the nation of one billion people.
_
Some 25 feet high, 15 to 30 feet wide at
the base, and 12 feet wide at the top,
```

3. Move the cursor to the new location:

   □ Type } to advance the cursor to the beginning of the next paragraph.

   □ The display should now look like this:

```
long, greater than the distance between
New York and Dallas.
_
The mayor told his host, "This wall was
here to keep people out.  We have a
```

4. Put the paragraph at the new location:

   □ Type capital **P** to put the second paragraph above the fourth.

   □ The display should now look like this (with moves highlighted):

Request for Wall

The Mayor of West Berlin
stood yesterday on top of the Great Wall
of China next to Deng Xiaoping, leader
of the nation of one billion people.

Some 25 feet high, 15 to 30 feet wide at
the base, and 12 feet wide at the top,
the wall is over 1,500 miles (2,400 km)
long, greater than the distance between
New York and **Beijing (Peking)**. Dallas.

**Begun during the Ch'in dynasty (about**
**the time Rome fought the first Punic**
**War), the wall was not completed until**
**the Ming dynasty (about the time the**
**Mayflower arrived at Plymouth Rock).**

The mayor told his host, "This wall was
here to keep people out. We have a
wall that is there to keep people in."

**The purpose of the wall was to protect**
**China against invaders from the north.**

## Moving lines

**dd** and **P**
**dd** and **p**

Moving a line is like moving a word. The main difference is that to delete a line we use **dd**, not **dw**. We plan to move the cursor to a line, delete two lines, then move to the new location and put the lines there. As an example, let's move the headline (and the blank line below it) to the space between the first and second paragraphs.

1. Move the cursor into position:

   □ Type **1G** and press (RETURN) to move the cursor to the headline.

   □ The display should now look like this:

   Request for Wall

   The Mayor of West Berlin
   stood yesterday on top of the Great Wall
   of China next to Deng Xiaoping, leader
   of the nation of one billion people.

2. Delete the lines at their current location:

□ Type **2dd** to delete the two lines.

□ The display should now look like this:

```
The Mayor of West Berlin
stood yesterday on top of the Great Wall
of China next to Deng Xiaoping, leader
of the nation of one billion people.
```

3. Move the cursor to the new location:

□ Type ***/Some*** to move the cursor to `Some` in the next paragraph.

□ The paragraph should look like this:

```
Some 25 feet high, 15 to 30 feet wide at
the base, and 12 feet wide at the top,
the wall is over 1,500 miles (2,400 km)
long, greater than the distance between
New York and Beijing (Peking) Dallas.
```

4. Put the line at the new location:

□ Type capital **P** to put the headline *above* the current line.

□ The screen should now look like this:

```
of China next to Deng Xiaoping, leader
of the nation of one billion people.

Request for Wall
_
Some 25 feet high, 15 to 30 feet wide at
the base, and 12 feet wide at the top,
```

## Other possible moves

In the examples you just tried, you moved one entire item (character, word, sentence, paragraph, or line) from one location to another. However, it is also possible to move part of an item (assuming the cursor is somewhere in the middle of it), or several items. Here are some suggestions:

| | |
|---|---|
| **db** | The left-hand side of a word |
| **d(** | The beginning of a sentence |
| **d{** | The beginning of a paragraph |
| **d^** | The left-hand side of a line |

| | |
|---|---|
| **dw** | The right-hand side of a word |
| **d)** | The ending of a sentence |
| **d}** | The ending of a paragraph |

| | |
|---|---|
| **d$** | The right-hand side of a line |

| | |
|---|---|
| **x3** | Three characters |
| **d3w** | Three words |
| **d3)** | Three sentences |
| **d3}** | Three paragraphs |
| **3dd** | Three lines |

| | |
|---|---|
| **P** | Before the cursor (or above the line) |
| **p** | After the cursor (or below the line) |

### Abandoning the file

Before going on to *copying* text within the editing buffer, let's abandon the changes we've made to the text in our work area. Then we can start all over again with **wall**. Type **:q!** and press (RETURN) to abandon the work area.

## 10.3 Copying text within a file

Copying text from one location to another with **vi** is similar to moving text. The only difference is that you use the yank command (**y**) instead of the *delete* command (**d**). Yanking leaves a copy of the text in its original location. Everything else is about the same. Let's begin by starting a new session:

```
$ vi wall
```

### Copying a word

**yw and P**
**yw and p**

We'll dispense with the subject of copying a character, and move right on to copying a word. *Copying* a word is like *moving* a word. The main difference is that you will yank the word, rather than delete it. We plan to move the cursor to a word, yank the word, then move to the new location and put the word there. As an example, let's copy `Great` from line 4 to several other places.

1. Move the cursor into position:

   □ Type ***/Great*** and press (RETURN) to move the cursor to `Great`.

   □ The display should now look like this:

```
Beijing (Peking).  The Mayor of West Berlin
```

```
stood yesterday on top of the Great Wall
of China next to Deng Xiaoping, leader
of the nation of one billion people.
```

2. Yank the word at its current location:

   □ Type **yw** to yank `Great`.

   □ The line should now look like this (the same):

   ```
   stood yesterday on top of the Great Wall
   ```

3. Move the cursor to a new location:

   □ Type ***/wall*** and press (RETURN) to move the cursor to `wall` in line 8.

   □ The line should now look like this:

   ```
   War), the wall was not completed until
   ```

4. Put the word at the new location:

   □ Type capital **P** to put `Great` in front of `wall`.

   □ The line should now look like this:

   ```
   War), the Great_wall was not completed until
   ```

Typing **n** twice, then repeating step 4 (that's **n n P**), insert copies of `Great` in front of the other three occurrences of `wall`. After you have done this, the fourth paragraph should look like this:

```
The purpose of the Great wall was to protect
China against invaders from the north.
The mayor told his host, "This wall was
here to keep people out.  We have a
wall_that is there to keep people in."
```

## Copying a sentence

**y) and P**
**y) and p**

Copying a sentence is like copying a word. The main difference is that the command to yank a sentence is **y)**, not **yw**. We plan to move the cursor to the beginning of a sentence, yank the sentence, then move to the new location and put the sentence there. As an example, let's copy the first sentence of the last paragraph to the end of the paragraph.

1. Move the cursor into position:

- □ Type ***?The p*** and press RETURN to move the cursor to the beginning of the last paragraph.
- □ The display should now look like this:

```
The purpose of the Great wall was to protect
China against invaders from the north.
The mayor told his host, "This wall was
here to keep people out.  We have a
wall that is there to keep people in."
```

2. Yank the sentence at its current location:

- □ Type **y)** to yank the entire sentence.
- □ The paragraph should look the same:

```
The purpose of the Great wall was to protect
China against invaders from the north.
The mayor told his host, "This wall was
here to keep people out.  We have a
wall that is there to keep people in."
```

3. Move the cursor to the new location:

- □ Type **G** to move the cursor to the blank line below the paragraph.
- □ The display should now look like this:

```
here to keep people out.  We have a
wall that is there to keep people in."
_
```

4. Put the sentence at the new location:

- □ Type **p** to put the sentence.
- □ The paragraph should now look like this:

```
The purpose of the Great wall was to protect
China against invaders from the north.
The mayor told his host, "This wall was
here to keep people out.  We have a
wall that is there to keep people in."

The purpose of the Great wall was to protect
China against invaders from the north.
```

## Copying a paragraph

**y**} and **P**
**y**} and **p**

Copying a paragraph is like copying a sentence. The main difference is that we use **y**} to yank a paragraph, not **y)**. We plan to move the cursor to the start of a paragraph, yank the paragraph, then move to the new location and put the paragraph there. As an example, let's make a copy of the third paragraph after the second.

1. Move the cursor to the beginning of the third paragraph by typing **3**{. This is how the display should look now:

```
the Ming dynasty (about the time the
Mayflower arrived at Plymouth Rock).

Some 25 feet high, 15 to 30 feet wide at
the base, and 12 feet wide at the top,
```

1. Yank the paragraph at its current location:

   □ Type **y**} to yank the entire third paragraph.

   □ The display should look the same:

```
the Ming dynasty (about the time the
Mayflower arrived at Plymouth Rock).

Some 25 feet high, 15 to 30 feet wide at
the base, and 12 feet wide at the top,
```

2. Move the cursor to the new location:

   □ Type { to move the cursor to the beginning of the previous paragraph.

   □ The display should now look like this:

```
of China next to Deng Xiaoping, leader
of the nation of one billion people.

Begun during the Ch'in dynasty (about
the time Rome fought the first Punic
```

3. Put the paragraph at the new location:

   □ Type capital **P** to put the third paragraph *above* the second.

   □ The display should now look like this (copies highlighted):

```
Request for Wall

Beijing (Peking).  The Mayor of West Berlin
stood yesterday on top of the Great Wall
of China next to Deng Xiaoping, leader
of the nation of one billion people.

Some 25 feet high, 15 to 30 feet wide at
the base, and 12 feet wide at the top,
the Great wall is over 1,500 miles (2,400 km)
long, greater than the distance between
New York and Dallas.

Begun during the Ch'in dynasty (about
the time Rome fought the first Punic
War), the Great wall was not completed until
the Ming dynasty (about the time the
Mayflower arrived at Plymouth Rock).

Some 25 feet high, 15 to 30 feet wide at
the base, and 12 feet wide at the top,
the Great wall is over 1,500 miles (2,400 km)
long, greater than the distance between
New York and Dallas.

The purpose of the Great wall was to protect
China against invaders from the north.
The mayor told his host, "This wall was
here to keep people out.  We have a
wall that is there to keep people in."

The purpose of the Great wall was to protect
China against invaders from the north.
```

## Copying a line

**yy** and **P**
**yy** and **p**

Copying a line is like all the other copies. This time we use **yy** to yank a line. We plan to move the cursor to a line, yank the line, then move to the new location and put the line there. As an example, let's copy the headline (and the blank line below it) to several other places.

1. Move the cursor into position:

   □ Type ***?Request*** and press (RETURN) to move the cursor to `Request` in the headline.

   □ The display should now look like this:

```
Request for Wall

Beijing (Peking).  The Mayor of West Berlin
```

```
stood yesterday on top of the Great Wall
```

2. Yank the two lines at their current location:

   □ Type **2yy** to yank the two lines.

   □ The display should look the same:

```
Request for Wall

Beijing (Peking).  The Mayor of West Berlin
stood yesterday on top of the Great Wall
```

3. Move the cursor to the new location:

   □ Type ***/Begun*** and press RETURN to move to the beginning of the next paragraph.

   □ The screen should look like this:

```
of China next to Deng Xiaoping, leader
of the nation of one billion people.

Begun during the Ch'in dynasty (about
the time Rome fought the first Punic
```

4. Put the line at the new location:

   □ Type a capital **P** to put the lines above the current line.

   □ The screen should now look like this:

```
long, greater than the distance between
New York and Dallas.

Request for Wall

Begun during the Ch'in dynasty (about
the time Rome fought the first Punic
```

## Other possible copies

In the examples you just tried, you copied one entire item (word, sentence, paragraph, or line) from one location to another. However, you can also copy part of an item (assuming the cursor is somewhere in the middle of it), or several items. Here are some suggestions:

| | |
|---|---|
| **yb** | The left-hand side of a word |
| **y(** | The beginning of a sentence |
| **y{** | The beginning of a paragraph |

| | |
|---|---|
| **y^** | The left-hand side of a line |
| | |
| **yw** | The right-hand side of a word |
| **y)** | The ending of a sentence |
| **y}** | The ending of a paragraph |
| **y$** | The right-hand side of a line |
| | |
| **y3w** | Three words |
| **y3)** | Three sentences |
| **y3}** | Three paragraphs |
| **3yy** | Three lines |
| | |
| **P** | Before the cursor (or above the line) |
| **p** | After the cursor (or below the line) |

### Abandoning the file

Before leaving this chapter, let's abandon the text in our work area. Then we can start all over again with the original version of **wall** in the next chapter. Type **:q!**(RETURN) to abandon the work area.

## 10.4 Summary

In this chapter you learned different ways to exit **vi**, along with methods for moving and copying text within a file.

### Exiting **vi**

To write the text in the work area to a file and end the session, type **:wq**(RETURN). To write the text only if there have been changes and quit, type **:x**(RETURN) or **ZZ**. To abandon the text in the work area (thereby retaining the text previously stored in the file), type **:q!**(RETURN).

### Moving text within a file

In general, the procedure for moving (or copying) text from one location to another in the same file is to delete (or yank) the text from its current location, then put the text into the new location. A special case of moving is transposing two characters. To transpose two adjacent characters, move the cursor to the first character and type **xp**.

| Text | Delete | | Yank | |
|---|---|---|---|---|
| Word(s) | **dW** | **d5W** | **yW** | **y5W** |
| | **dw** | **d5w** | **yw** | **y5w** |
| Sentence(s) | **d(** | **d3(** | **y(** | **y3(** |
| | **d)** | **d3)** | **y)** | **y3)** |
| Line(s) | **d0** | **4d0** | **y0** | **4y0** |
| | **dd** | **4dd** | **yy** | **4yy** |
| | **d$** | **4d$** | **y$** | **4y$** |
| Paragraph(s) | **d{** | **d2{** | **y{** | **y2{** |
| | **d}** | **d2}** | **y}** | **y2}** |

| | Put |
|---|---|
| In front of the cursor or above the current line | **P** |
| After the cursor or below the current line | **p** |

# 11

# Working with More Than One File

In Chapter 10 you learned how to move and copy text from one location in a file to another. In this chapter you will learn how to move and copy text from one file to another, which involves editing a second file without leaving **vi**. You will also learn how to tag a file and how to begin an editing session with various options.

## 11.1 Editing another file

In **vi**, as in **ed**, it is possible to begin editing a new file without having to leave your editor and return to the shell. There are several commands you can use, depending on how you want to save your current file.

### SAVE AND EDIT **:e**

If you have already saved your current file (**:w**), you can begin editing another file without leaving **vi** using the **:e** (edit) command. The sequence is as follows:

| | |
|---|---|
| **:w** *old file* RETURN | Save the old file |
| **:e** *new file* RETURN | Begin editing a new file |

### ABANDON AND EDIT **:e!**

If you don't want to save your current file, you can begin editing another file immediately with the **:e!** (edit!) command. The sequence is as follows:

| | |
|---|---|
| **:e!** *new file* RETURN | Abandon the old file and begin editing a new file |

### AUTO-EDIT **:n**

If you want to have your current file saved automatically whenever you begin work on another file, you have to have the *auto-write* option set with the **:set** command (as described in Chapter 12, Customizing **vi**). Once that has been done, you can then use the **:n** command to begin editing a new file. The sequence is as follows:

**:n** *new file* (RETURN) Begin editing a new file (the old file is automatically saved)

There's another variation of the **:n** command that you can use during an editing session with more than one file. Suppose you have invoked **vi** by naming three files to be edited, like this:

```
$ vi file.1 file.2 file.3
```

This editing session will begin with **file.1**. At the conclusion of your work with **file.1**, you can type **:n** by itself without a filename. If you have the *auto-write* feature set, the text in the editing buffer will automatically be written to **file.1** and **file.2** will be read into the editing buffer. After you're finished with **file.2**, type **:n** again to write to **file.2** and call in **file.3**. Let's try this feature right now.

1. Prepare for auto-editing:

   □ Begin an editing session with three different files:

   ```
   $ vi letter metric.c wall
   3 files to edit
   "letter" 27 lines, 442 characters
   ```

   □ When the text of **letter** appears on the screen, set *auto-write* mode:

   ```
   :set aw
   ```

2. Perform auto-editing:

   □ Without making any changes to **letter**, proceed to the next file (**metric.c**):

   ```
   :n
   "metric.c" 18 lines, 331 characters
   ```

   □ Without making any changes to **metric.c**, proceed to the next file (**wall**):

   ```
   :n
   "wall" 25 lines, 746 characters
   ```

3. End this short session:

   ***:q!***

So here you have a very handy editing tool that allows you to call up a number of files at the same time and move from one to another. If you try to move a third time in this example, you will see this: `No more files to edit.`

## 11.2 Moving text between files

Moving text from one file to another is similar to moving text within a file. There are only two differences:

- You must leave the text in a temporary storage area (a *buffer*) with a one-letter name
- You have to call up the new file without leaving **vi**, using one of the commands you learned earlier in this chapter:

| | |
|---|---|
| **:e** *file* (RETURN) | Edit after save (the save is required) |
| **:e!** *file* (RETURN) | Abandon text and edit |
| **:n** *file* (RETURN) | Edit with automatic save |

### Beginning a new session

Before moving any text, let's begin a new session of **vi** with **wall**. Type ***vi wall*** and press (RETURN):

```
$ vi wall
```

### Moving a sentence

**"xd)** and **"xP**
**"xd)** and **"xp**

Moving a sentence to another file is like moving a sentence within a file. We plan to move the cursor to a sentence, delete the sentence *to a buffer*, then move to the location in the new file and put the sentence there *from the same buffer*. As an example, let's move a sentence from **wall** to the end of **letter**.

1. Move the cursor into position:

   □ Type ***/The*** and press (RETURN) to move the cursor to `The`.

   □ The display should now look like this:

```
Beijing (Peking).  The Mayor of West Berlin
stood yesterday on top of the Great Wall
```

```
of China next to Deng Xiaoping, leader
of the nation of one billion people.
```

2. Delete the sentence at its current location:

   □ Type **"ad)** to delete the entire second sentence to buffer a.

   □ The paragraph should look like this:

```
Beijing (Peking).  _

Begun during the Ch'in dynasty (about
```

3. Call up **letter** and move the cursor to the new location:

   □ Type **:n! letter** and press (RETURN) to save **wall** automatically and bring in **letter** for editing.

   □ When the text appears on the screen, type **G** to move the cursor to the end of the file.

   □ The display should look like this:

```
Robert G. Sanders III

_
```

4. Put the sentence at the new location from buffer a:

   □ Type **"ap** to take the sentence from buffer a and put it after the end of the letter.

   □ The display should now look like this:

```
Robert G. Sanders III
The Mayor of West Berlin
stood yesterday on top of the Great Wall
of China next to Deng Xiaoping, leader
of the nation of one billion people.
```

[No blank line here]

## Moving a paragraph

**"xd**} and **"xP**
**"xd**} and **"xp**

Moving a paragraph to another file is like moving a sentence. This time we use **d**} to delete a paragraph, not **d)**. We plan to move the cursor to the start of a paragraph, delete the paragraph *to a buffer*, then move to the new location and put the paragraph there *from the same buffer*. As an example, let's move a paragraph from **letter** to **wall**.

1. Move the cursor into position:

□ Type ***?I'm*** to move the cursor to the beginning of the first paragraph of **letter**.

□ Type **k** to move the cursor up to the previous blank line.

□ The display should now look like this:

```

I'm sorry you fainted during last
Friday's interview.  You see, I was
just raising my arms to give you the
victory sign.
```

2. Delete the paragraph from here and place it in buffer b.

□ Type **"bd**} to delete the entire second paragraph to buffer b.

□ The display should look like this:

```
Dear Mr. Benson:

I had to rush straight from
the tennis courts.  There wasn't
```

3. Begin editing **wall** and move the cursor to the new location:

□ Type **:e! wall** to abandon the text from **letter** and bring in **wall**.

□ When the text appears, type ***/Some*** to advance the cursor to the beginning of the third paragraph.

□ Type **k** to move the cursor up to the previous blank line.

□ The display should look like this:

```
the Ming dynasty (about the time the
Mayflower arrived at Plymouth Rock).

Some 25 feet high, 15 to 30 feet wide at
the base, and 12 feet wide at the top,
```

4. Put the paragraph at the new location from buffer b:

□ Type **"bP** to put the paragraph *above* the current line.

□ The display should now look like this:

```
the Ming dynasty (about the time the
Mayflower arrived at Plymouth Rock).


I'm sorry you fainted during last
Friday's interview.  You see, I was
just raising my arms to give you the
```

```
victory sign.

Some 25 feet high, 15 to 30 feet wide at
the base, and 12 feet wide at the top,
```

### Abandoning the text

Before going on to *copying* text to another file, let's abandon the text in our work area. Then we can start all over again with **letter** and **wall**. Type **:q!** to abandon the work area, then type

```
$ vi wall    RETURN
```

to begin a new session with the original text.

## 11.3 Copying text to another file

*Copying* text from one file to another in **vi** is similar to *moving* text. The only difference is that you use the "yank" command (**y**) instead of the delete command (**d**).

### Copying a sentence

**"xy)** and **"xP**
**"xy)** and **"xp**

*Copying* a sentence is like *moving* a sentence. We plan to move the cursor to a sentence, "yank" the sentence *to a buffer*, then move to the new location and put the sentence there *from the same buffer*. As an example, let's copy a sentence from **wall** to **letter**.

1. Move the cursor into position:

   □ Move the cursor to the T in The Mayor:

   ***/The May*** RETURN

   □ The display should now look like this:

   ```
   Beijing (Peking).  The Mayor of West Berlin
   stood yesterday on top of the Great Wall
   of China next to Deng Xiaoping, leader
   of the nation of one billion people.
   ```

2. Yank the sentence at its current location into buffer c.

   □ Type **"cy)** to "yank" the entire second sentence.

□ The paragraph should look the same:

```
Beijing (Peking).  The Mayor of West Berlin
stood yesterday on top of the Great Wall
of China next to Deng Xiaoping, leader
of the nation of one billion people.
```

3. Begin editing **letter** and move the cursor to the new location:

□ Type **:e! letter** and press (RETURN) to abandon **wall** and call up **letter**.

□ When the display appears, type ***/Sincere*** and press (RETURN) to move the cursor to the line that says, `Sincerely,`.

□ The display should look like this:

```
I hope you will give very careful
consideration to my qualifications.

Sincerely,
```

4. Put the sentence at the new location from buffer `c`.

□ Type **"cP** to put the sentence above the current line.

□ The display should now look like this:

```
I hope you will give very careful
consideration to my qualifications.

The Mayor of West Berlin
stood yesterday on top of the Great Wall
of China next to Deng Xiaoping, leader
of the nation of one billion people.Sincerely
```

## Copying a paragraph — **"xy**} and **"xP** / **"xy**} and **"xp**

Copying a paragraph is like moving a sentence. We plan to move the cursor to the start of a paragraph, "yank" the paragraph to a buffer, then move to the new location and "put" the paragraph there from the same buffer. As an example, let's copy a paragraph from **letter** to **wall**.

1. Move the cursor into position:

□ Type **2**{ to move the cursor to the beginning of the previous paragraph.

□ The display should now look like this:

```
▬
I hope you will give very careful
consideration to my qualifications.

The Mayor of West Berlin
stood yesterday on top of the Great Wall
of China next to Deng Xiaoping, leader
of the nation of one billion people.  Sincerely
```

2. Yank the paragraph at its current location to buffer d:

□ Type **"dy}** to yank the entire second paragraph to buffer d.

□ The display should look like this (the same):

```
▬
I hope you will give very careful
consideration to my qualifications.

The Mayor of West Berlin
stood yesterday on top of the Great Wall
of China next to Deng Xiaoping, leader
of the nation of one billion people.  Sincerely
```

3. Call up wall and move the cursor to the new location:

□ Type **:e! wall** to abandon the text from letter and bring in wall.

□ Type ***/The p*** to advance the cursor to the beginning of the last paragraph, then type **k** to move the cursor up to the blank line.

□ The display should look like this:

```
long, greater than the distance between
New York and Dallas.
▬
The purpose of the wall was to protect
China against invaders from the north.
```

4. Put the paragraph at the new location from buffer d:

□ Type **"dP** to put the paragraph above the current line from buffer d.

□ The display should now look like this:

```
long, greater than the distance between
New York and Dallas.

▬
I hope you will give very careful
consideration to my qualifications.

The purpose of the wall was to protect
China against invaders from the north.
```

### Abandoning the text

Before leaving this discussion of copying text, let's abandon the text in our work area. Then we can start all over again with the original version of **wall** in the next chapter. Type **:q!** to abandon the work area.

## 11.4 Invoking **vi**

So far you have always invoked **vi** with the simplest form of the command line, which places the cursor at the beginning of the file with no options:

```
$ vi letter
```

However, there are a number of options available when you begin an editing session.

### Specifying a starting line *+line*

To begin with, you can specify the line number on which you wish to begin editing. Simply type a plus sign followed by the desired line number after **vi**. To see how this works (assuming you are in directory **text**), try calling up **metric.c** with the cursor positioned at the beginning of the main routine of the program (line 14). Here is the command:

```
$ vi +14 metric.c
```

The opening display should look like this:

```
        gallons = low;
        while (gallons <= high)
        {
            liters = (gallons * 3.785);
            printf("%4.0f \t %6.2f\n", gallons, liters);
            gallons += step;
        }
}
~
~
~
"metric.c" 18 lines, 331 characters
```

A simple variation of this is to use a search pattern in place of an actual number. To see how this works, type **:q** and press (RETURN) to leave **vi**, then re-enter the file at the first line that contains `float`:

```
$ vi +/float/ metric.c    [You can also type +/float here]
```

The opening display should look like this:

```
/* Convert gallons to liters   */
main()
{
     int low, high, incr;
     float gallons, liters;

     low = 10; high = 20; step = 2;

     printf(%4s \t %6s \n\n; "gals", "liters");
                         .    .    .
~
~
~
"metric.c" 18 lines, 331 characters
```

Another variation is to use a plus sign by itself to position the cursor on the last line of the file. (This would be handy if you wanted to add new text to the end.) To see how this works, type **:q** and press RETURN to leave **vi**, then re-enter the file on the last line:

```
$ vi + metric.c
```

The opening display should look like this:

```
 . . .
     gallons = low;
     while (gallons <= high)
     {
          liters = (gallons * 3.785);
          printf("%4.0f \t %6.2f\n", gallons, liters);
          gallons += step;
     }
}
~
~
~
"metric.c" 18 lines, 331 characters
```

Now you can type **:q** and press RETURN again to leave **vi** and return to the UNIX shell prompt.

## Selecting an option *-option*

While a plus sign introduces a starting line number, a minus sign indicates one of four possible options for invoking **vi**. Each option is selected by typing a single letter after the minus sign:

-r *Recover the file*—Use this option after a system or program crash (UNIX or **vi**) to retrieve the most recently-saved version of the file. If you omit the filename, the names of the files that can be recovered will be printed. (Note that a system or program crash is usually not catastrophic. You just have to get things restarted.)

-R *Read-only*—Use this option to edit a file in read-only mode, meaning that the file can be viewed, but not modified.

-t *Locate Tag*—Use this option to call up the file that contains the tag named and begin editing at the location of the tag's definition. (This option will be explained in the section that follows.)

-x *Decrypted Read*—Use this option to view the cleartext of an encrypted file. (The file itself is not actually decrypted; see also "Encrypting Information" in Chapter 5.)

Here are a few examples to illustrate the use of these options:

| | |
|---|---|
| $ **vi -r letter** | Recover **letter** after a crash. |
| $ **vi -R metric.c** | Edit **metric.c** as if were a read-only file. |
| $ **vi -t tag** | Begin editing the file that contains the tag named, and position the cursor at its location. |
| $ **vi -x remark.crypt** | Edit an encrypted file named **remark.crypt**. |

## Using tags for text files

You just learned how to have **vi** open a file and move the cursor immediately to a specified line (see the section, "Specifying a Starting Line," above). A *tag* is an extension of this same concept. **vi** allows you to set aside a file that contains a list of starting points (or tags) for a text file that you are working with. The name of the file that contains the tags must be **tags**; no other name can be used.

For example, suppose you are using **wall** so extensively that you would like to be able to retrieve the file very quickly. And suppose you would like to be able to mark the beginning of each paragraph, so that you could begin editing at any one of those lines right away. Here is how you could do this:

1. Set up your tags in a separate *tags file*:

   □ Create a new file called **tags**:

   ```
   $ vi tags
   ```

   □ Press **a** to request text entry mode and enter the following four lines, with columns set at tab stops:

```
par.1     wall     3
par.2     wall     /Ch'in/
par.3     wall     14
par.4     wall     /purpose/
ESC
```

□ Now store the text and return to the UNIX shell prompt with **:wq**:

```
:wq
"tags" [New file] 5 lines, 68 characters
$ _
```

Each line in a tag file contains three things, separated by tabs: Name of
the tag    Name of the file    Location of the tag

The location of the tag is indicated by either a line number or a search command, as described above under "Specifying a Starting Line." In a tag file, however, no plus sign is used in front of the information. Two of the entries above contain line numbers and two contain search commands. For the third entry, **par.3** (paragraph 3) is the *name* of the tag, **wall** is the name of the *file* that contains the tag, and `14` indicates the *location* of the tag (line 14, the line number of the beginning of the third paragraph).

2. Use this new tag file to start a new editing session:

□ Use the **-t** option in a **vi** command to start editing right at the beginning of the third paragraph (tag `p3`):

```
$ vi -t par.3    [You don't even need the name of the file]
```

□ File **wall** will be displayed, with the cursor at the first line of the third paragraph:

```
the Ming dynasty (about the time the
Mayflower arrived at Plymouth Rock).

Some 25 feet high, 15 to 30 feet wide at
the base, and 12 feet wide at the top,
```

You aren't restricted to using tags when *beginning* an editing session; you can also use tags in the middle of editing another file. Just use the **:tag** command from *vi command mode*, indicating the name of the tag. For example, suppose you are in the middle of a session with **letter**. If you have stored the text since your most recent changes, you can type a command like this in *vi command mode*:

```
:tag par.3
```

One final note: In our example above, we used only tags from a single file. However, your **tags** file can contain entries for many different files, provided that you have them in alphabetical order. (The **sort** command will take care of this for you.) Since line numbers can change with revisions, it is best to use a search command to indicate the location of a tag.

### Using tags in programs

For programs written in C, FORTRAN, or Pascal, you can use the **ctags** program to create a tag file automatically. The **ctags** program will insert one entry for each function in your program. For example, suppose you wanted a tag file for a program called **list.c**, which contained three functions: **main()**; **split(x)**; and **unite(y)**.

If you execute the command line

```
$ ctags list.c
$ _
```

the **ctags** program will create a three-line tags file called **tags**, which you can then look at with **cat**:

```
$ cat tags
Mlist     list.c    ?^main()$?
split     list.c    ?^split(x)$?
unite     list.c    ?^unite(y)$?
$ _
```

The first tag name used by **ctags** will always be the name of the source file (with **M** added in front of the name and **.c** deleted from the end). To avoid tagging a comment line that names one of the functions, **ctags** selects only those function names that appear on separate lines by themselves.

**Caution:** The **ctags** program will wipe out your current **tags** file, overwriting it with its own entries.

Since the **ctags** program tags only functions and since it doesn't spare your current tags, it has serious flaws. Use it only if you can tolerate its quirks and limitations.

## 11.5 Summary

In this chapter you learned how to begin editing a different file without leaving **vi**. You also learned two different methods for moving (or copying) text from one file to another. Finally, you learned about the options available when you begin a new editing session, and tried using tags.

## Editing another file

To save the text from your current file before calling up another, use **:w** to write to the old file, **:e** to read the new one. To abandon the text from your current file and begin editing another immediately, use **:e!**. To have your current file saved automatically before you call up another, first make sure you have the *auto-write* option set. Then use **:n**.

## Moving or copying text to another file

In general, the procedure for moving (or copying) text from one file to another is to delete (or "yank") the text from its current location into a buffer, then "put" the text from the same buffer into the new location.

| Text | Delete to Buffer *x* | | Yank to Buffer *x* | |
|---|---|---|---|---|
| Sentences(s) | "*x*d( | "*x*d3( | "*x*y( | "*x*y3( |
| | "*x*d) | "*x*d3) | "*x*y) | "*x*y3) |
| Paragraph(s) | "*x*d{ | "*x*d3{ | "*x*y{ | "*x*y3{ |
| | "*x*d} | "*x*d3} | "*x*y} | "*x*y3} |

| | Put from Buffer *x* |
|---|---|
| In front of the cursor or above the current line | "*x*P |
| After the cursor or below the current line | "*x*p |

## Options for beginning a new editing session

To specify a starting line when you begin a session with **vi**, type **vi**, a space, a plus sign, a line number, another space, and the name of the file. The line number may be any of the following: a number, a search string, a blank space.

To select one of four special options when you begin a session with **vi**, type **vi**, a space, a minus sign, a letter, another space, and the name of the file. The options are **-r** (recover the file), **-R** (read-only), **-t** (locate the tag), and **-x** (decrypted read).

You can set up a **tag file** to list starting-points for an editing session with **vi**, then begin editing at one of these points by using either the **-t** option in a **vi** command line or the **tag** command during an editing session in progress.

# 12

# Customizing **vi**

In the previous chapters you learned how to use **vi** in its standard operating modes. In this chapter you will learn how to modify the options of **vi** to change its many user-definable features. You will also learn how to use abbreviations and define keys.

## Setting options **set**

In Chapter 9 you used the ***:set*** command to set two of **vi**'s options, line-numbering and shift-width:

| | |
|---|---|
| ***:set nu*** | Set line-numbering. |
| ***:set sw=5*** | Set shift-width to 5 columns. |

You may already have noticed a few things about these options. First of all, most have abbreviations, such as `nu` (for `number`) and `sw` (for `shift-width`). The majority of options are set as toggles, which means that the feature is either on (`nu`) or off (`nonu`). The other options require specific assigned values, which may be either numbers or strings of characters.

In this chapter we'll discuss only a few of the most commonly-used options. For a complete list of all the options for **vi**, see Appendix D, "Summary of **vi** Options," where they are described in detail by type: toggled, number-valued, and string-valued.

## Making temporary changes

To set an option for the duration of an editing session, all you have to do is to type a ***:set*** command in **vi** command mode and press (RETURN). The option you set will be in effect only until you exit **vi** and return to the UNIX shell. This is the best way to try out the different settings. Here are three examples of commands:

***:set aw*** Set `auto-write` (force automatic write before calling up a new file or executing a UNIX shell command).

***:set report=3*** Set `report` to 3 (request a message after any operation involving at least three repetitions, such as three lines deleted, three lines appended, and so on—but no message for fewer than three).

***:set wm=6*** Set your wrap margin to 6.

## Changing the Width of Each Shift

To see what happens when you change the setting of an option, let's use a file to do some experimenting.

1. Begin a new session with a new file:

   □ At the UNIX shell prompt, type ***vi custom*** and press RETURN to begin a new session with file **custom**.

   □ Type ***a*** and enter the following lines, with a RETURN at the end of each line (and ESC at the end to return to **vi** command mode):

   ```
   Settings for vi

   Here are a few lines of text
   to demonstrate how we can
   change the way the editor
   works. ESC
   ```

2. Try shifting text with the current setting:

   □ Type ***?Here*** to move the cursor up to the `H` in `Here`, then type ***>)*** to shift the sentence to the right:

   ```
   Settings for vi

           Here are a few lines of text
           to demonstrate how we can
           change the way the editor
           works.
   ```

   □ As you can see, each shift moves the text eight columns to the right.

3. Change the `shift-width` option:

   □ Type ***:set_sw?*** (with a question mark) and press RETURN to check the current value of `shift-width`:

   ```
   :set sw?
   ```

   □ You will see the response on the status line at the bottom of the screen, telling you that each shift will currently move eight columns:

```
shiftwidth=8
```

□ Type ***:set_sw=5*** to change the value to 5:

```
:set sw=5
```

□ Type ***<(*** to shift the sentence back to the left.

```
Settings for vi

   Here are a few lines of text
   to demonstrate how we can
   change the way the editor
   works.
```

As you can see, each shift moves the text five columns now, instead of eight. Try a few more shifts, then restore the text to its original appearance.

## Wrapping words near the margin

Another option changes **vi**'s handling of words that are entered as the cursor approaches the right-hand margin of the screen. With the standard default setting, if you type a word into the margin, it just keeps going right past the edge of the screen. (Your terminal, not **vi**, may wrap the words to the beginning of the next line.) With the `wrap-margin` option, however, you can have **vi** wrap a word to the next line—a common feature of word processing systems. Just set it to any positive number.

1. Try entering some text with the standard default setting:

   □ Move the cursor to the end of the last line (***L***, then ***$***).

   □ Type ***a*** and press RETURN twice to append text, then type the following, *without* pressing RETURN at the end of the line:

   We're going to type a very long line of text to see what happens close to the side of the screen.

   □ Type ESC at the end to return to **vi** command mode.

The word `side` should be split at the edge of the screen (assuming you have an 80-column screen), like this: `s/ide`. But this is the work of your terminal, not of **vi**.

2. Change the `wrap-margin` option:

   □ Type ***:set wm?*** (with a question mark) and press RETURN to check the current setting of `wrap-margin`:

***:set wm?*** RETURN

□ You will see the response at the bottom of the screen, telling you that there is no wrapping of words:

```
wrapmargin=0
```

□ Type ***:set_wm=10*** and press RETURN to change the value to `10`:

***:set wm=10*** RETURN

3. Now enter the same text to try out the new setting:

□ With the cursor still at the end of the last line, type **a**, press RETURN, then type the following—again *without* pressing RETURN at the end of the line:

This time we're going to type a very long line of text to see what happens close to the side of the screen.

□ Type ESC at the end to return to **vi** command mode.

Because you had `wrap-margin` set to 10, **vi** wraps the word `happens` to the next line (assuming you have an 80-column screen). The `10` tells **vi** to wrap any word that comes within ten columns of the right-hand side of the screen.

## Making other changes

You may want to change a few more options to suit the kind of work you plan to be doing. Here are a few of the most commonly used:

***:set ai*** Set `auto-indent` (RETURN moves the cursor to the next line, but at the same indentation as for the previous line).

***:set list*** Set `list` (shows tab stop and new line symbols on the screen, as is customary with the **ed** list command).

***:set nu*** Set `number` (displays line numbers).

To see a list of the current values for all **vi** options, use this command:

***:set all***

to produce a display like this (see Appendix D, "Summary of **vi** Options," for details):

```
noautoindent                  remap
autoprint                     report=5
noautowrite                   scroll=11
nobeautify                    sections=NHSHH HUnhsh
directory=/tmp                shell=/bin/sh
noedcompatible                shiftwidth=5
noerrorbells                  noshowmatch
hardtabs=8                    nslowopen
noignorecase                  tabstop=8
nolisp                        taglength=0
nolist                        tags=tags /usr/lib/tags
magic                         term=adm3
mesg                          noterse
nonumber                      timeout
open                          ttytype=adm3
nooptimize                    warn
paragraphs=IPLPPPQPP LIpplpipbp  window=23
prompt                        wrapscan
noreadonly                    wrapmargin=10
redraw                        nowriteany
[Hit return to continue]_
```

## Making changes permanent

Once you've decided which options you would like to have in effect every time you use **vi**, you can store the commands that set them in a special file. This file, called .exrc (**ex** read command), is read at the beginning of every **vi** session. For example, suppose you would like to have automatic indentation, automatic writing, and word wrapping six columns from the side of the screen. End this session with **vi**, go to your home directory, create a file named **.exrc**, and enter the following lines (without colons):

```
set autoindent              set ai
set autowrite         or    set aw
set wrapmargin=6            set wm=6
```

or      set ai aw wm=6

Save the text (***:wq***) and return to text (***cd text***). The next time you begin a new session with **vi**, these options will take effect automatically. You won't have to type any ***:set*** commands.

## Assigning abbreviations **ab**

If you ever have to type a long name or other string repeatedly, it is convenient to have a way to enter a shortened form of it quickly. The **vi** editor provides a command that allows you to assign a long string to a shorter abbreviation. Then, once you've made the assignment, any time you type the abbreviation, **vi** will replace it with the longer text.

## Making temporary assignments

To have an abbreviation assigned for the duration of the current session, type the command ***:ab***, a space, the abbreviation, another space, the full, unabbreviated string, and then press (RETURN)For example, using file **custom** again, here's the procedure:

1. Call up **vi** (if necessary) and move to the end of the work area:

   □ If **vi** is not active, type ***vi + custom*** to begin a new session with **custom** with the cursor at the end of the file.

   □ If **vi** is active and the text of **custom** is already on the screen, type ***L*** to move the cursor to the end of the text.

2. Assign three abbreviations:

   □ Type ***:ab xen XENIX operating system*** and press (RETURN) to assign `XENIX operating system` to `xen`.

   □ Type ***:ab unx UNIX operating system*** and press (RETURN) to assign `UNIX operating system` to `unx`.

   □ Type ***:ab btl Bell Telephone Laboratories*** and press (RETURN) to assign `Bell Telephone Laboratories` to `btl`.

3. Type a sentence using the three abbreviations:

   □ Type ***a*** and enter the following sentence, followed by (ESC):

```
The xen was derived from the
 unx, which originated at
 btl in 1969. (ESC)
```

   □ This is how the text should now look on the screen:

```
The  XENIX operating system was derived from the
 UNIX operating system, which originated at
 Bell Telephone Laboratories in 1969.
```

This brief example should convince you of the value of this feature (often referred to in the word-processing industry as a *glossary* feature).

## Making assignments permanent

Any abbreviations that you plan to use frequently can be added to your .exrc file, along with your **set** commands. For example, to have the three abbreviations you just tried available to you every time you use **vi**, include them in **.exrc**, like this:

```
set autoindent
set autowrite
set wrapmargin=6
ab xen XENIX operating system
ab unx UNIX operating system
ab btl Bell Telephone Laboratories
```

In these examples, we've deliberately used three-character abbreviations. You can use shorter abbreviations if you want to, but you run the risk of typing your abbreviation as part of another word, thereby substituting in the longer text by accident. For example, suppose we had selected `un` as our second abbreviation. Then, if we attempted to type ***run***, we would get `rUNIX operating system` instead.

## DEFINING KEYS **map**

There's another kind of abbreviation you can perform with **vi**. It involves assigning a sequence of commands (or keystrokes) to a key on your keyboard (sometimes referred to as *mapping*). Once you've made such an assignment, then you can press that key to invoke the commands. This is another time-saving device that can mean pressing one key instead of a dozen. However, there is one slight catch to it: you have to select a key that you don't plan to use for any other purpose. Otherwise, you could trigger your command sequence unexpectedly. Your best bets are either the function keys or one of the following: `K V g q v k ; _ =` (which are not used by **vi**).

## MAKING A TEMPORARY ASSIGNMENT

To set up a mapping for the duration of an editing session, all you have to do is type a ***:map*** command in **vi** command mode and press (RETURN). The mapping will remain in effect only until you end the editing session.

The first thing you have to do is to find a key that doesn't have another function already assigned to it. To do this, just press a key in *vi command mode.* If your terminal beeps, then the key is unassigned. For example, try the asterisk (*). There's a beep, so it's available.

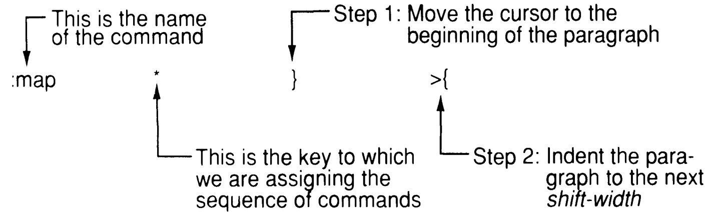
Now that we have an available key, we need a useful command sequence to assign to it. Suppose we want to be able to indent an entire paragraph to the next tab stop with one keystroke (*), assuming that we begin with the cursor somewhere in that paragraph. Let's go through the commands we'd have to enter to accomplish this manually:

1. Move the cursor to the beginning of the paragraph {

2. Indent the entire paragraph to the next tab stop >}

With an available key and a sequence of commands to assign to it, we're ready to construct our command (with the characters spread out for easier viewing) as shown in Figure 12.1: Here's how you actually type it (with spaces around * only):

FIGURE 12.1. A simple mapping command.


This is the name of the command
Step 1: Move the cursor to the beginning of the paragraph
:map
*
}
>{
This is the key to which we are assigning the sequence of commands
Step 2: Indent the paragraph to the next *shift-width*

***:map **** {**>**} (RETURN)

Now, with the key assigned, you are ready to try it out. So let's bring in a file you've worked with before.

1. Store the text of **custom** and start editing with **wall**:

   □ Type ***:w*** to store the text in **custom**.

   □ Type ***:e +/China/ wall*** and press (RETURN) to call up **wall** with the cursor in the middle of the first paragraph.

   □ The display should now look like this:

   Beijing (Peking). The Mayor of West Berlin stood yesterday on top of the Great Wall o̲f China next to Deng Xiao-ping, leader of the nation of one billion people.

2. Try out your new key:

   □ Type ******* to invoke the sequence of commands.

   □ You should see the cursor move to the `B` in `Beijing` and the entire paragraph shift to the right.

## MAKING ANOTHER TEMPORARY ASSIGNMENT

That was a simple sequence of commands—just three keystrokes and two commands. Let's try another example that's a little more involved. This time, we'll pick a key (=), and assign to it the sequence that will insert the following two words in front of the current sentence:

***Note this:***

This is how you would accomplish this manually:

1. Move the cursor to the beginning of the sentence **(**

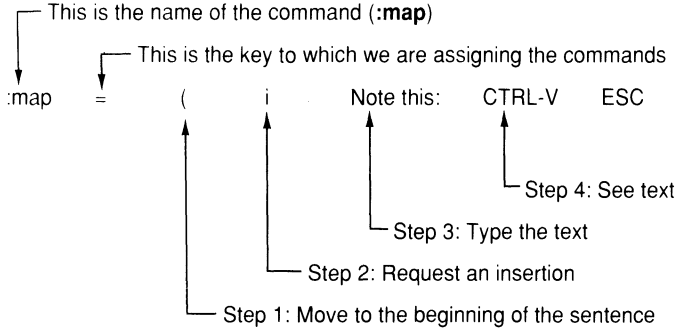
2. Request an insertion **i**

3. Type the text to be inserted ***Note this:***

4. Terminate insertion of text ESC

This is similar to the previous example, but it involves more keystrokes and one additional problem: How do we let **vi** know that we've typed an ESC key at the end of the text? (If we just press the ESC key, **vi** will read that as End of text entry.) **vi** provides a command to take care of this, as you'll see in a moment. Now that we have a key available (=) and a sequence of commands, we're ready to construct a command (with the characters spread out again for clarity):

FIGURE 12.2. A more involved mapping command.


This is the name of the command (:**map**)
This is the key to which we are assigning the commands
:map
=
(
i
Note this:
CTRL-V
ESC
Step 4: See text
Step 3: Type the text
Step 2: Request an insertion
Step 1: Move to the beginning of the sentence

Here's how you actually type it (without spaces):

```
:map = (iNote this: ^[    RETURN
```

To enter ESC as a text character, first type CTRL-V, then ESC. When you do this, you will see ^[ appear on the screen. CTRL-V is what you type in **vi** any time you want to enter a control character *as a text character.* For example, to enter RETURN as a character (as in terminating an **ex** command), type CTRL-V, then press the RETURN key. On the screen you will see ^M. (If you'd like to know why ^[ represents ESC and ^M represents RETURN, see "The ASCII Table," page 709, in Appendix N.)

With the = key assigned to a command sequence, you are now ready to try the key out.

1. Find a suitable location to start from:

   □ Move the cursor near the start of the last paragraph (***/China***).

   □ The display should look like this:

```
The purpose of the wall was to protect
China against invaders from the north.
The mayor told his host, ''This wall was
here to keep people out.  We have a
wall that is there to keep people in."
```

2. Try out your new key:

   □ Type **=** to invoke the sequence of commands.

   □ You should see the cursor move to the `T` in `The purpose` and the text inserted:

```
Note this:_The purpose of the wall was to protect
China against invaders from the north.
The mayor told his host, ''This wall was
here to keep people out.  We have a
wall that is there to keep people in."
```

## Making still another temporary assignment

Now you've assigned two sequences, one very simple and one a little more involved. Let's try one more sequence just for practice. This time, we'll pick a key (+), and assign to it the sequence that will append the following closing to the end of a letter:

[Leave one blank line]

TAB `Sincerely,`

[Leave three blank lines]

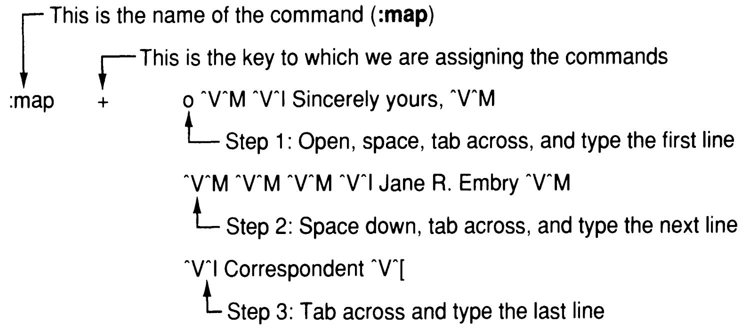
TAB `Jane R. Embry`<br>
TAB `Correspondent`

The following shows how you would accomplish this manually:

| | | |
|---|---|---|
| 1. | Request an opening below | **o** |
| 2. | Leave one blank line | RETURN |
| 3. | Move to the tab stop | TAB |
| 4. | Type the text | ***Sincerely yours,*** |
| 5. | Leave three blank lines | RETURN RETURN<br>RETURN RETURN |
| 6. | Move to the tab stop | TAB |
| 7. | Type the text | ***Jane R. Embry*** |
| 8. | Move to the next line | RETURN |
| 9. | Move to the tab stop | TAB |
| 10. | Type the text | ***Correspondent*** |
| 11. | Terminate insertion of text | ESC |

This sequence requires nearly fifty keystrokes, but we're going to reduce all that typing to just one keystroke. To accomplish this, recall from Chapter 5, "Communication in UNIX," that (TAB) is (CTRL-I) **(^I)** and (RETURN) is (CTRL-M) **(^M)**. Once again, we have a key available (+) and a sequence of keystrokes, so we're ready to contstruct a one-keystroke command (shown with the characters spread out for clarity in Figure 12.3).

FIGURE 12.3. A much more involved mapping command.



This is the name of the command (**:map**)
This is the key to which we are assigning the commands
:map
+
o ^V^M ^V^I Sincerely yours, ^V^M
Step 1: Open, space, tab across, and type the first line
^V^M ^V^M ^V^M ^V^I Jane R. Embry ^V^M
Step 2: Space down, tab across, and type the next line
^V^I Correspondent ^V^[
Step 3: Tab across and type the last line

Here's how you actually type it (on one line without spaces):

***:map + o^V^M^V^ISincerely yours, ^V^M^V^M^V^M^V^M***
***^V^IJane R. Embry^V^M^V^ICorrespondent^V^[*** (RETURN)

and this is how it will look on the screen:

```
:map + o^M    Sincerely yours,^M^M^M^M
   Jane R. Embry^M^ICorrespondent^[
```

With the + key assigned to a command sequence, you are now ready to try the key out.

1. Find a suitable location to start from:

   □ Move the cursor to the last line of main text (***/wall that***).

   □ The display should look like this:

   ```
   The purpose of the wall was to protect
   China against invaders from the north.
   The mayor told his host, ``This wall was
   here to keep people out.  We have a
   wall that is there to keep people in."
   ```

2. Try out your new key:

   □ Type ***+*** to invoke the sequence of commands.

□ You should see the cursor move below the text and insert the closing:

```
The purpose of the wall was to protect
China against invaders from the north.
The mayor told his host, ``This wall was
here to keep people out.  We have a
wall that is there to keep people in."

        Sincerely,



        Jane R. Embry
        Correspondent_
```

### Making assignments permanent

Once again, any abbreviations you plan to use all the time can be added to your **.exrc** file, along with your `set` and `ab` commands. For example, to have the two key assignments you just tried available to you every time you use **vi**, just include them in **.exrc** (without colons, of course):

```
set autoindent
set autowrite
set wrapmargin=6
ab xen XENIX operating system
ab unx UNIX operating system
ab btl Bell Telephone Laboratories
map * {>}
map = (iNote this: ^[
map + o^M^ISincerely yours,^M^M^M^M\
^IJane R. Embry^M^ICorrespondent^[
```

Now the *, =, and + keys will have these command sequences assigned to them every time you begin a new session with **vi**.

## 12.1 Summary

In this chapter you learned how to change **vi**'s options, how to abbreviate a long segment of text, and how to assign a sequence of commands to a key on the keyboard.

### Making temporary changes

To set a **vi** option for the duration of the current session, use the `set` command, as shown here:

```
:set report=3
```

To assign a longer segment of text to an abbreviation for the duration of the current session, use the `ab` command, as shown here:

```
:ab unx UNIX operating system
```

To assign a sequence of commands to a key on the keyboard for the duration of the current session, use the `map` command, as shown here:

```
:map = (iNote this: ^]
```

## Making permanent changes

To activate any of these modifications in **vi** each time you begin a new session, place these commands in the file **.exrc** (without colons), as illustrated here:

```
set report=3
ab unx UNIX operating system
map @ (iNote this: ^[
```

## For further reading

If you'd like to read more about the visual editor, refer to

Hansen, August, *vi: the UNIX Screen Editor*, New York: Brady, 1986.

Sonnenschein, Dan, *A Guide to vi: Visual Editing on the UNIX System*, Engelwood Cliffs, NJ: Prentice-Hall, 1987.

# Part III

# Text Processing

In Part II, you learned how to enter and edit text with **vi** and **ex**. In Part III, you will learn how to process text with a variety of UNIX tools. You will begin by learning how to search for text with the **grep** command, then how to sort lines of text with the **sort** command. Then you will learn how to write simple programs to perform custom changes on text with the **awk** command and the **c** language.

# 13

# Searching and Sorting

Now that you have learned how to create and edit text files, we can consider in detail processing text already stored in a file. In this chapter we'll discuss the UNIX programs that allow you to search for text in a file and sort lines of text. In Chapters 14, "Programming with **awk**," and 15, "Programming with C," we'll discuss the languages that allow you to program text output.

## Searching for text **grep**

You are already familiar with the search facilities that are built into the UNIX text editors **vi** and **ex**. These same search facilities have also been incorporated into another UNIX command called **grep** (globally find and replace regular expressions and print), to which you were briefly introduced in Part I. The acronym doesn't quite match, but the enhanced search features of **grep** will help you match text with ease.

To get some practice in searching with **grep**, let's create three new files called dawson.farms, jenkins.farms, and parker.farms. These names represent three suppliers for a maker of juices, jams, jellies, and canned fruit called "Fruit-of-the-Tree." Each file contains a short list of the fruits supplied and the prices. Let's move to subdirectory text and create these files, using the TAB key to position the prices, quality codes, and identifiers. In the examples that follow, we are using **vi** to enter the text. In each instance, type ***a*** to enter the text, press ESC to stop, type ***:w*** to write the text to a file, then ***:q*** to return to the shell prompt.

```
$ vi dawson.farms
Bing cherries            0.51    5   D
Concord grapes           0.48    4   D
D'Anjou pears            0.13    3   D
Comice pears             0.15    4   D ESC
```

[Press TAB twice after "cherries"]

```
$ vi jenkins.farms
Concord grapes            0.51    5   J
                [Leave a blank line]
Bartlett pears            0.12    3   J
                [Leave a blank line]
Navel oranges             0.26    4   J ESC
```

```
$ vi parker.farms
Bing cherries           0.48    4  P
Thompson grapes         0.35    5  P
Valencia oranges        0.29    5  P ESC
```

## Conducting simple searches

With three new files in **text** containing information about various fruits, we can now begin making some simple searches with **grep**. In each instance, **grep** will output every line that contains the string we are looking for.

1. Does Parker supply grapes?

   □ Check file **parker.farms** for `grapes`:

   ```
   $ grep grapes parker.farms
   Thompson grapes          0.35    5   P
   $ _
   ```

   □ Answer: Yes, Parker supplies Thompson grapes.

2. Does Dawson supply oranges?

   □ Check file **dawson.farms** for `oranges`:

   ```
   $ grep oranges dawson.farms
   $ _
   ```

   □ Answer: No. (There was no output.)

3. Does Jenkins supply Bartlett pears?

   □ Check file **jenkins.farms** for `Bartlett pears`:

   ```
   $ grep "Bartlett pears" jenkins.farms
   Bartlett pears          0.12    3    J
   $ _
   ```

   □ Answer: Yes, Jenkins supplies Bartlett pears.
   Because we were searching for two words separated by a space, quotation marks were required for this search.

4. Who supplies Bing cherries?

   □ Check all three files for `Bing cherries`:

   ```
   $ grep "Bing cherries" *farms
   dawson.farms: Bing cherries       0.51    5    D
   parker.farms: Bing cherries       0.48    4    P
   $ _
   ```

□ Answer: Dawson and Parker.

We used the wild-card symbol (*) here to abbreviate **dawson.farms jenkins.farms parker.farms**. This example illustrates one of the enhanced features of **grep**: its ability to search several different files for a given string. As the output shows, **grep** supplies the name of each file that contains the string.

## Using aids in searching

Like other UNIX programs that provide searching, **grep** allows you to use the special characters, or *metacharacters*, that make it easier to match patterns. The expressions formed from these characters are called *regular expressions* (the **re** in **grep**). Here is a quick review of these characters:

| | | | |
|---|---|---|---|
| ^ | Beginning of line | $ | End of line |
| . | Any single character | * | Repeat any number of times |
| [ ] | Any enclosed character | \ | Turn off special meaning |

Here are some examples of regular expressions formed from these characters, with a few strings they would match if used in a **grep** command line:

| **Regular Expression** | **Some of the Strings Matched** |
|---|---|
| "^3." | `30 3a 3+ 3)` (at beginning of a line) |
| "t..$" | `too t36 t\|T t+5` (at end of a line) |
| "[Ff]arm" | `Farm farm` |
| "\$[0-9]\.[0-9][0-9]" | `$4.67 $8.32 $1.99 $3.71` |
| "[abc]\[i\]" | `a[i] b[i] c[i]` |
| "^$" | (A blank line) |

Here are a couple of simple searches with regular expressions:

1. Does Dawson supply D'Anjou pears?

   □ Check file **dawson.farms** for **D'Anjou pears**:

   ```
   $ grep "D.Anjou pears" dawson.farms
   D'Anjou pears          0.13   3   D
   $ _
   ```

   □ Answer: Yes, Dawson supplies D'Anjou pears.

   A period was used in the search string because an apostrophe (') would have been interpreted as a single quotation mark ('), and an error would have resulted. (Both are the same on the keyboard.)

2. Are anyone's prices over $0.50?

TABLE 13.1. Options for **grep**

| Option | Effect |
|---|---|
| -b | Precede each matched line with a block number. |
| -c | Show how many times the string was found, but not the text. |
| -i | Ignore case during comparisons. |
| -l | Show the name(s) of the file(s) in which the string was found, but not the text. |
| -n | Show line numbers along with the text. |
| -s | Suppress any file error messages that may arise during processing. |
| -v | Invert (match those lines in which the string was *not* found). |

□ Check all three files for all amounts over $0.50:

```
$ grep "[0-9]\.[5-9][0-9]" *farms
dawson.farms:Bing cherries         0.51   5     D
jenkins.farms:Concord grapes       0.51   5     J
$ _
```

□ Answer: Dawson's bing cherries and Jenkins' Concord grapes.

## Using options to modify the output

The **grep** command has many options that you can use to vary the output, as shown in Table 13.1.

You can use more than one of these options in a single **grep** command, but you can't bundle them (combine more than one behind a single minus sign). Here are some examples of **grep** commands with these options:

1. How many varieties of pear does each farm supply?

□ Check all three files for `pear`, but show counts only:

```
$ grep -c pear *farms
dawson.farms: 2
jenkins.farms: 1
parker.farms: 0
$ _
```

□ Answer: Dawson supplies two varieties, Jenkins supplies one, and Parker does not supply any.

2. Which farms supply Thompson grapes?

□ Check all three files for `Thompson grapes`, and show line numbers:

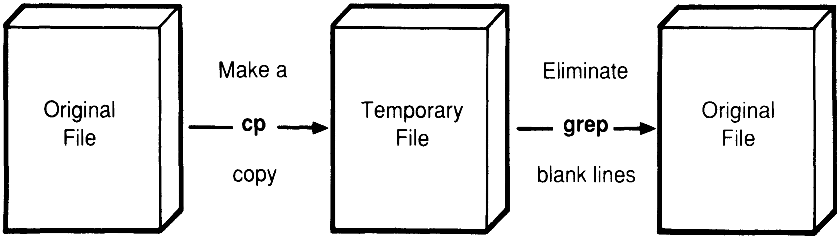
```
$ grep -n "Thompson grapes" *farms
parker.farms:2:Thompson grapes      0.35    5    P
$ _
```

□ Answer: Parker only, line 2.

## Sending the output to a file

As with most UNIX programs, you can redirect the output of **grep** to a file instead of just displaying the results on the screen. Use the redirection symbol (>). Here are two examples, both employing the **-v** (invert) option to delete lines of text. In both examples, the plan is to make a temporary copy of the file, then do the processing from the temporary copy back to the original file (see Figure 13.1).

FIGURE 13.1. Processing from a temporary file.



1. Clear all the blank lines out of jenkins.farms:

   □ Make a temporary copy of jenkins.farms:

   ```
   $ cp jenkins.farms jenkins.temp
   $ _
   ```

   □ Eliminate all blank lines (that is, send only non-blank lines from the temporary file to jenkins.farms):

   ```
   $ grep -v "^$" jenkins.temp > jenkins.farms
   $ _
   ```

   The file jenkins.farms will be overwritten by the output from this **grep** command. (Clearing blank lines this way can be handy.)

2. Update your records to show that Dawson has dropped all varieties of pear:

   □ Make a temporary copy of dawson.farms:

```
$ cp dawson.farms dawson.temp
$ _
```

□ Eliminate all lines that refer to pears (that is, send only non-pear lines from the temporary file to **dawson.farms**):

```
$ grep -v pear dawson.temp > dawson.farms
$ _
```

□ Verify the update:

```
$ cat dawson.farms
Bing cherries          0.51   5   D
Concord grapes         0.48   4   D
$ _
```

## USING RELATED SEARCH PROGRAMS

**fgrep**
**egrep**

Here are two programs in the **grep** family that you may be interested in:

- **fgrep** (fast **grep**)—a scaled down version of **grep** that allows all the options, but no metacharacters for constructing regular expressions
- **egrep** (extended **grep**)—an enhanced version of **grep** that allows you to search for repeated strings and alternate strings

Both of these programs (**fgrep** and **egrep**) allow you to search for an expression that begins with a hyphen, and both allows reading from a file. In addition, **fgrep** allows you to request exact matches of entire lines. The enhancements to these to programs are shown in Table 13.2.

TABLE 13.2. Additional Options for **fgrep** and **egrep**

| **fgrep Option** | **Effect** |
|---|---|
| -e *string* | Match a string that begins with a hyphen |
| -f *file* | Read strings from *file* |
| -x | Match only entire lines |
| **egrep** option | Effect |
| -e *expr* | Match an expression that begins with a hyphen |
| -f *file* | Read expressions from *file* |

To summarize what each program can match during a search:

| | |
|---|---|
| **fgrep** | Literal strings only |
| **grep** | Patterns (strings and regular expressions) |
| **egrep** | Compound expressions |

Here are some examples of **fgrep** and **egrep** searches:

1. Does Jenkins supply Bartlett pears?

   □ Check file **jenkins.farms** for `Bartlett pears:`

   ```
   $ fgrep "Bartlett pears" jenkins.farms
   Bartlett pears          0.12   3   J
   $ _
   ```

   □ Answer: Yes, Jenkins supplies Bartlett pears.

   This is identical to an earlier example, with **fgrep** substituted for **grep**. With **fgrep**, it should run a little faster.

2. Who supplies either Concord grapes or Thompson grapes?

   □ Check all three files for either Concord grapes or Thompson:

   ```
   $ egrep "(Concord|Thompson) grapes" *farms
   dawson.farms: Concord grapes         0.48   4   D
   jenkins.farms: Concord grapes        0.51   5   J
   parker.farms: Thompson grapes        0.35   5   P
   $ _
   ```

   □ Answer: Dawson and Jenkins supply Concord grapes, while Parker supplies Thompson.

## Searching from a file

The following example shows how you could store strings or expressions in a separate file, then have the search program read from the file before beginning the search for matches. This technique can be used with either **fgrep** or **egrep**. Here we use **fgrep**.

1. Store strings in a file:

   □ Enter ***pears*** and ***oranges*** into a file called **juicy**:

   ```
   $ cat > juicy
   pears
   oranges
   CTRL-D
   $ _
   ```

   □ This example is absurd, of course. You would never use a separate file to store just two items; but you would to store twenty-five items.

2. Who supplies pears and oranges?

□ Check all three files for either pears or oranges:

```
$ fgrep -f juicy *farms
jenkins.farms: Bartlett pears     0.12    3    J
jenkins.farms: Navel oranges      0.26    4    J
parker.farms: Valencia oranges    0.29    5    P
$ _
```

□ Answer: Jenkins supplies pears and oranges, while Parker supplies only oranges.

## Sorting files **sort**

You had a brief introduction to the **sort** command in Chapter 4, "Using UNIX Commands." Now we'll discuss it in greater detail. The **sort** command arranges the lines of a file in either alphabetical or numerical order, depending on whether the lines contain strings or numbers. As you will learn in this chapter, you can focus **sort** on specific strings within each line of a file. We'll use the three farms files to illustrate this. But before we start, let's add a few more lines of text to dawson.farms:

```
$ cat >> dawson.farms
Navel oranges         0.26   3   D
Kadota figs           0.42   4   D
Valencia oranges      0.28   5   D
Smyrna figs           0.48   4   D
CTRL-D
$ _
```

### Selecting a Field (or Fields)

If we take a look at dawson.farms after displaying its contents with a **cat** command, we see four columns of text: a list of varieties of fruits, a list of prices, and an identifier. However, **sort** sees the contents as *five* columns (or *fields*) of text, as shown below the display:

```
$ cat dawson.farms
Bing cherries       0.51      5       D
Concord grapes      0.48      4       D
Navel oranges       0.26      3       D
Kadota figs         0.42      4       D
Valencia oranges    0.28      5       D
Smyrna figs         0.48      4       D
$ _
```

| **Field 1** | **Field 2** | **Field 3** | **Field 4** | **Field 5** |
|---|---|---|---|---|
| Bing | cherries | 0.51 | 5 | D |
| Concord | grapes | 0.48 | 5 | D |
| Navel | oranges | 0.26 | 3 | D |
| Kadota | figs | 0.42 | 4 | D |
| Valencia | oranges | 0.28 | 5 | D |
| Smyrna | figs | 0.48 | 4 | D |
| **Variety** | **Type** | **Price** | **Quality** | **Farm** |

Unless we specify otherwise, **sort** views spaces and tabs as *field separators*, marking boundaries between different fields. So, from the point of view of **sort**, field 1 is *variety*, field 2 is *generic type*, field 3 is *price*, field 4 is *quality code*, and field 5 is *farm code*. If we issue a simple **sort** command without specifying any fields or options, we get this ordering (alphabetical, fields 1-4):

```
$ sort dawson.farms
Bing cherries           0.51    5       D
Concord grapes          0.48    4       D
Kadota figs             0.42    4       D
Navel oranges           0.26    3       D
Smyrna figs             0.48    4       D
Valencia oranges        0.28    5       D
$ _
```

Once we see how **sort** views fields, we can indicate to **sort** which field (or sequence of fields) we wish to have sorted, using numbers to represent the different fields:

- A plus sign (+) in front of a number means "*begin* sorting after this field."

- A minus sign (-) means "*stop* sorting after this field."

Using these conventions, we can use the notation +1 -2 to mean, "Begin sorting after field 1 and stop sorting after field 2" (that is, sort field 2 only). Let's try this (see Figure 13.2):

```
$ sort +1 -2 dawson.farms
Bing cherries       0.51      5         D
Kadota figs         0.42      4         D
Smyrna figs         0.48      4         D
Concord grapes      0.48      4         D
Navel oranges       0.26      3         D
Valencia oranges    0.28      5         D
$ _
```

FIGURE 13.2. Sorting field 2 only.

```
$ sort +1 -2 dawson.farms
Bing cherries       0.51      5          D
Kadota figs         0.42      4          D
Smyrna figs         0.48      4          D
Concord grapes      0.48      4          D
Navel oranges       0.26      3          D
Valencia oranges    0.28      5          D
$ _
```

```
          Field 1  | Field 2      | Field 3  Field 4   Field 5
                   |              |
          Bing     | cherries     |  0.51       5         D
          Kadota   | figs         |  0.42       4         D
          Smyrna   | figs         |  0.48       5         D
          Concord  | grapes       |  0.48       4         D
          Navel    | oranges      |  0.26       3         D
          Valencia | oranges      |  0.28       5         D
                   |              |
              +1   |              |  -2
                   |              |
Begin sorting      |              |  Stop sorting after field 2
after field 1      |              |  (don't include fields 3-5
(start with field 2)              |  in this sort)
```

In this example, we are sorting only one field. But, as we'll see in a moment, it's possible to sort one field, then another, and so on, to create secondary sorting. Here is the notation required to include various fields in a sort:

| Notation | field | notation | fields | notation | fields |
|---|---|---|---|---|---|
| +0 -1 | 1 | +0 -2 | 1, 2 | +0 -3 | 1-3 |
| +1 -2 | 2 | +1 -3 | 2, 3 | +1 -4 | 2-4 |
| +2 -3 | 3 | +2 -4 | 3, 4 | +2 -5 | 3-5 |

## 13.1 Using options to modify the output

The **sort** command has a number of options that you can include in the command line to modify **sort**'s output, as shown in Table 13.3.

Note that you can use more than one option in a given **sort** command line, and that options can be bundled. The most logical candidates for field separator are colon ( : ) and vertical bar ( | ). If you decide to use a vertical bar, remember that it must be escaped with a backslash (typed as **\|**).

Let's illustrate the use of some of these options by sorting on price (highest price first). To do this correctly, we need to use the **n** (numeric) and **r** (reverse) options. We could enter the command line this way:

TABLE 13.3. Options for **sort**

| Option | Effect |
|---|---|
| -b | Ignore leading blanks (spaces and tabs) when comparing fields—useful if items in a field vary in length. |
| -n | Sort a numeric field (allow for optional blanks, minus signs, zeroes, or decimal points)—this option includes **-b** automatically. |
| -M | Sort a field that contains months (JAN, FEB, ..., DEC) |
| -r | Sort in reverse order (Z-A, z-a, 9-0) |
| -f | Fold uppercase letters onto lowercase (for example, treat `CASE`, `Case`, and `case` as identical when sorting). |
| -d | Sort in dictionary order. |
| -i | Ignore non-printing characters when sorting. |
| -u | Sort uniquely (if lines are identical, discard all but the first). |
| -m | Merge several files: the files are already sorted; just merge them. |
| -c | Make sure the input file has already been sorted. |
| -t *x* | Make *x* (any character) the field separator. |
| -y *k* | Set aside *k* kilobytes of memory for this sort. |
| -z *n* | Allow no more than *n* characters per line of input. |
| -o *file* | Place the output in *file*—the same as **>** *file*. |

```
$ sort -nr +2 dawson.farms
Bing cherries           0.51    5  D
Smyrna figs             0.48    4  D
Concord grapes          0.48    4  D
Kadota figs             0.42    4  D
Valencia oranges        0.28    5  D
Navel oranges           0.26    3  D
$ _
```

[Dawson fruits in order of price (highest price first)]

If we type it this way, it means that **n** and **r** apply to *all* fields to be sorted. Since we are sorting on just one field, it doesn't make any difference in this example. But if we sort one field after another, as in the example that follows, we have to bundle **n** and **r** with the price field indicator. **n** and **r** will then apply only to that one field.

```
$ sort +2nr -3 +0 -2 dawson.farms
Bing cherries           0.51    5  D
Concord grapes          0.48    4  D
Smyrna figs             0.48    4  D
Kadota figs             0.42    4  D
Valencia oranges        0.28    5  D
Navel oranges           0.26    3  D
$ _
```

[Dawson fruits in order by price (alphabetical order under identical prices)]

Comparing this example to the previous one, you will notice one slight difference: `Concord grapes` moved ahead of `Smyrna figs` on the list (`Concord` before `Smyrna`). Figures 13.3 and 13.4 show what happened on each pass.

FIGURE 13.3. First pass: sorting fields 1 and 2.

| **Field 1** | **Field 2** | **Field 3** | **Field 4** | **Field 5** |
|---|---|---|---|---|
| Bing | cherries | 0.51 | 5 | D |
| Concord | grapes | 0.48 | 4 | D |
| Kadota | figs | 0.42 | 4 | D |
| Navel | oranges | 0.26 | 3 | D |
| Smyrna | figs | 0.48 | 5 | D |
| Valencia | oranges | 0.28 | 5 | D |

```
+0
Begin sorting
after field 0
(start with
field 1)
```

```
-2
Stop sorting after field 2
(don't include fields 3-5
in this sort)
```

FIGURE 13.4. Second pass: sorting field 3.

| **Field 1** | **Field 2** | **Field 3** | **Field 4** | **Field 5** |
|---|---|---|---|---|
| Bing | cherries | 0.51 | 5 | D |
| Concord | grapes | 0.48 | 4 | D |
| Smyrna | figs | 0.48 | 5 | D |
| Kadota | figs | 0.42 | 4 | D |
| Valencia | oranges | 0.28 | 5 | D |
| Navel | oranges | 0.26 | 3 | D |

```
+2

Begin sorting after
field 2 (start with
field 3)
```

```
-3

Stop sorting after
field 3 (don't in-
clude fields 4 and
5 in this sort)
```

## Sending the output to a file

As with most UNIX programs, you can redirect the output of **sort** to a file, instead of just displaying the results on the screen. You can use either the redirection symbol (>) or the **-o** option interchangeably. Here are two examples, both involving sorting multiple files:

1. Store a list from all three suppliers sorted *by generic type* in **fruit.sort**:

□ Perform the sort:

```
$ sort +1 -2 *farms > fruit.sort
$ _
```

□ Display the results:

```
$ cat fruit.sort
Bing cherries           0.48   4   P
Bing cherries           0.51   5   D
Kadota figs             0.42   4   D
Smyrna figs             0.48   4   D
Concord grapes          0.48   4   D
Concord grapes          0.51   5   J
Thompson grapes         0.35   5   P
Navel oranges           0.26   3   D
Navel oranges           0.26   4   J
Valencia oranges        0.28   5   D
Valencia oranges        0.29   5   P
Bartlett pears          0.12   3   J
$ _
```

Fruits from all suppliers in alphabetical order by generic type

2. Store a list from all three suppliers sorted by price in **price.sort**:

□ Perform the sort:

```
$ sort -o price.sort +2nr -3 +1 -2 *farms
$ _
```

□
```
$ cat price.sort
Bing cherries           0.51   5  D
Concord grapes          0.51   5  J
Bing cherries           0.48   4  P
Smyrna figs             0.48   4  D
Concord grapes          0.48   4  D
Kadota figs             0.42   4  D
Thompson grapes         0.35   5  P
Valencia oranges        0.29   5  P
Valencia oranges        0.28   5  D
Navel oranges           0.26   3  D
Navel oranges           0.26   4  J
$ _
```

Fruits from all suppliers in order by price

In this file (**price.sort**), we have all fruit listed in order of price (highest price first).

## 13.2 Summary

In this chapter you learned how to search for text with **grep** and sort files with **sort**.

## Searching for text

To conduct a simple search for a string in a file (or files), type ***grep***, a space, the string, another space, and the name(s) of the file(s) in a command line and press (RETURN). Rather than just type a literal string, you can form a *regular expression* with UNIX *metacharacters*: `^ $ . * [ ] \`

By including an option (or combination of options) in the command line, you can modify the output of **grep** to show how many times the string was found, to show the name(s) of the file(s) in which the string was found, to show line numbers, to match either upper case or lower case, to invert the search, and so on. To send the output to a file, redirect the standard output with > and a filename.

For faster searches with only fixed literal strings, use **fgrep**; for enhanced searching with provisions for finding repeated strings and alternate strings, use **egrep** in place of **grep**.

## Sorting files

To perform a simple sort of entire lines of text, type ***sort***, a space, and the name(s) of the file(s) to be sorted in a command line and press (RETURN). To perform sorting on a particular field (or fields), use the plus and minus options with field numbers.

By including an option (or combination of options) in the command line, you can modify the operation of **sort** to ignore leading blanks, sort a numeric field, sort in reverse order, fold upper case onto lower case, sort uniquely, merge files, designate a *field separator*, redirect the output, and so on.

# 14

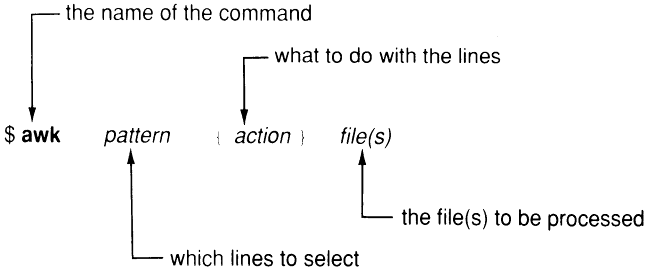
# Programming with **awk**

## 14.1 Introduction

The **awk** program, which processes files that contain either text or numerical data, allows you to rearrange fields, perform arithmetic operations, and retrieve lines selectively, using programming statements that resemble those used in C. You can use it to set up a spreadsheet in UNIX or to generate reports from files already stored in the system. (The name **awk** is derived from the names of its originators: *A*ho, *W*einberger, and *K*ernighan.)

A full command line for **awk** contains four parts: the name of the command, an optional *pattern* statement, an optional *action* statement, and the name(s) of the file(s). The *pattern* statement is used to select lines from a file; the *action* statement, which must be enclosed within braces, is used to decide what to do with the lines selected. A general **awk** command line is shown in Figure 14.1.

FIGURE 14.1. A general **awk** command line.


the name of the command
what to do with the lines
$ awk pattern { action } file(s)
the file(s) to be processed
which lines to select

Once again, we'll be using the three files in our farms series.

### The defaults

You may omit either the pattern statement or the action statement, but not both. If you omit the pattern statement, **awk** will select by default every line in the file named. For example, in the absence of a pattern statement, **awk** will print (display) every line of jenkins.farms unchanged:

```
$ awk '{ print }' jenkins.farms              [No pattern statement]
Concord grapes          0.51    5       J
Bartlett pears          0.12    3       J
Navel oranges           0.26    4       J
$ _
```

If you omit the action statement, **awk** will copy each selected line to the standard output by default. For example, in the absence of an action statement, **awk** will copy each line selected to the standard output unchanged:

```
$ awk '/orange/  ' jenkins.farms             [No action statement]
Navel oranges           0.26    4       J
$ _
```

If there's even the remotest possibility that your pattern and action statements may contain special characters, enclose them within single quotes.

## Referring to fields

Like **sort**, **awk** is oriented to *fields*, and offers a simple method for referring to fields by name. Just type a dollar sign (`$`) in front of a number to refer to a field. By this convention, `$1` means field 1, `$2` means field 2, `$3` means field 3, and so on, as shown in Figure 14.2. The notation `$0` has a special meaning: all fields (the entire record).

FIGURE 14.2. Naming fields in **awk**.

| Field 1 | Field 2 | Field 3 | Field 4 | Field 5 | |
|---|---|---|---|---|---|
| Concord | grapes | 0.51 | 5 | J | Entire record (`$0`) |
| **$1** | **$2** | **$3** | **$4** | **$5** | |

Using these conventions, we can display fields selectively by naming or not naming them in the action statement. For example, to display only prices, we could use this:

```
$ awk '{ print $3 }' jenkins.farms
0.51
0.12
0.26
$ _
```

We can also switch fields 1 and 2 (and then place field 3 at a tab stop):

```
$ awk '{ print $2 ", " $1 "TAB" $3 }' jenkins.farms
grapes, Concord         0.51
pears, Bartlett         0.12
oranges, Navel          0.26
```

```
$ _
```

The *action* statement requests the following: Print field 2 (generic type of fruit), a comma and a space, field 1 (variety of fruit), a TAB, and then field 3 (price). (When you press the TAB key, the cursor will, of course, advance to the next tab stop on the screen, leaving a gap in your command. Do not become alarmed when this happens.) That's all there is to it; we have switched fields 1 and 2.

## Placing statements in a file

In the examples above, the command lines were fairly short, but often the command line becomes too long to fit on a single line. One way to get around this (and to avoid repetitious typing) is to place your statements in a separate file, then activate this file in the **awk** command line with the **-f** option. Here's an easier way to type the command line for switching fields 1 and 2:

1. Place the action statements (without the single quotes) in a file:

   □ Enter the text with **cat**:

   ```
   $ cat > sw
   { print $2 ", " $1 "TAB" $3}
                  [CTRL-D to terminate input]
   $ _
   ```

   □ Now **sw** contains the pattern (empty) and action statements, and we won't have to type them again on the command line.

2. Name this file when you execute **awk**:

   □ Use the **-f** option to have **awk** read the pattern and action statements stored in **sw**:

   ```
   $ awk -f sw jenkins.farms
   grapes, Concord         0.51
   pears, Bartlett         0.12
   oranges, Navel          0.26
   $ _
   ```

   □ This command line is equivalent to the command line in the previous example (under "Referring to Fields").

Any time you have to run the same statements repeatedly, placing them in a file is much easier than having to type them all over again for each command line.

## Placing the entire command line in a file

In the example above, we placed only the action statement in another file. By using several more features of UNIX, we can also place the entire **awk** command line in another file:

1. Place the command line (including the single quotes) in a file:

   □ Enter the text with **cat**, using **dawson.farms** this time:

   ```
   $ cat > sw.dawson
   awk '{ print $2 ", " $1 "TAB" $3}' dawson.farms
                 [CTRL-D to terminate input]
   $ _
   ```

   □ Now **sw.dawson** contains the entire command line.

2. Make this file executable:

   □ Use the **chmod** command from the shell prompt:

   ```
   $ chmod u+x sw.dawson
   $ _
   ```

   □ Now **sw.dawson** has become a UNIX command.

3. Try using the command:

   □ Type ***sw.dawson*** as a command after the shell prompt (because of its length, the line for "oranges, Valencia" won't be aligned with the other lines):

   ```
   $ sw.dawson
   cherries, Bing   0.51
   grapes, Concord  0.48
   oranges, Navel   0.26
   figs, Kadota     0.42
   oranges, Valencia       0.29
   figs, Smyrna     0.48
   $ _
   ```

   □ This command line is similar to the command line in the previous example, but restricted to only one file.

The disadvantage of **sw.dawson** is that is works only for one file (**dawson.farms**). However, we can fix that by using the shell variable `$1`—which is unrelated to the **awk** variable `$1`. (Shell variables are introduced and explained in detail in Part V.)

4. Place another command line (including the single quotes) in a file:

□ Enter the text with **cat**, using `$1` this time:

```
$ cat > switch
awk '    { print $2 ", " $1 "TAB" $3}' $1
              [CTRL-D to terminate input]
$ _
```

□ Now **switch** contains the entire command line, including a shell variable (`$1`) to allow you to type any filename.

5. Make this file executable:

□ Use the **chmod** command from the shell prompt:

```
$ chmod u+x switch
$ _
```

□ Now **switch** has become a UNIX command.

6. Try using the command:

□ Type ***switch*** as a command with **parker.farms** as its argument after the shell prompt:

```
$ switch parker.farms
cherries, Bing          0.48
grapes, Thompson        0.35
oranges, Valencia       0.29
$ _
```

□ This command line is similar to the command line in the previous example (under "Placing Statements in a File"), but more general.

The shell variable `$1` leaves the filename open, so that you can type it as an argument after the command name (**awk**). For more about this shell variable, see the section entitled "Positional Parameters," page 360, in Chapter 23, "Bourne Shell Variables."

## 14.2 Using the **awk** program

To sum up what you've learned so far, the **awk** program allows you to work with lines of text (records) by selecting lines matched in a pattern statement, then processing those lines from a programming script given in an action statement. Based largely on the **C** language, **awk** allows reserved variables, user-defined variables, a variety of pattern-matching constructions (such as regular expressions, relational operators, ranges, and compound expressions), and a number of programming features (such as built-in functions, arrays, and program control statements). We'll discuss these now, beginning with built-in variables.

## BUILT-IN VARIABLES

The **awk** program views the input file as a sequence of records (lines of text), each divided into fields, as shown in Figure 14.3. Associated with these entities are a number of built-in variables, some of which you have already become acquainted with.

FIGURE 14.3. Records and fields.

```
                  Field 1   Field 2  Field 3   ...   Field n


Record 1            $1    |  $2    |  $3    | ... |  $n      /
Record 2            $1    |  $2    |  $3    | ... |  $n      /
Record 3            $1    |  $2    |  $3    | ... |  $n      /
    ⋮

Record m            $1    |  $2    |  $3    | ... |  $n
```

Referring to Figure 14.3, each record contains *n* fields, each separated from other records by a field separator (symbolized as |). The entire file, in turn, contains *m* records, each separated from other records by a record separator (symbolized as /). The built-in variables that describe all this are shown in Table 14.1, with the symbols used in Figure 14.3 for easy identification.

TABLE 14.1. Built-In Variables Used by **awk**

| Variable | Description | Default | Symbol |
|---|---|---|---|
| **$1-$n** | Individual fields | None | **$1-$n** |
| **NF** | Number of fields in a record | None | **n** |
| **FS** | Input field separator | Space or tab | **+** |
| **OFS** | Output field separator | Space | None |
| **$0** | The entire current record | None | **$0** |
| **NR** | Number of the current record | None | None |
| **RS** | Input record separator | Newline | / |
| **ORS** | Output record separator | Newline | None |
| **FILENAME** | Name of the current input file | None | None |

You can change any of the variables with default values in an **awk** program with an assignment statement like this: `FS = ":"`. (You can also change the field separator by including the **-F** option on an **awk** command line, like this: **-F:**.)

## Separating fields

If you type a comma between field names in an action statement, **awk** will use OFS (by default a space) to separate them in the output; if you type only a space between them, **awk** will concatenate the fields in the output:

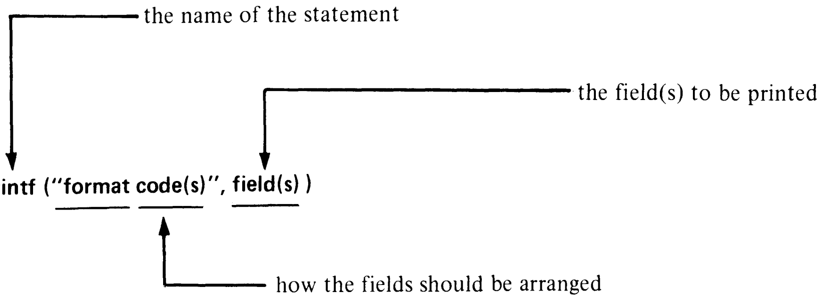
```
{ print $4, $5 } [Separate fields 4 and 5 with a space when printing]
{ print $4 $5 }  [Concatenate fields 4 and 5 when printing]
```

To place a field at a tab stop, type the TAB between double quotes in front of the field name (this will leave a gap between the double quotes).

## Printing

In this chapter, we use mainly the simple `print` statement. But **awk** also allows custom formatting via the `printf` statement, which is identical to the `printf` statement used in the **C** language. Whereas the `print` statement simply prints each line "as is," the `printf` statement gives you more control over the appearance of each printed line of text. The general format of a `printf` statement is shown in Figure 14.4.

FIGURE 14.4. A general **printf** statement.


the name of the statement
the field(s) to be printed
$ printf ("format code(s)", field(s) )
how the fields should be arranged

Format codes for control characters begin with backslashes, such as `\t` to represent a tab and `\n` to represent a newline. Format codes for fields begin with percent signs; each code includes a number to indicate the width of the field in columns and a letter to indicate the general type of field. For example, `%8s` represents a field *8* columns wide that contains a *s*tring.

Numeric fields for decimals must also indicate how many columns must be reserved for the digits that follow the decimal point. This is done by typing a decimal point and a second number after the number that indicates the total width of the field. For example, `%6.2f` represents a numeric field *6* columns wide, with *2* columns reserved for digits that follow the decimal point, that contains a *f*loating-point (non-integral numeric) field.

Fields other than floating-point are understood to be right-justified. Floating-point fields are justified on the decimal point. To make a field left-justified, you can type a hyphen (or minus sign) immediately after the percent symbol.

The fields specified in *format code(s)* (between the quotation marks) must match the fields named in *field(s)*. In other words, there must be one field specifier preceding the comma for each field name following the comma. The field specifier before the comma provides formatting information for its corresponding field. For example, you could use the following statement to produce the results shown:

```
$ awk '{ printf("%8s\t%8s\t%6.2f\n", $1, $2 $3 )}' \
> parker.farms

    Bing cherries        0.51
Thompson   grapes        0.51
Valencia  oranges        0.51
```

In this example, the first `%8s` matches `$1` (the first field), the second `%8s` matches `$2` (the second field), and `%6.2` matches `$3` (the third field). Fields are generally separated by tabs, and each line is generally followed by a newline.

The general format for each individual field format indicator is

```
%[-][ F.[ D]] T
```

where - means left-justification, *F* is the field width, *D* is the number of decimal places, and *T* indicates the type of field:

| | |
|---|---|
| s | String |
| d | Decimal |
| f | Floating point |

In the following example, each entire record (`$0`) is interpreted as being the "second field" (even though each record actually contains five fields). This "second field" is preceded by the number of the current record (`NR`) and followed by the number of fields in each record (`NF`). This example illustrates how you can use **awk**'s built-in variables, as well as a creative way of defining a "field."

```
$ awk '{ printf (" %3d \t %25s \t %3d \n ", NR, $0, NF) }' \
> jenkins.farms

1     Concord grapes     0.51    5      J        5
2     Bartlett pears     0.12    3      J        5
3      Navel oranges     0.26    4      J        5
```

Other field types for those who may be interested are as follows:

| | |
|---|---|
| c | Single character |
| u | Unsigned decimal |
| o | Octal (base 8 numeric) |
| x | Hexadecimal (base 16 numeric) |
| e | Exponential (scientific notation) |
| g | Shorter of floating point (f) or exponential (e) |

## 14.3 Search patterns

In **awk** pattern statements you can use all of the search facilities of **grep**, plus a few borrowed from other UNIX programs. These include pre- and post-processing, regular expressions, arithmetic and relational operators, ranges, and compound statements.

### Initial and end processing

A special BEGIN statement allows you to perform any pre-processing required before reading the first record from the input file. Another statement, called END, allows you to perform any post-processing required after the last record has been read. Typically, BEGIN is used to initialize any variables, if necessary, and END is used to produce summary information. Here is a very simple example, with comments on the right:

```
BEGIN { FS = ":" }  # Use a colon as a field separator
      { other statements }
END   { print NR }  # Print the total number of records
```

### Regular expressions

Once again, **awk** uses all the search facilities of **grep**, including regular expressions, relational operators, ranges, and compound statements. To summarize from Chapter 13, **grep** allows you to use special characters, or *metacharacters*, match patterns. The expressions formed from these characters are called *regular expressions.* Here is a quick review of these characters:

| | | | |
|---|---|---|---|
| ^ | Beginning of line | $ | End of line |
| . | Any single character | * | Repeat any number of times |
| [ ] | Any enclosed character | \ | Turn off special meaning |

Here are some examples of regular expressions formed from these characters, with a few strings they would match if used in an **awk** statement:

| Regular Expression | Some of the Strings Matched |
|---|---|
| `"^3."` | 30 3a 3+ 3) (at beginning of a line) |
| `"t..$"` | too t36 t—T t+5 (at end of a line) |
| `"[Ff]arm"` | Farm farm |
| `"\$[0-9]\.[0-9][0-9]"` | $4.67 $8.32 $1.99 $3.71 |
| `"[abc]\[i\]"` | a[i] b[i] c[i] |
| `"^$"` | (A blank line) |

To match an expression with a particular field, **awk** uses tilde for match (~) and exclamation-tilde for no-match (!~). Here are two examples:

| | |
|---|---|
| **$5 ~ /[Ff]ox/** | Select any record that contains either "Fox" or "fox" in field 5. |
| **$1 !~ /^999$/** | Select any record other than one that contains "999" and nothing else. |

## Arithmetic and relational operators

You can use any of the arithmetic operators found in the **C** language:

| | | | |
|---|---|---|---|
| **+** | Add | ***** | Multiply |
| **-** | Subtract | / | Divide |
| | | **%** | Take remainder |

You can also use the relational operators to perform comparisons:

| | | | |
|---|---|---|---|
| **<** | Less than | **>** | Greater than |
| **<=** | Less than or equal to | **>=** | Greater than or equal to |
| **==** | Equal to | **!=** | Not equal to |

You can also use the assignment operators to work with variables:

| | | | |
|---|---|---|---|
| **++** | Increment the variable | **–** | Decrement the variable |
| **+=** | Add and assign | **-=** | Subtract and assign |
| ***=** | Multipy and assign | /= | Divide and assign |
| | | **%=** | Take remainder and assign |

Here are a few examples of using these operators in pattern-searching:

| | |
|---|---|
| **$5 >$4 + 40** | Select a record if field 5 is larger than field 4 by at least 40 |
| **NF % 2 == 1** | Select odd-numbered fields |
| **$1 >= "x"** | Select any record that begins with x, y, or z |

As another example, let's have **awk** go through each line (or *record*) of each of the farms files and display those that show a price higher than 0.50. Since this one is so simple, we won't bother with a separate file (Which fruits from all suppliers cost more than 0.50?):

```
$ awk '$3 > 0.50 { print $0}' *farms
cherries, Bing          0.51    D
grapes, Concord         0.51    J
$ _
```

So there it is—a list of all the fruits that cost more than 0.50. The *pattern* statement (`$3 > 0.50`) selects these lines, and the *action* statement (`print $0`) displays them. By the way, you could save the output lines in a file by just adding a redirection statement (> exp.fruits) to the end of the command line before executing it.

## Setting ranges

By entering two search patterns, separated by a comma, you can set a range of lines for **awk** to select. Here are some examples:

**NR==11,NR==15** Select records 11, 12, 13, 14, and 15.

**/^000$/,/^999$/** Select all records between, and including, the record that contains only "000" and the record that contains only "999".

## Forming compound statements

By using the following logical operators, you can form compound search patterns for selecting lines:

| | |
|---|---|
| **\|\|** | OR |
| **&&** | AND |
| **!** | NOT |

Here are some examples:

**$3 == ”10” && $4 <”D”** Select any record for which field 3 is equal to 10 and field 4 begins with A, B, or C

**$1 <”b” && $1 != ”ace”** Select any record that begins with a, as long as it doesn't contain "ace"

Here is another example, using the farms files (Which fruits that have a quality code of 5 begin with A, B, or C?):

```
$ awk '$4=="5" && $1< "D" { print $0 }' *farms
Bing cherries       0.51      5         D
Concord grapes      0.51      5         J
$ _
```

Here is yet another example (Which grapes cost between 0.31 and 0.35 inclusive?):

```
$ cat > grapes.35
$1 ~ /grapes/ && $3 >= 0.31 && $3 <= 0.35 { print $0}
                    [CTRL-D to terminate input]
$ _

$ awk -f grapes.35 *farms
grapes, Thompson        0.35    5       P
$ _
```

## 14.4 Action statements

When constructing action statements, you can use built-in functions, your own variables, arrays, and program control statements. We'll discuss each of these now briefly.

### Using built-in functions

Your statements can include the built-in functions shown in Table 14.2.

### Assigning your own variables

You can make assignments to any variables you wish to use, without previous declaration or initialization (**awk** automatically initializes all user-defined variables to the null string). All user-defined variables can be freely converted between floating-point numeric and string. Here are some examples:

| | |
|---|---|
| **x = 5** | Assign 5 to x, making x floating point |
| **y = "quite"** | Assign quite to y, making y a string variable |
| **z = "3" + "4"** | Assign 7 to z, making z floating point |
| **y = "quite" + 0** | Add zero to y, converting y to floating point |
| **x = 5""** | Concatenate a null string to x, converting x to a string variable |

Field variables, as well as user-defined variables, can be reassigned new values at will, regardless what their previous contents may have been.

### Using arrays

As when using **C**, you are allowed to use arrays, which in **awk** require neither declaration nor initialization. You can use either numeric or string variables as indices in an **awk** array. In the following example, we place

TABLE 14.2. Built-In Functions Used by **awk**

| Function | Description |
|---|---|
| **length [***£i***]** | Without an argument, returns the length of the current record; with an argument, returns the length of the field named as an argument |
| **split(***string,array***[,***fs***])** | Split string *string* into *n* fields, place them in separate array elements *array[1]*, *array[2]*, . . . , *array[n]*, and return the number of elements *n*. If the optional field separator *fs* is omitted, then the value of **FS** becomes the default |
| **substr(***string,p***[,***max***])** | Returns the number of elements in the substring of *string* that begins at position *p* and contains no more than *max* characters |
| **index(***STRING,string***)** | Returns the position in *STRING* where *string* begins (or 0 if *string* doesn't occur) |
| **sprintf(***format*,$ex_1$,$ex_2$,. . .**)** | Returns the value of the expressions named in **printf** format statement *format* |
| **sqrt(***n***)** | Returns the square root of *n* |
| **log(***n***)** | Returns the natural logarithm of *n* |
| **exp(***n***)** | Returns the value of *e* to the power *n* |
| **int(***n***)** | Returns the truncated integral value of *n* |

field 1 of a record into an array named *name* and field 2 into an array named *value*, using the current record number as the index for each array:

```
{ name[NR] = $1; value[NR] = $2 }
```

Here is an example from the **farms** series (What is the generic type and price of each item produced on Jenkins' farms?):

```
$ cat array
        { type[NR] = $2; price[NR] = $3 }
END     { for (i=1; i<=NR; i++)
                  print type[i] "TAB" price[i] }
$ awk -f array jenkins.farms
grapes   0.51
pears    0.12
oranges  0.26
$ _
```

If you would like to split a string into individual characters, placing the characters into an array for subsequent reference, you can use the **split** function, shown here:

```
{ n = split(arctic,alpha) }
```

This statement sets n to 6 and produces the following array:

```
alpha[1] = a                    alpha[4] = t
alpha[2] = r                    alpha[5] = i
alpha[3] = c                    alpha[6] = c
```

## Using program control statements

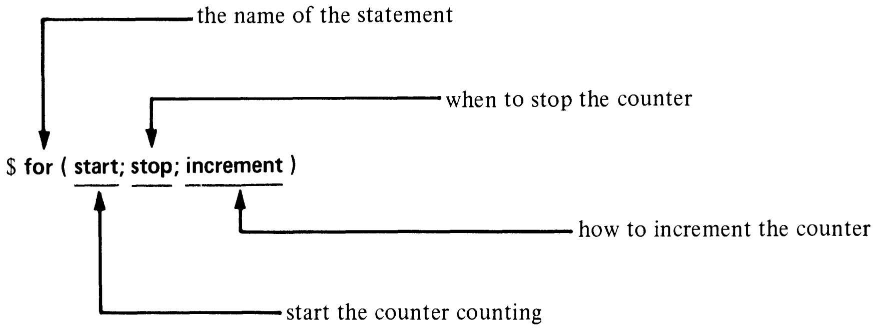
**awk** allows you to use the program control statements of the **C** language (such as `if...else`, `while`, and `for`). It also allows you to use the directives shown in Table 14.3.

TABLE 14.3. Directives Used by **awk**

| Directive | Action |
|---|---|
| next | Proceed to the next record and begin looking for a match |
| continue | Proceed to the next iteration of the current loop |
| break | Exit from the current `while` or `for` loop immediately |
| exit | Proceed as if the last record had just been read |

You can also use the `for` statement, which allows you to perform a sequence of steps under the control of a "counter." For each number provided by the counter, one step is carried; then the counter goes to the next number for another setp; and so on. It is customary, but not mandatory, to use the letters `i`, `j`, `k`, and `l` as counters. The general form of a `for` statement is shown in Figure 14.5.

FIGURE 14.5. A general **for** statement.


the name of the statement
when to stop the counter
$ for ( start; stop; increment )
how to increment the counter
start the counter counting

In the following example, the counter (`i`) starts at 1; the counter stops at 9 (or continues counting as long as it's less than or equal to 9); and the counter is incremented by 2 for each step in the procedure:

```
for (i=1; i<=9; i+=2)
```

Here is a more involved example, in which the for statement is used to print fields 1, 3, and 5 of each record in a file. (Note how the field numbers have been offset from the counter i to select the correct fields.)

```
for (i=5; i<=NF; i+=5)
  printf(" %3.0u \t %12s \t %6.2f \n ",$(i-4),$(i-2),$i)
```

You can also use another form of the statement for arrays; this form allows you to step through each element in the array. For example, suppose we'd like to know the average price of each generic type of fruit sold by any of our three farms. We could do something like this:

1. Place the *action* statements (without the single quotes) in a file:

   □ This time, call the file *av.price*:

   ```
   $ cat > av.price
          { price[$2] += $3 ; n[$2]++ }
   END    { for (type in price)
             print type, "TAB", price[type]/n[type]}
              [CTRL-D to terminate input]
   $ _
   ```

   □ Now the pattern and action statements are in a file named av.price.

2. Name this file with the **-f** option when you execute **awk**:

```
$ awk -f av.price *farms
oranges              0.275
cherries             0.495
figs     0.45
pears    0.12
grapes   0.446667
$ _
```

You could also do something similar with the quality code:

```
$ cat av.quality
       { quality[$2] += $4 ; n[$2]++ }
END    { for (type in quality)
           print type, "(TAB)", quality[type] / n[type] }

$ awk -f av.quality *farms
oranges              4.0
cherries             4.5
figs     4.0
pears    3.0
grapes   4.666667
$ _
```

### More examples

This conclusides our discussion of **awk** programming, which allows you to do a lot of sophisticated processing with direct, straightforward statements.

## 14.5 Error messages

If you enter a statement incorrectly, **awk** will terminate your program and display an error message. For example, suppose you accidently omit the symbol to increment the variable in **av.price** (++), like this:

```
$ cat av.price
       { price[$2] += $3 ; n[$2]    }
END    { for (type in price)
           print type, "(TAB)", price[type] / n[type] }
$ _
```

When you attempt to use **av.price** in a **awk** command line, you will see something like this:

```
$ awk -f av.price *farms
awk: syntax error near line 3
awk: bailing out near line 3
$ _
```

The remedy is to insert the missing plus signs after `n[$2]` and re-enter the command line.

## 14.6 Summary

In this chapter you learned how to process text with **awk**.

## Introduction

An **awk** command line consists of the name of the command (**awk**), a space, an optional pattern statement, a space, an optional action statement, and the name(s) of the file(s). The default pattern statement is, "Select every line in the file;" the default action statement is, "Copy each line selected to the standard output." To refer to fields in an **awk** command, use a dollar sign and a number typed together (for example, `$3` to indicate field 3).

Like many other UNIX commands, **awk** allows you to place your statements in a separate file, then activate them from a command line. With **awk**, use **-f** and the name of the file to activate pattern and action statements only. To place an entire command line in a file, enter the statements, store the file, make it executable with **chmod**, then type the name as a UNIX command.

## Using the **awk** program

Here is some general information about the **awk** program as a whole. Built-in variables allow you to identify fields and records, the number of fields in a record, the number of the current record, field and record separators, and the name of the current file. You can reassign any variable that has a default value.

You have the option of either separating or concatenating two fields when sending them to the output. You have the option of using either the ordinary `print` statement or the custom `printf` statement. The `printf` statement allows you to specify a page format with great precision.

## Search patterns

Whenever you construct a pattern statement, you have available to you pre- and post-processing statements, regular expressions, arithmetic and relational operators, ranges, and compound statements.

You can use `BEGIN` as a pattern statement to provide for any initialization to be performed before reading the first record; you can use `END` to provide for any final processing to be performed after reading the last record.

You can use any of the search facilities of **grep** to match strings (beginning and end of line, wild card, repeat, and character sets), along with symbols to indicate either match or no match. You can use **C**'s arithmetic, relational, and assignment operators to construct pattern statements. You can set a range of lines to be selected by typing two search patterns, separated by a comma. You can use logical operators to construct compound search statements.

## Action statements

Whenever you write an action statement, you have available to you **awk**'s built-in functions, built-in and user-defined variables, arrays, and program control statements.

You can use **awk**'s own functions for returning length of a record, field, or string; the number of elements in a string or substring; position of a substring in a string; the value of the expressions given in a `printf` statement; and also square root, natural logarithm, exponentiation, and integer value of a number.

You can either use the built-in variables provided by **awk** or assign your own. You can use your own variables without prior declaration or initialization, and then convert them freely between numeric and string types (string is the default). You can also use arrays, again without prior declaration or initialization, and you can use either numbers or strings as indices.

You can use various program control statements to proceed to the next record, proceed to the next iteration of the current loop, exit from the current loop, or proceed to the `END` statement (if present).

# 15

# Programming with C

In this chapter we'll take a brief look at the **C** language. We won't describe the features of the language in any detail, but simply show how to compile and run a program in UNIX. (Actually, you'll find more information about **C** in Chapter 14, "Programming with **awk**," than you'll find in this chapter.) This chapter is really for people who are mainly interested in how to generate **C** programs in a UNIX environment. For detailed information about the language itself, see the list of books at the end of the chapter.

In this chapter it is assumed that you are using **vi** to enter text. We'll begin with a discussion of executing UNIX commands without leaving **vi**. Since **C** produces programs, you may need to run some of them while you are editing.

## 15.1 Staying in an editing session

Suppose you're in the middle of an editing session with **vi** and you need to execute a UNIX command. For example, you may want to run **date** to find out what time it is or you may want to run **wc** to check the length of a file. Ordinarily, you would have to end your editing session, run the UNIX command from the shell prompt, then return to your editing session. UNIX has a feature that allows you to *escape* temporarily without having to end your editing session: just precede the UNIX command with a colon and an exclamation mark (**:!**). The command will be handed over to the shell to be executed, and you won't have to leave **vi**. This is known as *escaping* to the shell.

### Checking the time

For example, suppose you're in the middle of an editing session and you'd like to check the time. Let's try this:

1. Start an editing session with a new file:

   □ Start **vi** with spaces:

```
$ vi spaces
```

□ Type the text with blank lines between:

```
This is line 1

                [Two blank lines]
This is line 2

                [Three blank lines]

This is line 3
```

2. Now check the time:

□ Type a colon and an exclamation mark (**:!**) in front of the **date** command:

```
:!date
Tue Mar 23 11:36:43 PST 1987
!
_
```

□ Now you can resume editing (or execute another UNIX command).

### Checking the length of a file

As another example, you could use an escape (**:!**) to determine the length of a file. Continuing the steps just given above, use the **wc** (word count) command to display the statistics on **wall**:

```
:!wc wall
     24      138      1002   wall
!
_
```

This tells you that **wall** contains 24 lines, 138 words, and 1002 characters.

## 15.2 Executing a C program

### Compiling as an example

In the previous section you learned how to execute a UNIX command without leaving **vi**. In this section we'll combine that concept with a very brief discussion of C programming. We'll enter a C program and then compile the program. This book is not about C programming, so we will use only simple C programs in this chapter. The main point of this chapter is not how to write a C program, but how to execute a UNIX command during an editing session with **vi**. Some of the UNIX commands we'll be using here will be the C compiler **cc**, **chmod**, **ls**, and **sort**.

## Starting with a new directory

Once again, let's start this chapter by creating a new subdirectory in your home directory to contain your C programs, then move to this directory for the exercises in this chapter.

1. Create a new directory:

   □ From your home directory, create a directory called **c_progs** with the **mkdir** command:

   ```
   $ mkdir c_progs
   $ _
   ```

   □ The directory has been created, but you are still in your home directory.

2. Move from your home directory to **c_progs**:

   □ Move to subdirectory **c_progs** with the **cd** command:

   ```
   $ cd c_progs
   $ _
   ```

   □ **c_progs** is now your working directory.

## Entering the first program

In this section you will use **vi** to enter a short C program, compile the program, make corrections with **vi**, compile again, and then run the program. To run the C compiler **cc** during an editing session, precede the command with a colon and an exclamation point (*:!*). Once compilation is completed, another exclamation point will appear and control will return to **vi**.

The C program you copy from directory **text** will convert gallons to liters for six specified amounts (10, 12, 14, 16, 18, and 20 gallons). You can also make adjustments in the variables to convert other amounts. Before you begin step 1, make sure you are still in subdirectory **c_progs**. If you aren't sure, use the **pwd** command to find out the name of your working directory.

## Entering the program

You now have a directory for storing your two C programs.

1. Copy the first program from directory **text** to this directory:

```
$ cp ../text/metric.c
```

2. Start a new editing session:

□ Type ***vi*** and the name of the file and press RETURN to start:

```
$ vi metric.c
```

□ Display the file on the screen:

```
 1   /* Convert gallons to liters   */
 2   main()
 3   {
 4      int low, high, step;
 5      float gallons, liters;
 6
 7      low = 10; high = 20; step = 2;
 8
 9      printf("%4s \t %6s \n\n", "gals", "liters");
10
11      gallons = low;
12      while (gallons <= high)
13      {
14         liters = (gals * 3.785)  [Error on this line]
15         printf("%4.0f \t %6.2f\n", gals, ltrs);
16         gallons += step;
17      }
18   }
"metric.c" [New file] 18 lines, 378 characters
```

3. Compile the program without leaving **vi**:

□ Type ***:!cc metric.c*** and press RETURN:

```
:!cc metric.c
_
```

□ The C compiler will respond with these messages:

```
!cc metric.c
"metric.c", line 15: syntax error
"metric.c", line 15: illegal character: 134 (octal)
"metric.c", line 15: cannot recover from earlier
errors: goodbye!
!
_
```

## MAKING CORRECTIONS

Your job is to correct the error reported by the C compiler, recompile the program, and try running it. Although there are three error messages this time, there is only one error (a missing semicolon).

1. Return to **vi** to correct the error:

   □ Display the line with the error:

   ```
   liters = (gallons * 3.785)
   ```

   □ Add the missing semicolon and display the corrected line:

   ```
   liters = (gallons * 3.785);
   ```

   □ Save the correction:

   ```
   :w metric.c
   "metric.c" 18 lines, 379 characters
   ```

   This time, instead of allowing **cc** to send the output to the standard output file **a.out**, we're going to redirect the output to another file.

2. Recompile the program without leaving **vi**:

   □ Use the **-o** option to redirect the output to a file named **metric** and press (RETURN):

   ```
   :!cc -o metric metric.c
   _
   ```

   □ The C compiler will respond with another exclamation point, which tells you that the program has compiled successfully.

3. Now you are ready to run the program:

   □ Type ***:!metric*** and press (RETURN):

   ```
   :!metric
   gals   liters

   10     37.85
   12     45.42
   14     52.99
   16     60.56
   18     68.13
   20     65.70
   _
   ```

   □ The program works. Type ***q*** and press (RETURN) to leave **vi**.

## Modifying the program

As mentioned earlier, you can modify **metric.c** to obtain different results. One way to do this is to change the initializations on lines 7, 8, and 9. For example, by changing `low` to 11, `high` to 25, and `incr` to 1, you can obtain conversions for 11, 12, 13, . . . , 25 gallons:

```
low = 11; high = 25; step = 1;
```

## Entering another C program

1. Start a new editing session with **vi**:

□ Start an editing session with a new file called **compact.c**:

```
$ vi compact.c
```

□ Enter and save the following program (but *not* the line numbers).

```
/* Remove newlines */[Descriptive comment line]
#include <stdio.h>    [This is the standard I/O file for C]
main()                [Start of program]
{                     [Start of first block]
  int c, n=0, max=2;  [Declare c as an integer; declare n as an
                       integer and set it to zero; declare max
                       also and set it to 2]
  while ((c=getchar() != EOF)
                      [Read a character; stop if it's an end-of-
                       file character (^D)]
  {                   [Start of second block]
    if (c=='\n')      [Is the character newline?]
      n++;            [If so, add to the counter]
    else              [If not, . . .
      n=0;            reset the counter to zero]
    if (n<=max)       [Are there fewer than two newlines?]
      putchar(c);     [If so, output this character]
  }                   [End of second block]
}                     [End of first block]
(ESC) :w              [Don't type :q, stay in vi]
"compact.c" [New file] 15 lines, 192 characters
```

Before proceeding, we should include a short discussion of this program. It takes characters from the keyboard and counts the number of consecutive newlines encountered. Any time there are at least two, the program ignores all but the first two and goes on to the next character, thereby compacting the file. Line 6, as you have entered it, contains an error: a right parenthesis is missing. Because of this, error messages will result when the program is compiled, and these error messages will provide a few useful lessons. You will get a chance to correct the error in line 6 shortly.

2. Compile the program without leaving **vi**:

□ Use the temporary exit symbol (**:!**) in front of **cc** to have **cc** compile compact.c:

```
:!cc compact.c
```

□ The C compiler **cc** will respond with a message like this:

```
"compact.c", line 7: syntax error
"compact.c", line 15: syntax error
!
_
```

Error messages resulted on lines 7 and 15 when **cc** attempted to compile compact.c. Because of the missing parenthesis back on line 6, **cc** has run into trouble on lines 7 and 15, and compilation of compact.c is unsuccessful. The solution is to add the missing parenthesis and try **cc** again.

## Making corrections to the program

Now that you have entered the program and attempted a first compilation, you are ready to correct the error reported by the compiler and try a second compilation:

1. Return to your editing session and add the missing parenthesis:

□ Display line 6:

```
while ((c=getchar() != EOF)
```

□ Add the missing parenthesis and display the corrected line:

```
while ((c=getchar()) != EOF)
```

□ Store the corrected text:

```
:w
"compact.c" 15 lines, 193 characters
```

2. Recompile the program without leaving **vi**:

□ Use the temporary escape (**:!**) in front of **cc** again:

```
:!cc -o compact compact.c
          _
```

□ After **cc** has run, **vi** will respond with another exclamation point:

```
!cc -o compact compact.c [Place the output in compact]
!
_
```

This time the compilation is successful; there are no error messages. Again, unless you specify another file, **cc** places the output of any compilation in a file named **a.out**. This is the standard output for **cc**. However, we have directed output to **compact** by using the **-o** option with **cc**.

## Running the program

With **compact.c** successfully compiled, you are now ready to run the program. Note that **compact** is an executable file, which means that all you have to do to run your program is type ***a.out*** after the UNIX shell prompt. However, since you are still in an editing session with **vi**, you won't see the shell prompt this time.

1. Run the program:

   □ Copy file **spaces** from **text** to **c_progs**:

```
:!cp ../text/spaces spaces
!

_
```

   □ Use the temporary escape symbol (***:!***) again to run **compact**:

```
:!compact < spaces
_
```

   □ The output will appear on the screen:

```
This is line 1

This is line 2

This is line 3
!            [Conclusion of the program's output]

_
```

   This program illustrates one method for eliminating extra blank lines from a file.

2. End the editing session by typing ***:q***.

### OTHER PROGRAMMING LANGUAGES

Although C and UNIX have been closely related from the beginning, some of the other languages popular with microcomputers, such as FORTRAN and Pascal, have also become available with UNIX. The standard FORTRAN compiler, called **f77**, compiles source programs in files that end in .f. One widely-used Pascal compiler from the University of California, called **pc**, compiles source programs in files that end in .p. An assembler (**cc** again) assembles source programs in files that end in .s.

Both compilers and the assembler, like C, place the executable program produced by compilation in a.out if no other output file is named. All three compilers (C, FORTRAN, and Pascal) and the assembler have an **-o** option that allows you to direct the output to another file.

Finally, there is another language closely associated with UNIX that is covered later in this book. You have been dealing with the UNIX *shell* almost since you started reading this book. The shell is the command processor that takes each command you enter, makes sure that UNIX executes it for you, and then returns the results to you. However, the shell also offers a language that you can use to write programs called *shell procedures*, or *shell scripts*, which you can write to carry out simple tasks or to design a new UNIX interface. This aspect of the shell is discussed in Part V, "Shell Programming."

For some jobs you may prefer using shell procedures because they don't require compilation and storing an extra file. For other jobs you may prefer using C programs because they can handle character-processing better. For still other jobs you may prefer to use FORTRAN, Pascal, or some other language available on your system.

## 15.3 Summary

In this chapter you have learned how to execute UNIX commands without leaving **vi**. As examples of executing UNIX commands, you entered, compiled, corrected, recompiled, and ran two C programs. To interrupt editing with **vi** to execute another UNIX command, type a colon and an exclamation point (*: !*), the name of the command, and press (RETURN). After the other UNIX command has been executed, you will be able to return to **vi**, which will remain in the same state unchanged.

To produce a C program ready to run, you have to follow these steps:

1. Call up **vi** (or another text editor), enter the code, and store it in a file (which must have a .c suffix).

2. Use the **cc** compiler, sending the object code either to the standard output file a.out or to a file of your choice:

3. Since you are still in **vi**, you can easily make corrections to any errors reported by **cc** using the **s** (substitution) command.

4. To run your program, just type the name of the output file produced by **cc**.

FORTRAN is a standard language, while Pascal and other languages are also available on many UNIX systems. The FORTRAN compiler (**f77**) compiles source code stored in files that end in .f; one Pascal compiler (**pc**) compiles source code stored in files that end in .p; the assembler (**cc** again) assembles source code stored in files that end in .s. All of these, like **C**, allow you to direct output to another file with the **-o** option.

## For further reading

In this chapter you worked briefly with **C**, the language in which most of UNIX is written. If you would like to know more about **C**, you will need another book. It won't be possible to say much more about **C** in this book, so here are a few titles:

Hancock, Les and Morris Krieger, *The C Primer*, New York: McGraw-Hill, 1986. This contains thorough, comprehensive coverage of all aspects of C programming from basic to intermediate.

Kelley, Al and Ira Pohl, *A Book on C*, Menlo Park, CA: Benjamin/Cummings, 1984. Designed as a college-level textbook at the University of California, Santa Cruz, this book covers all aspects of C programming thoroughly. It includes many examples, diagrams, tables, and exercises. However, the reader is assumed to be familiar with some college-level mathematics.

Kernighan, Brian W. and Rob Pike, *The UNIX Programming Environment*, Englewood Cliffs, NJ: Prentice-Hall, 1984. This book, which seems to be intended for experienced users of UNIX and C, covers a lot of ground: the file system, the shell, filters, developing a variety of utility programs with the shell, C programming, and document preparation. This provides good reference material on many topics.

Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, 1978. This book provides excellent material, possibly the best there is, for experienced C programmers, but is practically incomprehensible to beginners. Start with one of the other books before trying this one.

Kochan, Stephen G., *Programming in C*, Hasbrouck Heights, NJ: Hayden Book Company, Inc., 1983. Over-all this is one of the best books on C for beginners. It develops most topics from the ground up, and uses only a moderate amount of high school-level mathematics. However, there are some inconsistencies: some elementary concepts are

discussed extensively, while more difficult concepts are passed over quickly.

Plum, Thomas, *Learning to Program in C*, Cardiff, NJ: Plum Hall, 1983. This is the first in a series of books on C programming from Plum Hall. While it's more expensive than other books mentioned, it is oversized, contains a blackjack program and includes quick reference cards.

Purdum, Jack, *C Programming Guide*, Indianapolis, IN: Que Corporation, 1983. Also aimed at beginners, this book covers a wide range of topics, though not always in depth. There is very little mathematics for the reader to contend with.

Swartz, Ray, *Doing Business with C*, Englewood Cliffs, NJ: Prentice-Hall, 1988. This book starts with simple programs, and builds them step-by-step into larger programs with all the necessary safeguards.

Swartz, Ray, *Introduction to C Programming* (videotape and workbook), Santa Cruz, CA: Berkeley/Decisions, 1986. The first videotape (two diskettes) takes three and a half hours to play, and comes with a workbook. The instructor assumes some programming experience, but develops each new topic very carefully. The course provides an excellent discussion of pointers and arrays, an area where many beginners stumble. A second videotape continues with intermediate topics, such as structures and unions.

Wortman, Leon A. and Thomas O. Sidebottom, *The C Programming Tutor*, New York: Brady, 1984. This book is divided into two parts: tutorial and sample programs. The tutorial covers the language briefly, then seven programs handle a variety of text-processing and administrative tasks for software developers.

# Part IV

# Text-Formatting

In Part II you learned how to perform text-editing with **vi** and **ex**. In Part IV you will learn how to format text. (Once again, recall that in UNIX, formatting is completely separate from entering text.) Formatting in UNIX involves using the **nroff** and **troff** tools — **nroff** for daisy-wheel printers, **troff** for laser printers and phototypesetters.

# 16

# Introduction to **mm**

## 16.1 Introduction to formatting

### THE UNIX FORMATTING TOOLS

As mentioned earlier, text editing and text formatting are two separate processes in UNIX. You enter text into a file with one of the editors (**ed** or **vi**), then you use another program to format the text for printing. New *word processing* programs, which combine editing and formatting as two different functions of a single program, are becoming available for UNIX every month.

There are two main formatting programs in UNIX, along with some auxiliary programs and pre-processors. The main formatting programs, both derived from an earlier "runoff" program from MIT, are called **nroff** ("en-roff," new runoff) and **troff** ("tee-roff," typesetting runoff). These two programs have many similarities, but the main difference is that **nroff** is intended for ordinary printers, while **troff** is intended for laser printers and typesetting equipment. Therefore, **troff** has to handle font changes and proportional spacing.

To aid beginners in using **nroff** and to provide standard sets of formats, people have designed various pre-defined formatting tools called *macro packages.* (The term "macro" here refers to larger requests that have been constructed from smaller primitive requests—"micros" if you will.) Some of the most well-known are called **ms**, **me**, and **mm**.

In addition, there are several pre-processors that handle specialized types of text: **tbl** for tables, and **eqn** and **neqn** for mathematical expressions, among others. Finally, there are utilities: **checkeq** to check usage in **eqn** and **neqn** and **deroff** to remove all formatting requests (**nroff, troff, tbl, eqn**, or **neqn**) from a file, among others. In this chapter we'll focus on **mm**, which supercedes **ms** in System V. Throughout Part IV, references to **nroff** actually refer to both **nroff** and **troff**.

### AN EXAMPLE OF FORMATTING

The **mm** macro package offers you a wide selection of pre-defined features for text formatting that you can activate and deactivate by placing *embedded commands* (or *requests*) in your file.

1. Prepare an input text file for formatting:

□ Move to subdirectory **text** and start an editing session with a file called **pfl.deal**.

□ Insert the embedded requests (but not the descriptions in brackets), and type this letter as shown here. We'll shorten the page length to make the output more compact.

```
.pl 33                        [Page length = 33 lines]
.SA 1                         [Justify the right margin]
.P 0                          [Left-justified paragraph]
Dear Mr. Madison:
.P 1                          [Indented paragraph]
The purpose of this letter is to confirm Monday's
agreement.  Tupelo gets "Porkchop" Peterson, you
get "Earthquake" Emerson, and we get "Rotunda"
Robinson. Here are the players' names, heights,
weights, and new teams:
.DS I N                       [Start of display]
Emerson, Ezekiel R.     6'5"    273     Rochester
Robinson, Charles F.    6'3"    287     Des Moines
Peterson, Paul N.       5'9"    178     Tupelo
.DE                           [End of display]
.P 1                          [Indented paragraph]
You will be getting one of the finest players in
the People's Football League.
.DS C N                       [Centered display]
Your Friend and Mine, Bill
.DE                           [End of display]
```

□ Store the text and return to the UNIX shell prompt.

2. Format the text on your screen with **nroff**:

□ Type an **nroff** command that includes the **-cm** option:

```
$ nroff -cm pfl.deal | pg
```

or

```
$ mm pfl.deal | pg
```

```
                              - 1 -

Dear Mr. Madison:

     The  purpose of  this letter  is  to  confirm
Monday's   agreement.   Tupelo   gets   "Porkchop"
Peterson, you get "Earthquake" Emerson, and we get
"Rotunda"  Robinson.  Here are the players' names,
heights, weights, and new teams:

   Emerson, Ezekiel R.     6'5"   273   Rochester
   Robinson, Charles F.    6'3"   287   Des Moines
   Peterson, Paul N.       5'9"   178   Tupelo

     You will be getting one of the finest players
in the People's Football League.

                Your Friend and Mine, Bill
```

□ The words of each paragraph have been joined together, the right margin has been *justified* (aligned), the tabular material have been indented, and the closing has been centered.

## USING THE **mm** MACRO PACKAGE

In formatting this letter, which gives you a brief introduction to **nroff** and **mm**, you used one **nroff** command:

`.pl 33` Set the page length to 33 lines (without this and the **more** command, the letter would sail off the top of the screen before you got a chance to look at it).

and six **mm** requests:

| | |
|---|---|
| **.SA 1** | Right-justified text |
| **.P 0** | Left-justified paragraph |
| **.P 1** | Indented paragraph |
| **.DS I N** | Start of display (text is indented, but otherwise left alone) |
| **.DS C N** | Start of display (centered, but otherwise left alone) |
| **.DE** | End of display |

Each embedded **mm** command (or *request*) consists of a period (or dot) in column 1 followed by one or two capital letters (this conveniently distinguishes them from **nroff** commands, which are lower case). For example, the **mm** request for a paragraph is *.P* (with or without a digit following). By placing (or *embedding*) this request directly above lines of text, you can have those lines formatted into a paragraph.

Some embedded requests also require additional information. For example, the start of display request *.DS* may be used either by itself or with one or more characters added (in the example above, we added `C N` to get `.DS C N` for centering without filling).

In the sections that follow, we'll describe these and other formatting requests in greater detail. To save space, we'll use miniature examples, showing the input text in the left and the printed result on the right. Full-sized examples work the same way—they just take up more room.

## 16.2 Forming paragraphs

### Block paragraph .P

To form a block paragraph, with all lines left-justified, precede the first line with the `.P` request:

```
 Input                         Output

.P
bbbbbbbbbbbbbbbbbbbbbbbbbbbb   bbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbb                   bbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbb        bbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbb.              bbbbbbbbbbbbb.
```

In the event that the default may have been changed from block to indented paragraphs in **nroff**, use `.P 0` to guarantee block paragraphs.

### Indented paragraph .P 1

To make the first line indented five spaces, precede the first line with the `.P 1` request.

```
 Input                         Output

.P 1
iiiiiiiiiiiiiiiiiiiiiiiiiiiii       iiiiiiiiiiiiiiiiiii
iiiiiiiiiiii                   iiiiiiiiiiiiiiiiiiiiii
iiiiiiiiiiiiiiiiiiiiiii        iiiiiiiiiiiiiiiiiiiiiii
iiiiiiiiiiiiiiii.              iiiiiiiiiiiii.
```

## 16.3 Forming lists

In **mm** a sequence of related items indented by the same amount is called a *list*. Each item in the list is preceded by a number, letter, bullet, hyphen, or word(s), as shown in this example:

- `French`

- `German`

- `Italian`

Here you see three names (French, German, Italian) presented as a list of items, each double-spaced and preceded by a bullet (•). With **mm** you can produce a list like this by embedding the following requests into the text:

```
.BL                [Begin a bullet list]
.LI
French             [The first item is French]

.LI
German             [The second item is German]
.LI
Italian            [The third item is Italian]
.LE                [This is the end of the list]
```

In this simple example, we used one-word items in the list; but the items in many lists will be entire paragraphs. In the sections that follow we'll discuss the different kinds of lists you can request in **mm**.

## SIMPLE LISTS

**.BL**
**.DL**
**.ML**

As shown in the example above, you can form a *bullet list* by using the `.BL` request:

```
Input                           Output
.P 1
iiiiiiiiiiiiiiiiiiiiiiii             iiiiiiiiiiiiiiiiiii
iiiiiiiiiiiii:                  iiiiiiiiiiiiiiii:
.BL
.LI                                  + aaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaa                 aaaaaaaaaaaa
aaaaaaa
.LI                                  + bbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbb                bbbbbbbbbbb
bbbbbb
.LE
```

You can form a *dash list* by using the `.DL` request. The result is the same as a bullet list, except that a hyphen is used in place of a bullet to precede each item:

```
Input                           Output

.P 1
iiiiiiiiiiiiiiiiiiiiiiii             iiiiiiiiiiiiiiiiiii
iiiiiiiiiiiii:                  iiiiiiiiiiiiiiii:
.DL
.LI                                  - aaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaa                 aaaaaaaaaaaaa
aaaaaaa
.LI                                  - bbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbb                bbbbbbbb
bbbbbb
.LE
```

You can form a *mark list* by using the .ML request, followed by one or more characters to precede each item. In the following example, an asterisk (*) is used:

**Input**

```
.P 1
iiiiiiiiiiiiiiiiiiiiiiii
iiiiiiiiiiiii:
.ML *
.LI
aaaaaaaaaaaaaaaaaaaaaa
aaaaaaa
.LI
bbbbbbbbbbbbbbbbbbbbbbb
bbbbbb
.LE
```

**Output**

```
     iiiiiiiiiiiiiiiiiii
iiiiiiiiiiiiiiii:

     * aaaaaaaaaaaaaaaa
       aaaaaaaaaaaaa

     * bbbbbbbbbbbbbbb
       bbbbbbbbbbbb
```

## Reference lists .RL

To form a *reference list*, as is customary at the end of a book to indicate sources for quotations, use the .RL request. The items will be numbered automatically for you, with the numbers placed between brackets:

**Input**

```
.P 1
iiiiiiiiiiiiiiiiiiiiiiii
iiiiiiiiiiiii:
.RL
.LI
aaaaaaaaaaaaaaaaaaaaaa
aaaaaaa
.LI
bbbbbbbbbbbbbbbbbbbbbbb
bbbbbb
.LE
```

**Output**

```
     iiiiiiiiiiiiiiiiiii
iiiiiiiiiiiiiiii:

     [1] aaaaaaaaaaaaaaaa
         aaaaaaaaaaaaa

     [2] bbbbbbbbbbbbbbb
         bbbbbbbbbbbb
```

## Variable-item lists .VL

To form a *variable-item list* for a series of definitions or explanations, use the .VL request. You must provide a number after .VL to indicate how much space **mm** should leave between the current left margin and the start of text for each item in the list. In the following example, we leave ten columns:

**Input**

```
.P 1
iiiiiiiiiiiiiiiiiiiiiiii
iiiiiiiiiiiii:
```

**Output**

```
     iiiiiiiiiiiiiiiiiii
iiiiiiiiiiiiiiii:
```

```
.VL 10

.LI cattle                          cattle ccccccccccccc
cccccccccccccccccccccc                     cccccccccccc
ccccccc
.LI hogs                            hogs   hhhhhhhhhhh
hhhhhhhhhhhhhhhhhhhhhhh                    hhhhhhhhhhhhh
hhhhhhhhhhhh                               hhhhhh
.LE
```

## AUTOMATICALLY-NUMBERED LISTS **.AL**

To form an *automatically-numbered list* (or *alphabetic list*) for an outline, use the `.VL` request. If you don't type anything after `.AL`, **mm** will number the list with Arabic numbers. As you can see in the following example, the result is nearly the same as a reference list:

```
Input                          Output

.P 1
iiiiiiiiiiiiiiiiiiiiiii             iiiiiiiiiiiiiiiiiii
iiiiiiiiiiiii:                 iiiiiiiiiiiiiiii:
.AL
.LI                                 1. aaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaa                 aaaaaaaaaaaaa
aaaaaaa
.LI                                 2. bbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbb                bbbbbbbbbbbb
bbbbbb
.LE
```

By typing one of the following characters after `.AL` you can instruct **mm** to use either letters of the alphabet or Roman numerals:

| | | | |
|---|---|---|---|
| A | Uppercase letters | I | Uppercase Roman numerals |
| a | Lowercase letters | i | Lowercase Roman numerals |

Here is an example with uppercase letters:

```
Input                          Output

.P 1
pppppppppppppppppppppppp            ppppppppppppppppppp
ppppppppppppp:                 pppppppppppppppp:
.AL A
.LI                                 A. aaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaa                 aaaaaaaaaaaaa
aaaaaaa
.LI                                 B. bbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbb                bbbbbbbbbbbb
bbbbbb
.LE
```

Here is an example with lowercase Roman numberals:

```
 Input                          Output

.P 1
rrrrrrrrrrrrrrrrrrrrrrrr            rrrrrrrrrrrrrrrrrrr
rrrrrrrrrrrrr:                  rrrrrrrrrrrrrrrr:
.AL i
.LI                                 i.   aaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaa                   aaaaaaaaaaaaa
aaaaaaa
.LI                                 ii.  bbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbb                  bbbbbbbbbbbb
bbbbbb
.LE
```

To produce an outline, keep in mind that automatic numbering remains in effect for a given level until **mm** encounters a `.LE` request. Here is an example:

```
 Input                          Output

.AL A      [Start of "ABC" list]
.LI                             A. xxxxxxxxx
xxxxxxxxxx
.AL 1      [Start of "123" list]   1. aaaaaaaaaaaaaaaaaa
.LI                                   aaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaa
.LI                                2. bbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbb               bbbbbbb
.AL a      [Start of "abc" list]
.LI                                   a. mmmmmmmmmmmmmmmm
mmmmmmmmmmmmmm
.LI                                   b. nnnnnnnnnnnnnnnn
nnnnnnnnnnnnnnnnn
.LE        [End of "abc" list]     3. cccccccccccccccccc
.LI                                   ccccccc
ccccccccccccccccccccccccc
.LE        [End of "123" list]  B. yyyyyyyyy
.LI
yyyyyyyyy                       C. zzzzzzzzzzzz
.LI
zzzzzzzzzzzzzz
.LE        [End of "ABC" list]
```

## 16.4 Displaying text

### Static display

**.DS**
**.DE**

To display text (that is, to have it set apart from ordinary text), precede the first line with the `.DS` request and follow the last line with the `.DE` request. The result will be a *static display*, which will appear either on the current page (if space permits) or on the following page. A little later we'll discuss another type of display.

### Indented display **.DS I**

Although a display can be left left-justified with ordinary text by using the `.DS` request by itself, it often helps to make the display stand apart. In the following display, we indent the text five spaces by adding the `I` (indent) option to the `.DS` request:

**Input** **Output**

```
.P 1
ssssssssssssssssssssss              ssssssssssssssssss
ssssssssssss                   sssssssssssssssssssssss
sssssssssssssssssssssss.       ssssssssssss.
.DS I
ddddddddddddddddd,                  ddddddddddddddddd,
ddddddddddddddd.                    ddddddddddddddd.
ddddddddddddddddd,                  ddddddddddddddddd,
ddddddddddddd.                      ddddddddddddd.
.DE
```

### Double-indented display **.DS I F n**

To have a block of text indented from both margins, add the `I` (indent) option, the `F` (fill) option, and a number to indicate the indentation from the right margin, as in this example:

**Input** **Output**

```
.P 1
ssssssssssssssssssssss              ssssssssssssssssss
ssssssssssss                   sssssssssssssssssssssss
sssssssssssssssssssssss.       ssssssssssss.
.DS I F 5
ddddddddddddddddd                   ddddddddddddd
ddddddddddddddd                     dddddddddddddd
ddddddddddddddddd                   dddddddddddd
ddddddddddddd.                      ddddddddddddd
.DE                                 ddddddddd.
```

### Blocked display **.DS CB**

Another style is to center the text, with left-justification. To have text indented and left-justified, but centered when displayed, add the `CB` (center block) option to the `.DS` request:

**Input** **Output**

```
.P 1
sssssssssssssssssssssssssss         sssssssssssssssssss
ssssssssssss                   ssssssssssssssssssssssss
sssssssssssssssssssss.         sssssssssssss.
.DS CB
dddddddddddddddddd,               dddddddddddddddddd,
dddddddddddddddd.                 dddddddddddddddd.
dddddddddddddddddd,               dddddddddddddddddd,
dddddddddddddd.                   dddddddddddddd.
.DE
```

### Centered display **.DS C**

Sometimes it's attractive to show a group of lines centered horizontally, as in a title or a beautiful verse. To have each individual line of text centered when displayed, add the **C**option to the `.DS` request.

**Input** **Output**

```
.P
sssssssssssssssssssssssss           sssssssssssssssssss
ssssssssssss                   ssssssssssssssssssssssss
sssssssssssssssssss.           sssssssssssss.
.DS C
dddddddddddddddddddd,            dddddddddddddddddddd,
dddddddddddddddd.                  dddddddddddddddd.
dddddddddddddddddd,               dddddddddddddddddd,
dddddddddddd.                        dddddddddddd.
.DE
```

## 16.5 Emphasizing and de-emphasizing text

### Emphasizing text **.I** **.B** **.R**

If you would like to emphasize text in a paragraph, you can underscore it by inserting the `.I` (italic) request. One way to do this is to place .I on a separate line, followed by the text to be emphasized. (Since **mm** will underscore only one word here, we must type an underscore character between the two words to make them appear like one to **mm**.) Although this may appear to break up the paragraph, **mm** will fill the paragraph without interruption.

```
Input                           Output

.P 1
Now we are coming to the             Now we are coming
.I most important              to the most important
item of the agenda.            item of the agenda.
```

Another way to accomplish about the same thing is to bracket the text between one `.I` (italic) and one `.R` (roman) request on separate lines. With this method, any number of words can be underscored. Once again, **mm** will reconstruct the paragraph for you.

```
Input                           Output

.P 1
Now we are coming to the             Now we are coming
.I                             to the most important
most important                 item of the agenda.
.R
item of the agenda.
```

The `.B` (bold) request works just like `.I`, as shown in these examples:

```
Input                           Output

.P 1
Now we are coming to the             Now we are coming
.B most                        to the most urgent
urgent item on the agenda.     item on the agenda.

.P 1
Now we are coming to the             Now we are coming
.B                             to the most urgent
most urgent                    item on the agenda.
.R
item on the agenda.
```

### De-emphasizing text

The best way to de-emphasize text in your document is to leave it out. No formatting request is required.

## 16.6 Other features

### Entering accent marks

You can enter any of seven accent marks for text in languages other than English by using a three-character string. These are illustrated in the example that follows:

```
Input                              Output

.P 1
Franc\*,oise a dit, "J'ai          Françoise a dit,
vu un des arbres ce matin.         "J'ai vu un des arbres ce
Voila\*` la fore\*^t               matin.  Voilà la forêt
la\*`-bas."                        là-bas."
.P 1
"Queremos escuchar las                  "Queremos escuchar
canciones de los                   las canciones de los
pa\*'jaros por la                  pájaros por la mañana,"
man\*~ana," dijo Javier.           dijo Javier.
.P 1
"U\*;brigens," sagte Erik,            "Übrigens," sagte
"die Vo\*:gel erfreuen             Erik, "die Vögel erfreuen
sich an einer scho\*:nen           sich an einer schönen
Aussicht auf den Wald."            Aussicht auf den Wald."
```

Here is a summary of the accent marks available in **mm**:

| | | | |
|---|---|---|---|
| Grave accent | *` | Cedilla | *, |
| Acute accent | *' | Umlaut for uppercase | *; |
| Circumflex accent | *^ | Umlaut for lowercase | *: |
| Tilde | *~ | | |

## Justifying text

**.SA 1**
**.SA 0**

Ordinarily, **mm** does not justify the right margin when printing text. To produce justification, use this request:

```
Input                              Output

.SA 1
jjjjjjjjjjjjjjjjjj                 jjjjjjjjjjjjjjjjjjjj
jjjjjjjjjjjjjjjjjjjjj              jjjjjjjjjjjjjjjjjjjj
jjjjjjj.                           jjjjjjjjj.
```

To turn justification off again, use the following:

```
Input                              Output

.SA 0
uuuuuuuuuuuuuuuuuu                 uuuuuuuuuuuuuuuuuu
uuuuuuuuuuuuuuuuuuuuu              uuuuuuuuuuuuuuuuuuuu
uuuuuuuu.                          uuuuuuuu.
```

## Skipping lines

**.SP** *n*

To leave a blank lines on a page, use the .SP request, followed by the number of blank lines desired, as shown here:

**Input**

```
rrrrrrrrrrrrrrrrrrrr
rrrrrrrrrrr.
.SP 5
ssssssssssssssssssss
sssssssss.
```

**Output**

```
rrrrrrrrrrrrrrrrr
rrrrrrrrr.





sssssssssssssssssss
sssssss.
```

## Changing point size

The size of printed characters is measured in small units called *points.* Since each point is 1/72 of an inch, 72 points equal one inch, 36 points equal half an inch, 18 points a quarter of an inch, 9 points an eighth of an inch, and so on. The size of any given typeface is measured from the top of an uppercase letter to the bottom of a lowercase p.

The vertical spacing between lines of text is the distance, measured in points, from bottom of one line to the bottom of the next. For the sake of legibility, vertical spacing is generally about 20% larger than the point size of the typeface. (The difference between the vertical spacing and the point size is called the *leading.* This term relates to the practice of placing lead spacers between lines of hand-set cold type.)

The default for **troff** is to print 10-point characters with 12-point vertical spacing between lines. When you're formatting **mm** requests with **troff**, you can change the point size and the vertical spacing with the `.S` request. This request accepts either one or two arguments, allowing you to change point size or vertical spacing or both at the same time. In the following example, we'll increase the point size and vertical spacing for the title, then restore them for the main text.

**Input**

```
.DS C
.S 18 22
The Final Test
.S -4 -5
By James Houston
.DS E
.S D
.P
The forces of change are at
work in society, bringing
people face to face with new
```

**Output**

# The Final Test

## By James Houston

```
The forces of change are at work
in society, bringing people face
to face with new concepts and
different challenges.
```

As illustrated in the example above, you can use relative numeric values (+2, -3), absolute numeric values (14, 9), or alphabetic values (see below). The alphabet values allow you to make changes more easily and quickly.

If you enter `.S` by itself, without any arguments, it means to restore the previous settings. Here are the alphabetic values:

| | |
|---|---|
| **P** | Previous settings (same as no argument) |
| **D** | Default settings |
| **C** | Current settings |

## 16.7 Summary

In this chapter you learned how to format text with the **nroff** command, focusing primarily on the **mm** macro package.

### Introduction to formatting

The main formatting programs in UNIX are **nroff** (for printing) and **troff** (for typesetting). These are accompanied by *macro packages* (which provide standard, predefined formatting requests) and *preprocessors* (which handle specialized types of text). Macro packages, such as **ms**, **me**, and **mm**, are activated via option switches in an **nroff** command string. Preprocessors, such as **tbl** (for tables), **eqn** and **neqn** (for mathematical expressions), are activated by separate commands that send text to **nroff** via a pipeline.

To use the **mm** macro package, place embedded requests in the text to be formatted, then execute an **nroff** command that includes the **-cm** option and the name of the file to be formatted. To obtain printed results, send the text to the **lp** print spooler via a pipeline.

### Forming paragraphs and lists

- *Paragraphs*—The **mm** embedded requests for forming paragraphs are as follows. No terminator is required.

| | |
|---|---|
| Block paragraph | `.P 0` |
| Indented paragraph | `.P 1` |

- *Lists*—The **mm** embedded requests for forming lists are as follows. Precede each list with one request, begin each item in the list with `.LI`, then terminate the list with another request (`.LE`).

| | | |
|---|---|---|
| Bullet list | `.BL` | (`.LE`) |
| Dash list | `.DL` | (`.LE`) |
| Marked list | `.ML` *mark* | (`.LE`) |
| Reference list | `.RL` | (`.LE`) |
| | ( `1` ) 1 2 3 (default) | |
| | ( `A` ) A B C | |
| Auto-numbered list | `.AL` { `a` } a b c | (`.LE`) |
| | ( `I` ) I II III | |
| | ( `i` ) i ii iii | |
| Variable-item list | `.VL` *n* | (`.LE`) |

## Displaying text

The `.DS` request allows you to display text in various formats. Precede the text to be displayed with one request, and terminate it with another (`.DE`).

| | | |
|---|---|---|
| Standard Display | `.DS` | (`.DE`) |
| Indented Display | `.DS I` | (`.DE`) |
| Blocked Display | `.DS CB` | (`.DE`) |
| Centered Display | `.DS C` | (`.DE`) |

## Emphasizing and De-emphasizing Text

The **mm** `.I` and `.B` requests produce underscoring and bold for emphasis; use the `.R` (roman) request to terminate underscoring and bold:

| | | |
|---|---|---|
| Underscoring | `.I` | (`.R`) |
| Bold | `.B` | (`.R`) |
| De-emphasis | (None required) | |

## Entering accent marks

You can use the following strings to enter accent marks for languages other than English:

| | | | |
|---|---|---|---|
| Grave accent | `\*`` | Cedilla | `*,` |
| Acute accent | `\*'` | Umlaut for uppercase | `\*;` |
| Circumflex accent | `\*^` | Umlaut for lowercase | `\*:` |
| Tilde | `\*~` | | |

## Justifying the right margin

The **mm** `.SA` request allows you to turn justification on and off:

Justifying `.SA 1` (`.SA 0`)

## SKIPPING LINES

The `.SP` *n* request instructs **mm** to leave *n* blank lines before printing the next line of text.

## CHANGING THE POINT SIZE

The `.S` request allows you to change the point size and the vertical spacing on the same line.

# 17

# Formatting with **mm**

In this chapter we'll continue our discussion of formatting with **mm**, focusing on features that relate to entire pages of text.

## 17.1 Keeping lines of text together

### Static display

**.DS**
**.DE**

If **mm** doesn't have room for a static display on the current page, it will move the entire display to the following page. In the following example we have a paragraph that is five lines long, but there are only three lines available for it at the bottom of page 7. So **mm** moves the entire paragraph to page 8, leaving a gap at the bottom of page 7.

```
Input                            Output
bbbbbbbbbbbbbbbbbbbbbb           bbbbbbbbbbbbbbbbbbbbbbbb
bbbbbb.                          bbbbbbbbbbb.
.DS
sssssssssssssssssssssssss
ssssssssssss                                                  7
sssssssssssssssssss.          -----------
ssssssssssssssss                      sssssssssssssssssss     8
sssssssssssssssssssss            ssssssssssssssssssssssss
ssssssssssss.                    sssssssssssss.  ssssssss
.DE                              ssssssssssssssssssssssss
                                 sssssssssssssssssss.
```

### Floating display

**.DF**
**.DE**

A static display performs a valuable service, but some people object to the blank space it sometimes leaves at the bottom of a page. One way to avoid this is to use a *floating display.* The block of text is still moved intact to the following page, as with a standard "keep." However, **nroff** will use the text that follows the block to fill the blank lines at the bottom of the previous page, as in this example:

```
Input                           Output

.P 1
ssssssssssssssssss                   sssssssssssssssssss
sssssssss                       sssssssssssssssssssssss.
ssssssss
ssss.                           bbbbbbbbbbbbbbbbbbbbbbbb
.DF I N 5                       bbbbbbbbbbbbbbbbbbbbbbbb     7
qqqqqqqqqqqqq                 ----------
qqqqqqqqqqq                                                  8
qqqqqqqqqqqqqqqqqqqqq.               qqqqqqqqqqqqqq
.DE                                  qqqqqqqqqqqqqq
.P 0                                 qqqqqqqqqqq.
bbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbb             bbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbb         bbbbbbbbbbbb.
bbbbbbbbbbbbbbbbbbbb.
```

Since page 7 doesn't have room for the three lines of the quotation, the quotation is moved to page 8, leaving two blank lines on page 7. Two lines of the block paragraph are then moved ahead of the quotation to the bottom of page 7. With two lines of the block paragraph moved to page 7, this leaves two more lines to follow the quotation on page 8.

## 17.2 Using footnotes

### Setting up a footnote .FS .FE

If you're typing a document that requires footnotes, **nroff** and **mm** can save you a lot of work. All you have to do is type the footnote right after the location of the reference to it (surrounded by the footnote requests `.FS` and `.FE`), and **mm** will take care of making room for the note at the bottom of the page, and will also print a separator between the footnote and the last line of main text. Here is an example:

```
Input                              Output

.P 1
"I have a dream."*                      "I have a dream."*
.FS *
* Dr. Martin Luther King                     In  a public address
.FE                                 given  yesterday in Wash-
.P 1                                ington,  D.C.,  the  Rev.
In a public address given           Martin  Luther King  told
yesterday in Washington, D.C.,
the Rev. Dr. Martin Luther...       --------
                                    * Dr. Martin Luther King
```

### Using numbers with footnotes

If you are planning to refer to your footnotes by number, you can request numbers in place of asterisks by using the `\*F` (automatic-numbering) request instead of an asterisk. Here is an example of automatic numbering:

```
Input                                Output

.P 1
sssssssssssssss                         sssssssssssssssssss
sssssss.\*F                          sss.7  ssssssssssssssss
.FS                                  sssssssssssssssssssssss
ffffffffffffffffff                   sssssssssssss.8
.FE
sssssssssssssssssss                     sssssssssssssssssss
sssssssssss                          sssssssssssssssssssssss
sssssssssssss.\*F                    sssssssssss.
.FS
ggggggggggggggggggg                  ---------
.FE                                    7. fffffffffff
.P 1                                      fffffff.
sssssssssssssssssssss
sssssssssssss                          8. ggggggggggg
sssssssssssssssssss.                      ggggggg.
```

For **nroff**, the default format for each footnote is as follows: no hyphenation within the footnote, no adjustment of text, indentation of footnote text, right justification of the numbers, and one blank line separating one footnote from another.

### Controlling the format **.FD**

If you'd like to use a format other than the **nroff** default, you can add the `.FD` request, followed by a number code from 0 to 11, to produce one of the twelve options shown in Table 17.1.

## 17.3 Using headings

In this section you will learn how to use both unnumbered and numbered section headings.

### Unnumbered headings **.HU**

To produce a plain, unnumbered heading of less than one full line, precede it with the `.HU` (heading unnumbered) command, which by default produces a second-order heading.

TABLE 17.1. Formatting Options for Footnotes

| Code | Hyphenate? | Adjust? | Indent<br>the text? | Justify<br>the number |
|---|---|---|---|---|
| 0 | No | Yes | Yes | Left |
| 1 | Yes | Yes | Yes | Left |
| 2 | No | No | Yes | Left |
| 3 | Yes | No | Yes | Left |
| 4 | No | Yes | No | Left |
| 5 | Yes | Yes | No | Left |
| 6 | No | No | No | Left |
| 7 | Yes | No | No | Left |
| 8 | No | Yes | Yes | Right |
| 9 | Yes | Yes | Yes | Right |
| 10 | No | No | Yes | Right (**nroff** default) |
| 11 | Yes | No | Yes | Right |

**Input**

```
.HU "Introduction"

iiiiiiiiiiiiiiiiiiiiiiiii
iiiiiiiiiiii
.HU "Background"
bbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbb.
.HU "Basic Facts"
fffffffffffffffffffffffff
```

**Output**

```
Introduction

iiiiiiiiiiiiiiiiiiiiiiii
  iiiiiiiiiiiii.


Background

bbbbbbbbbbbbbbbbbbbbbbb
bbbbbbb.

Basic Facts

fffffffffffffffffffffff.
```

## NUMBERED HEADINGS .H

Numbered headings work like unnumbered headings, except that each heading is preceded by a level number. When you use the `.H` (numbered heading) request, leave a space after `.H` and type a number to indicate the level of subordination (up to seven levels again). You can leave it up to **mm** to assign the next available number at any given level, and **mm** can generate a table of contents from your headings automatically, as shown in the next section.

**Input**

```
.H 1 "THE MAIN THING"
This is a description of
the main thing.
.H 2 "Where to Start"
```

**Output**

```
1.    THE MAIN THING

This is a description of
the main thing.
```

```
This section tells you
where to start.                 1.1  Where to Start

.H 2 "When to Start"
This section helps you          This section tells you
decide when to start.           where to start.
.H 3 "The Best Time"
In this subsection you          1.2   When to Start
will learn the best time.
.H 3 "The Worst Time"           This section helps you
In this subsection you          decide when to start.
will learn the worst time.
.H 1 "THE NEXT THING"           1.2.1  The Best Time  In
This is a description of        this subsection you will learn
the next thing.                 the best time.


                                1.2.2  The Worst Time  In
                                this subsection you will learn
                                the best time.


                                2.    THE NEXT THING

                                This is a description of
                                the next thing.
```

To summarize, the default formats for first-, second-, and third-order are as follows:

**Level 1** —Two blank lines above, one blank line below, all letters capitalized and bold

**Level 2** —One blank line above and below, initial letters capitalized, all letters bold

**Level 3–7** —One blank line above, two *spaces* following (run-in style), with underscoring

## PRODUCING A TABLE OF CONTENTS .TC

To produce a table of contents, place a `.TC` request near the end of the document, following all text. Then **mm** will use all first- and second-order headings for the entries, and print "CONTENTS" near the top of the page. Given the numbered headings above (assuming page 1), `.TC` would produce this:

```
                         CONTENTS

THE MAIN THING ................... 1
Where to Start ................... 1
When to Start .................... 1
THE NEXT THING ................... 1
```

## 17.4 Page layout

### Standard layout

Ordinarily, **nroff** lays out a page as shown in Figure 17.1. In the standard **mm** page layout, printing appears in a 6 by 9 inch area on each 8-1/2 by 11 inch page, with the page number centered at the top between hyphens. The 6 by 9 inch area includes

- Six inches of text across (65 columns)
- Nine inches of text down (54 lines) (66 total lines per page)

### Changing the date **.ND**

If you want to change the date, place the `.ND` (new date) request near the beginning of the text file, typing the desired date after `.ND`, as shown here:

```
.ND September 15, 1986
```

To print the date with **mm**, you can use the `DT` (date) string, which is indicated in an **mm** request by the notation `\\\\*(DT`.

This section discusses how you can change the standard page layout.

### Changing the header **.PH** **.EH** **.OH**

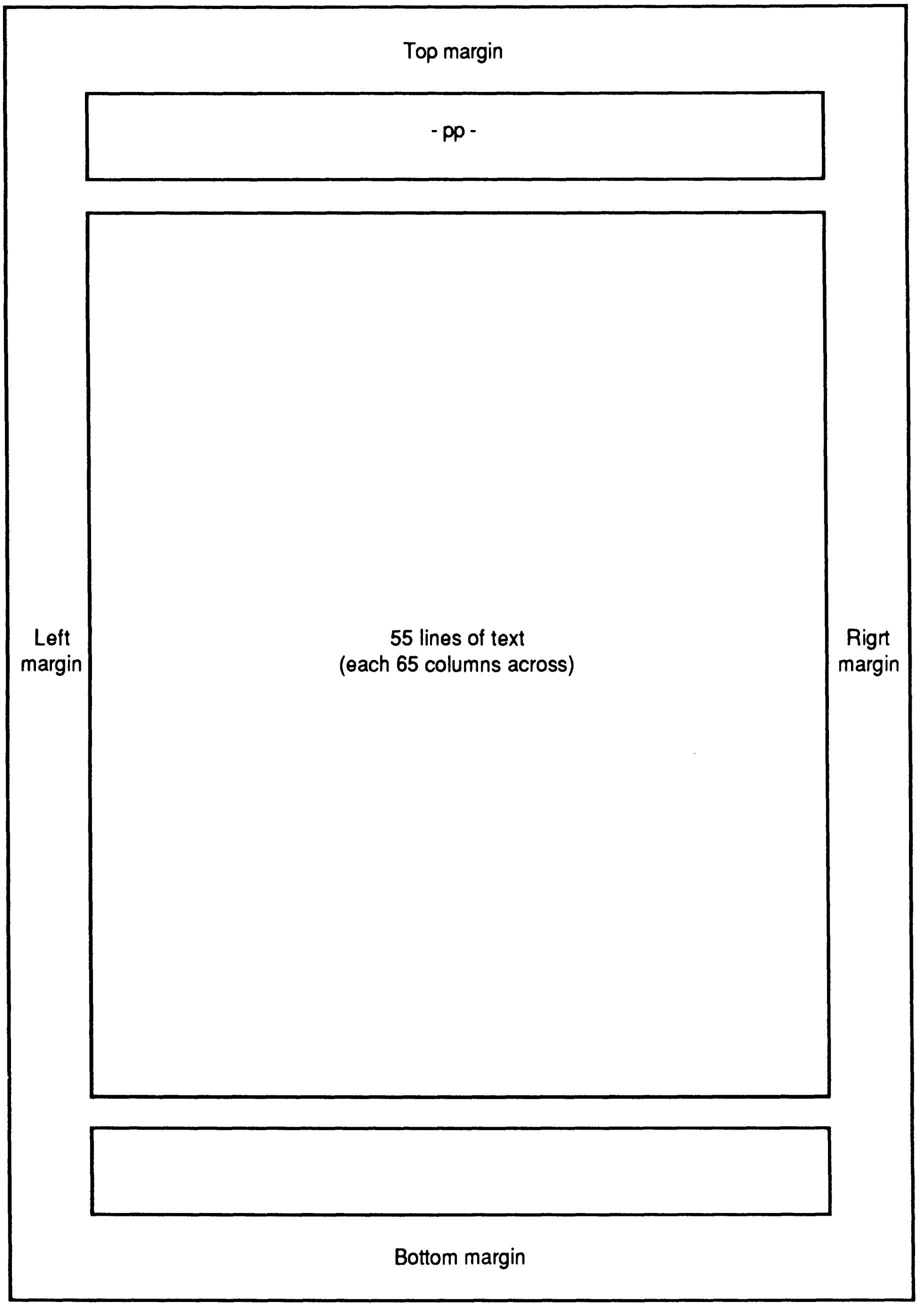
The *header*, an extra block above the body of text at the top of the page, usually remains fixed throughout the document. In **mm**, you are allowed two lines for every page, using the `.PH` (page header) request. You are also allowed one line for even-numbered pages, using `.EH` (even-page header) and one for odd-numbered pages, using `.OH` (odd-page header). You can place text in any of three locations on these lines: the left, the middle, or the right. (By default, the page number occupies the middle position on every page.)

To add text to the header, use one of these requests, using apostrophes to indicate placement of the text, as shown here:

```
.PH "'left'center'right'"    [Specify a header for all pages]
.EH "'left'center'right'"    [Specify a header for even pages]
.OH "'left'center'right'"    [Specify a header for odd pages]
```

For example, you could use this request to place the heading that follows on every page (omitting the page number in the middle):

FIGURE 17.1. Standard **mm** page layout.



```
.PH "'Draft Copy″Revision 3'"
```

```
Draft Copy                                    Revision 3
```

To place today's date at the top of every page, centered (as shown below), you could use the following request:

```
.PH "″\\\\*(DT″"
```

```
                    September 15, 1986
```

## Changing the footer

**.PF**
**.EF**
**.OF**

The *footer* is an extra line below the body of text at the bottom of the page. As with the header, you can place your own message in any of three locations on this line: the left, the middle, or the right.

You have three requests for footers analogous to the three for headers:

```
.PF "'left'center'right'"     [Specify a footer for all pages]
.EF "'left'center'right'"     [Specify a footer for even pages]
.OF "'left'center'right'"     [Specify a footer for odd pages]
```

For example, you could use this request to alternate the following two footers on even and odd pages, respectively:

```
.EF "'Confidential″Internal Use Only'"
.OF "″Project 432″"
```

```
Confidential                          Internal Use Only

                    Project 432
```

To place the date at the bottom of every page, centered (as shown below), you could use the following request:

```
.PF "″\\\\*(DT″" [Nothing left, date center, nothing right, all pages]
```

```
                    September 15, 1986
```

### Moving the page number **nP**

Although **nroff** places the page number in the middle of the header by default, you are free to place it somewhere else and in a different format. To change the position of the page number, you can use the P register, which is indicated in an **mm** request by the notation \\\\nP. For example, to place the page number in the header next to the outer margin (the left side for even-numbered pages, the right side for odd-numbered pages), preceded by the word Page, you could use the following pair of requests:

```
.EH "'Page \\\\nP″'"     [Page number left, nothing center or right]
.OH "″'Page \\\\nP'"     [Nothing left or center, page number right]
```

To produce the same result at the bottom of the page, you could use the following pair of requests:

```
.EF "'Page \\\\nP″'"     [Page number left, nothing center or right]
.OF "″'Page \\\\nP'"     [Nothing left or center, page number right]
```

### Skipping pages **.SK** **.OP**

To have **mm** skip to the top of the next page, leaving the bottom of the current page blank, use this:

```
.SK                [Skip to the top of the next page]
```

To have **mm** skip to the top of the next *odd-numbered* page, leaving blank space on at least one page, use this:

```
.OP                [Skip to the top of the next odd-numbered page]
```

If the current page number is even, .OP is equivalent to .SK; if it is odd, .OP will leave the following even-numbered page blank. It is a common practice to begin new sections or chapters on an odd-numbered (front) page.

## 17.5 Initiating formatting

One advantage of using **mm** in System V is that it offers simplified command lines to initiate formatting. We'll summarize **mm** options and ways of invoking them here, even though some of the features are described in Chapter 19, "More on Formatting." For example, the standard **nroff** command line for initiating formatting on a file called **sec.5** that contained tables and equations, using a plain printer, would look like this:

```
$ tbl sec.5 | neqn /usr/pub/eqnchar - \
> | nroff -Tlp -cm | col | lp
```

You could accomplish the same thing with this **mm** command line:

```
$ mm -t -e -c sec.5 | lp
```

## THE OPTIONS FOR **mm**

Here are the options available for **mm**:

| Option | Function |
|---|---|
| -E | Print with equal spacing (equivalent to **nroff -e**) |
| -t | Process tabular material with the **tbl** preprocessor, which is described in Chapter 19 (**-t** must precede **-e** when both are used, and may require **-c**, depending on the printer) |
| -e | Process equations with the **neqn** preprocessor, which is described in Chapter 19 (**-e** must follow **-t** when both are used, and may require **-c**, depending on the printer) |
| -c | Process double-column text, tables, or equations for printers that are not capable of reverse paper motion, using the **col** postprocessor, which is described in Chapter 19 but still allowed) |
| -T | Name the printer that will receive the text by appending characters from the partial list that follows:<br>-Tlp Ordinary line printer without reverse paper motion or partial line motions<br>-Tx EBCDIC line printer<br>-T37 Teletype Model 37<br>-T40/4 Teletype Model 40/4 (**-c** included)<br>-T43 Teletype Model 43 (**-c** included)<br>-T300 DASI 300<br>*-T450 DASI 450 (the default)*<br>-T2631 Hewlett-Packard 2631 (**-c** included)<br>-Thp Hewlett-Packard 264x (**-c** included)<br>-T382 DTC-382<br>-T4000a Trendata 4000a |

## THE DEFAULT AND ALTERNATIVES

The default selection for **mm**, in the absence of an explicit **-T** option, is the DASI 450, which has the following characteristics:

- Pitch—10 characters per inch (pica)—use **-12** to request elite
- Lines per inch—6

- Offset for left margin—0.75 inch—use **-rO** to change
- Line length—60 characters (6 inches)—use **-rW** to change

In other words, either of the following command lines imply that the printer receiving the text will have these characteristics:

```
$ mm sec.5              or             $ mm -T450 sec.5
```

If your printer requires it, you can change the pitch, the offset, and the line length within the command line by adding options. For example, suppose your printer uses 12 characters per inch (elite), and you would like to change the line length to 79 characters, with an offset of 12 characters. Then you could enter a command line like this:

```
$ mm -12 -rW79 -rO12 sec.5
```

## More examples

Here are more examples of **mm** command lines:

| | |
|---|---|
| `$ mm -12 -Tlp` | [Print on a plain line printer at 12 characters per inch (elite)] |
| `$ mm -E -T2631` | [Print with even spacing on a Hewlett-Packard 2631] |
| `$ mm -t -c -T4000a` | [Print text that includes tables on a Trendata 4000a—**c** required] |
| `$ mm -t -e -T43` | [Print text that includes both tables and equations on a Teletype Model 43—**c** not required] |

The general format of an **mm** command line for text that does or does not include tables and equations is summarized in Figure 17.2.

FIGURE 17.2. General formats for **mm** command lines.

| | **No Equations** |
|---|---|
| No Tables | $ ***mm*** *[options] file(s)* |
| Tables | $ ***mm -t*** *[options*] file(s)* |
| | **Equations** |
| No Tables | $ ***mm -e [options*] file(s)*** |
| Tables | $ ***mm -t -e [options*] file(s)*** |

* **-c** may be required, depending on the printer.

# 17.6 Summary

In this chapter you learned more about formatting with **mm**.

## FORMATTING TEXT

The **mm** embedded requests for keeping lines of text together are as follows. Precede the text with one request, and follow it with another (`.DE`).

| | | |
|---|---|---|
| Standard display | `.DS` | (`.DE`) |
| Floating display | `.DF` | (`.DE`) |

The **mm** embedded commands for printing footnotes at the bottom of the page are as follows. Precede the text of the footnote with the `.FS` command, and follow it with the `.FE` command.

| | | |
|---|---|---|
| Setting Up a Footnote | `.FS` | (`.FE`) |
| Using auto-numbering | **`\*F`** | |

The **mm** embedded commands for printing headings of order *n* are as follows. Headings do not require terminators.

| | |
|---|---|
| Unnumbered headings | `.HU` *n* |
| Numbered headings | `.H` *n* |

To produce a table of contents, place the following **mm** request at the end of the document, after all main text:

| | |
|---|---|
| Table of contents | `.TC` |

## PAGE LAYOUT

The standard page layout for **mm** is a printing area of 6 by 9 inches on an 8-1/2 by 11 inch page, with the page number centered at the top of the page. Using the requests that follow, you can make some changes in the layout.

| | |
|---|---|
| Change the date | `.ND` |
| Change the header | `.PH "'left'center'right'"`<br>`.EH "'left'center'right'"`<br>`.OH "'left'center'right'"` |
| Change the footer | `.PF "'left'center'right'"`<br>`.EF "'left'center'right'"`<br>`.OF "'left'center'right'"` |
| Place the page number | **`\\\\nP`** |
| Place the date | **`\\\\*(DT`** |
| Skip to the next page | `.SK` (absolute) |
| Skip to the next page | `.OP` (odd-numbered) |

## INITIATING FORMATTING

To preview formatting on your screen, simply type a comand line with the name of the command (**mm**), a space, and the name of the file to be formatted. Include the **-c** option if the text contains double columns, tables, or equations; pipe the output to **pg** (UNIX) or **more** (XENIX) to allow a pause with each screenful of text. For printed results, pipe the output to the **lp** print spooler, selecting a printer type if necessary. Here are the options for **mm**:

| | | | |
|---|---|---|---|
| -E | Equal spacing | -12 | Pica output |
| -t | Tables | -y | No compact |
| -e | Equations | -T*name* | Name the printer |
| -c | Columns | | |

# 18

# Formatting with **nroff** and **troff**

In the first two chapters of Part IV, you learned how to use the **nroff** program to format text, with a major emphasis on the **mm** macro processor. In this chapter we'll explore different ways to initiate formatting and a few basic **nroff** requests (which are also **troff** requests).

## 18.1 Initiating formatting

In this section we'll look at some of the different ways you can begin formatting, along with some of the options available to you when you format.

### EXECUTING AN **nroff** COMMAND LINE

In Chapter 16, to get a quick introduction to formatting, we used only the simplest form of the **nroff** command, which sends its output to your screen to be previewed. For **nroff** alone, such a command would look like this:

```
    $ nroff text
```

For processing with the **mm** macro package, the command would look like this:

```
$ nroff -cm text              or    $ mm text
$ _                                 $ _
```

To have the formatted text printed on a line printer, send the output of **nroff** to the **lp** print spooler via a pipeline:

```
    $ nroff text | lp
    $ _
```

If you prefer to have this done in the background, add the background processing symbol **&** to the end of the command line. The UNIX kernel will then assign a process identifier to the job, like this:

```
    $ nroff text | lp &
    1763
    $ _
```

If you have a very large file split into smaller files (possibly with the **split** command), you can take advantage of the pattern-recognition features of UNIX and have **nroff** process the smaller files consecutively:

```
    $ nroff jumbo.[a-d] | lp
    $ _
```

## Specifying a macro package

As mentioned above, you can include the **mm** macro package by using the **-mm** option in an **nroff** command line:

```
$ nroff -cm text              or    $ mm text
$ _                                 $ _
```

While **nroff** will accept the **-mm** option to invoke **mm**, the **-cm** option is preferable. It selects a compact version if it's available, defaults to the ordinary version if not.

Actually, **mm** is just one of a number of macro packages that may be stored in a directory called /usr/lib/tmac. The full pathname of the file that contains **mm** is /usr/lib/tmac/tmac.s. Here is a summary of some names of macro packages and pathnames of the files that contain them:

| Name | Option | Full Pathname of File |
|---|---|---|
| ms | -ms | /usr/lib/tmac/tmac.s |
| mm | -mm | /usr/lib/tmac/tmac.m |
| me | -me | /usr/lib/tmac/tmac.e |

## Requesting a starting page number **-n***p*

If you have printed the first fifteen pages of a document with **nroff**, you may want to continue printing at the next page. To begin formatting at a given page of your document (say 16), you can include an option like **-n16** in your **nroff** command line:

```
    $ nroff -n16 text | lp  [Start printing on page 16]
    $ _
```

## Requesting specific pages **-o***p*

To format only one page of your document (say 7), you can include an option like **-o7** in your **nroff** command line:

```
    $ nroff -o7 text | lp   [Print page 7 only]
    $ _
```

You can also command more than one page at a time, using either commas to list individual page numbers or hyphens to indicate ranges of page numbers:

```
$ nroff -o3,9,18 text | lp       [Print pages 3, 9, and 18]
$ _

$ nroff -o11-13,22 text | lp     [Print pages 11 through 13, then 22]
$ _

$ nroff -o-5 text | lp           [Print pages 1 through 5]
$ _

$ nroff -o14- text | lp          [Print from page 14 to the end of
$ _                               the document]
```

### PAUSING BETWEEN PAGES **-s***n*

If you are printing on stationery or other paper that has to be fed into the printer one sheet at a time, you can have **nroff** pause between pages with the **-s** (stop) option:

```
$ nroff -s text | lp     [Stop printing after each page]
$ _
```

If necessary, you can also type a number after **-s** to have **nroff** stop after a given number of pages. For example, to have **nroff** stop after every third page, you can use this:

```
$ nroff -s3 text | lp    [Stop printing every third page]
$ _
```

### SPECIFYING A PRINTER **-T***name*

The average printer can handle only the most routine printing tasks. It takes a special printer to perform such things as subscripts, superscripts, reverse paper motion, overstriking, and so on. If you have such a printer on your system, you can make this known to **nroff** with the **-T** (type) option. If you don't make this known, the special printing functions won't be performed. For example, if your system has a NEC 5520 Spinwriter, you could use this:

```
$ nroff -T5520 text | lp    [Print on a NEC 5520 Spinwriter]
$ _
```

Since a printer like this can also handle *micro-spacing*, you could also include the **-e** (even) option to command equal spacing of words on a line (also called *proportional spacing*):

```
$ nroff -e -T5520 text | lp   [Print with equal spacing on a NEC
$ _                           [5520 Spinwriter
```

Here is a partial list of printers that can be named in a **-T** option:

| -T Name | Full Name |
|---|---|
| 37 | Teletype Corporation Model 37 (default) |
| lp | Ordinary line printer |
| tn300 | GE TermiNet 300 (or a terminal lacking half-line capability) |
| 300 | DASI 300 |
| 300S | DASI 300S |
| 450 | DASI 450 |
| 5510 | NEC 5510 Spinwriter |
| 5520 | NEC 5520 Spinwriter |

## 18.2 Setting up pages

The basic, atomic instructions used by **nroff** are called *requests.* In this section we'll take a brief look at the requests that allow you to control the overall appearance of each page of text. As discussed in Chapter 19, "More on Formatting," you can use these micro requests to construct your own custom *macro* requests.

### Setting the length of the page **.pl**

The **.pl** (page length) request sets the length of each page you are printing on in lines (where each line is 1/6 of an inch high). By default, the page length is 66 lines, or 11 inches (see Figure 18.1). If you are printing on envelopes or something else that is not 8-1/2 by 11, you will have to reduce the value of **.pl**. For example, suppose you are printing on envelopes that are 4 inches high. You will then have to use this:

```
.pl 24          [Set the page length to 24 lines]
```

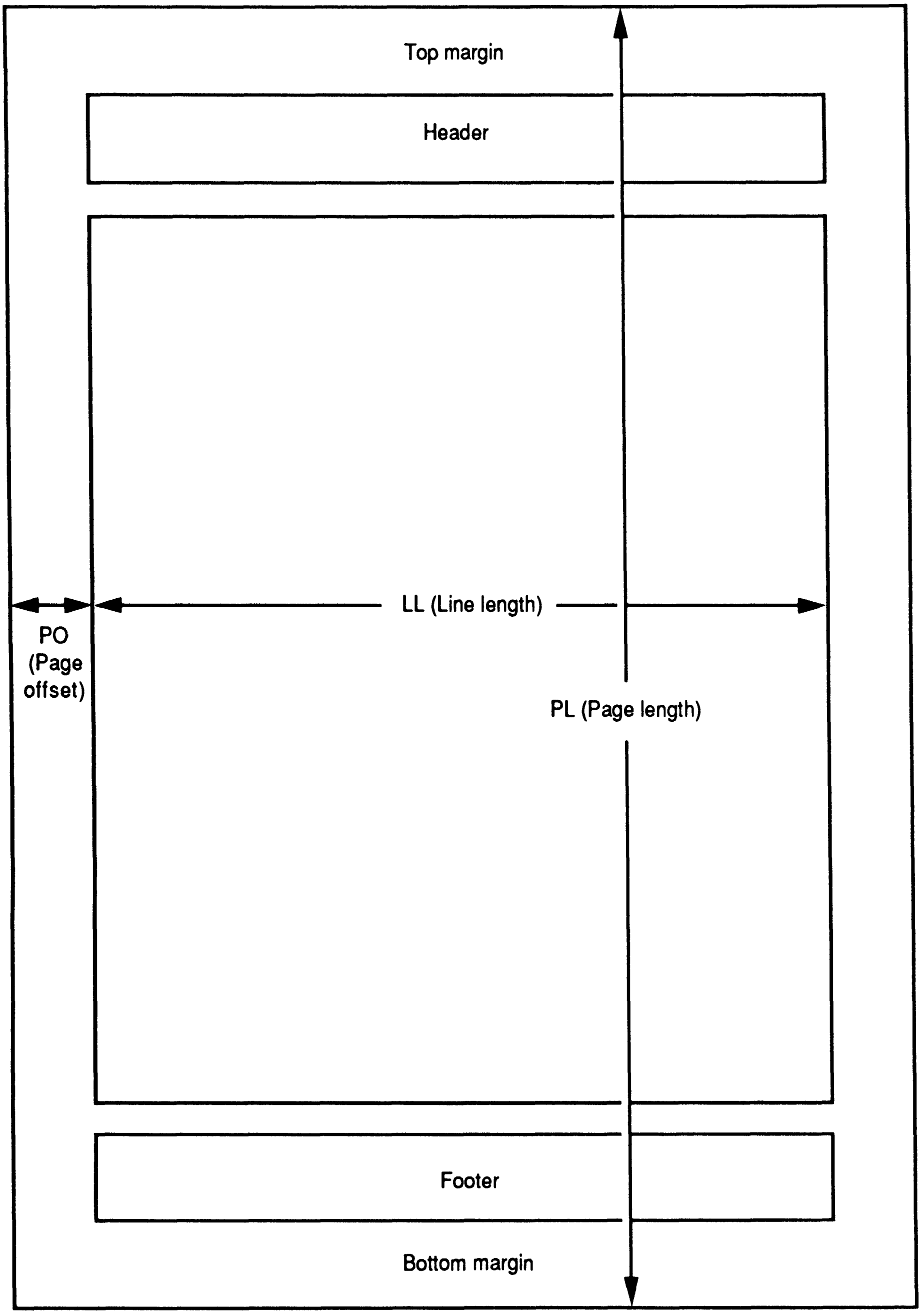
You can use either an absolute value (as in the example above) or a relative value, like this:

```
.pl -12         [Reduce the page length by 12 lines]
```

In this case, the new value of **.pl** is understood to be relative to its previous value. To restore the value to the default (66), simply use **.pl** by itself, without any number:

```
.pl             [Restore the page length to 66 lines]
```

FIGURE 18.1. Standard **nroff** page layout.



## Setting the offset from the edge **.po**

The **.po** (page offset) command sets the distance from the left-hand edge of the paper to the first column of printing in character columns, where each character is 1/10 of an inch wide. By default, the page offset is 0 (right on the edge), which you will probably never want to use (see Figure 18.1). A macro package like **mm** usually sets **.po** to about 10, which just about centers the text between the two edges of the paper when the line length is set to 60. To set **.po** to 8, use this:

```
.po 8              [Set the page offset to 8 characters]
```

You can use either an absolute value (as in the example above) or a relative value, like this:

```
.po -2             [Reduce the page offset by 2 characters]
```

In this case, the new value of **.pl** is understood to be relative to its previous value. To restore the value before your last change to **.po**, simply use **.po** by itself, without any number:

```
.po                [Restore the page offset to its previous value]
```

## Setting the length of the line **.ll**

The **.ll** (line length) command sets the length of each line in character columns (where each character is 1/10 of an inch wide). By default, the line length is 65 characters, or 6.5 inches (see Figure 18.1). (In **mm**, the default is 60 characters.) If you are printing on wide computer paper or something else that is not 8-1/2 by 11, you will have to increase or reduce the value of **.ll**. For example, suppose you are printing on wide computer paper that is 15 inches wide. You will then have to use this:

```
.ll 132            [Set the line length to 132 characters]
```

You can use either an absolute value (as in the example above) or a relative value, like this:

```
.ll +12            [Increase the line length by 12 characters]
```

In this case, the new value of **.ll** is understood to be relative to its previous value. To restore the value to the default (65), simply use **.ll** by itself, without any number:

```
.ll                [Restore the line length to 65 characters]
```

## INDENTING TEXT

**.in** *n*
**.ti** *n*

The **.in** (indentation) request, followed by an optional number, sets the indentation of text with respect to the current left margin. Use a positive number to indent to the right; use a negative number to indent to the left. If you don't include a number, indentation reverts to its previous location. Here are some examples.

```
Input                          Output
dddddddddddddddddd             dddddddddddddddddd
dddddddd.                      dddddd.
.in +5                              eeeeeeeeeeeee
eeeeeeeeeeeeeeeeeee.                eeeeeee.
.in -3                           fffffffffffffffff
ffffffffffffffffff               ffffff.
ffffff.                          gggggggggg.
.in
gggggggggggg.
```

The **.ti** (temporary indentation) request, followed by a required number, is usually used to provide indentation for a single paragraph. Here is an example:

```
Input                          Output
hhhhhhhhhhhhhhhhhhh            hhhhhhhhhhhhhhhhhhh
hhhhhhhhhh.                    hhhhhhh.
.ti +5                              iiiiiiiiiiiii
iiiiiiiiiiiiiiiiiiii                iiiiiiiiiiii.
iiiiii.                        jjjjjjjjjjjjjjjjjjjj
                               jjjjjjjj.
jjjjjjjjjjjjjjjjjjjj
jjjjjjjjjjj.
```

The **.ti** request can also be combined with the **.ll** (line length) request to indent a paragraph on both sides, as shown here:

```
Input                          Output
kkkkkkkkkkkkkkkkkkkkk          kkkkkkkkkkkkkkkkkkk
kkkkkkkkkk.                    kkkkkkkkk.
.ti +5                               llllllllll
.ll -5                               lllllllll.
llllllllllllllllllll.          mmmmmmmmmmmmmmmmmmmmmmmmm
.ll +5                         mmmmmmmmmmmmmmmmm.
```

## CHANGING THE PAGE NUMBER

**.pn**

Whenever **nroff** starts formatting a new file, it sets its *page number register* to 1, and then increments this register for each page printed. Ordinarily, you would not want to disrupt this scheme. However, suppose you have a larger file broken into two smaller files called **section_1.a** and **section_1.b**, where **section_1.b** is a continuation of **section_1.a**. If **section_1.a** is 26 pages long, then you would want to have this near the beginning of **section_1.b**:

```
.pn 27              [Set the page number to 27]
```

As with the previous commands, you can also use a relative value. For example, to increase the page number by two, you could use this:

```
.pn +2              [Increase the page number by 2]
```

If you use **.pn** by itself, without any number following, **nroff** will ignore the request:

```
.pn                 [Ignored by nroff]
```

## Forcing a page break **.bp**

Ordinarily it won't make any difference to you where **nroff** ends one page and starts another. But there will be times when you want to be sure that a particular item appears only at the top of a new page. To force a page break at such a point, you can use the **.bp** (break page) command. For example, if you had a major heading in your document, you could precede it with this:

```
.bp                 [Start a new page here]
```

This guarantees that **nroff** will print the heading at the top of a new page, no matter how much text appears on the previous page. You can also force a new page number. For example, to start a new page called page 21, you could use this:

```
.bp 21              [Start a new page and call it "page 21"]
```

You can also use a relative number. For example, suppose the current page is page 15 and you want to make sure that the next page begins on page 17 (the front of a two-sided sheet). You could then use this:

```
.bp +2              [Start a new page and increase the page number by 2]
```

## Keeping lines of text together **.ne**

Continuing the reasoning from the previous request (**.bp**), you may have a block of text that you never want to be split. In other words, if it will all fit together on the current page, fine; but if it won't, then you want it all moved to the next page. The **.ne** (need) request, with a number following, allows you to keep a certain number of lines together on the same page. For example, suppose you have a heading followed by five lines of text (a total of seven lines). To ensure that these seven lines never become separated by a page break, precede the heading with this:

```
.ne 7               [Keep the next seven lines together on one page]
```

## 18.3 Formatting lines of text

In the previous section we discussed briefly the **nroff** requests that allow you to set up the overall format of entire pages. In this section we'll cover the requests that relate to individual lines of text.

### Filling lines

**.fi**
**.nf**

With titles, lines of poetry, and certain lists of items, it's desirable to print each line exactly as it is entered, open spaces and all. However, with paragraphs, you usually want the open space at the end of a line filled in with text from the following line, like this:

```
Input                          Output
xxxxxxxxxxxxx                       xxxxxxxxxxxxxxxxx
xxxxxxx                        xxxxxxxxxxxxxxxxxx.
xxxxxxxxxxx.
```

This is called *filling.* To ensure that filling takes place, precede text with this:

```
.fi            [Fill the text that follows]
```

To turn off this feature, use the following:

```
.nf            [Do not fill the text that follows]
```

### Adjusting lines

**.ad**
**.na**

Positioning lines of text on the page is called *adjusting* text. The **nroff** request for doing this is **.ad**, which must be followed by a one-letter code that indicates how you want the lines adjusted (flush left, flush right, flush left and right, or centered):

```
Input                          Output

                               .ad l
llllllllllllllll               llllllllllllllllllll
lllllllllllllllllll            lllllllllllllllll
lllllll.                       llllllll.
.ad r
rrrrrrrrrrrrrrrr                  rrrrrrrrrrrrrrrrr
rrrrrrrrrrrrrrrrrr                 rrrrrrrrrrrrrrrr
rrrrrrr.                                 rrrrrrrrr.
.ad b
bbbbbbbbbbbbbbbb               bbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbb            bbbbbbbbbbbbbbbbbbbb
bbbbbbbb.                      bbbbbbbb.
```

```
.ad c
cccccccccccccccc                  cccccccccccccccc
cccccccccccccccccccc            cccccccccccccccccccc
cccccccc.                              cccccccc.
```

Flush left and right is often referred to as *justified* text, which can also be specified by the following request:

```
Input                           Output

                              .ad n
nnnnnnnnnnnnnnnn                nnnnnnnnnnnnnnnnnnnn
nnnnnnnnnnnnnnnnnnnn            nnnnnnnnnnnnnnnnnnnn
nnnnnnnn.                       nnnnnnnnn.
```

In other words, **.ad b** (both) and **.ad n** (normal) are identical. To turn off adjusting of text, use this:

```
    .na                 [No adjustment]
```

## Hyphenating words **.hy .nh**

You can have **nroff** automatically hyphenate words near the right margin by placing the following request near the beginning of your document:

```
Input                           Output

.hy
This resulted in a              This resulted in a pro-
proliferation of requests...    liferation of requests
```

To turn off hyphenation again, use this:

```
Input                           Output

.nh
This resulted in a              This resulted in a
proliferation of requests...    proliferation of
                                requests for...
```

## Breaking a line **.br**

To break in the middle of a line and begin a new line, enter the **.br** (break) request at the desired location, as shown here:

```
Input                           Output

xxxxxxxxx                       xxxxxxxxx
.br                             yyyy.
yyyy.
```

Without the **.br** request, the two lines would have become a single line:

```
xxxxxxxxx                       xxxxxxxxxyyyy.
yyyy.
```

### Setting line spacing **.ls n**

You can request double-, triple-, or any other spacing with the **.ls** request, followed by a number, as shown below. If you type it without a number, **nroff** reverts to the previous spacing.

```
Input                                Output

.ls 2                      Double
aaaaaaaaaaaaaaaaaaaaaaa             aaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaa
aaaaaaaaaa.                         aaaaaaaaaaaaaaaaaa
.ls
bbbbbbbbbbbbbbbbbbbbbbbbb           aaaaaaaaaaaa.
bbbbbbbbbbbbbbbbbbbb.
                                    bbbbbbbbbbbbbbbbbbbbbbb
                                    bbbbbbbbbbbbbbbb.
```

### Spacing vertically **.sp** *n*

Sometimes you need to have a segment of text printed either above or below the current line. The **.sp** (space) request, followed by a positive number, allows you to place text lower than the current line, as shown here:

```
Input                                Output

wwwwwwwwwwwwwwwwww                   wwwwwwwwwwwwwwwww
.sp 2
xxxxxxxx.                            xxxxxxx.
```

### Centering lines **.ce** *n*

This centering request, which is similar to **.ad c**, allows you to center a given number of lines, as shown here:

```
Input                                Output

.ce 2
tttttttttttttttttttt                  tttttttttttttttttt
tttttttt                                   tttttttt
uuuuuuuuuuuuuuuuuuuuu                uuuuuuuuuuuuuuuuuuuu
uuuuuuuuuu.                          uuuuuuuuu.
```

### Underlining text **.ul** *n* **.cl** *n*

The request for ordinary underlining (**.ul**) places an underscore under each visible character, but leaves blank spaces blank, like this:

| Input | Output |
|---|---|
| **.ul**<br>`This is how to underline`<br>`text.` | `This is how to underline`<br>`text.` |

Another request (**.cl**) provides for continuous underlining, including spaces between words, like this:

| | |
|---|---|
| **.cl**<br>`This is how to underline`<br>`text.` | `This is how to underline`<br>`text.` |

### Producing superscripts and subscripts

**u**
**d**

To produce superscripts and subscripts, you can use one request to cause printing one-half line up (\**u**) and one to cause printing one-half line down (\**d**). Here are some examples:

| Input | Output |
|---|---|
| **.ls 2**<br>`The atmosphere of Venus`<br>`contains sulphuric acid`<br>`(H\d2\u SO\d4\u).` | `The atmosphere of Venus`<br><br>`contains sulphuric acid`<br><br>$(H_2SO_4)$. |

## 18.4 Summary

In this chapter you learned to initiate formatting with **nroff** and to use some basic **nroff** requests.

### Initiating formatting

To preview formatting on your screen, simply type a command line with the name of the command (**nroff**), a space, and the name of the file to be formatted. For printed results, send the output of **nroff** to the **lp** print spooler via a pipeline. To use the **mm** macro package, include the **-mm** option between **nroff** and the name of the file. This package is stored (with other macro packages) in /usr/lib/tmac/tmac.s. Other options:

| | |
|---|---|
| **-n***p* | Starting page number |
| **-o***p* | Specific pages |
| **-s***n* | Pausing between pages |
| **-T***name* | Specifying a printer |

## FORMATTING ENTIRE PAGES

| | |
|---|---|
| **.pl** | Page length |
| **.po** | Page offset |
| **.ll** | Line length |
| **.in n** | Indent by n |
| **.ti n** | Temporary indentation |
| **.pn** | Set the page number |
| **.bp** | Page break |
| **.ne** | Need (lines together) |

## FORMATTING LINES OF TEXT

| | |
|---|---|
| **.fi** | Turn on filling |
| **.nf** | No filling |

| | | | |
|---|---|---|---|
| **.ad** | l | Adjust | flush left |
| | r | | flush right |
| | c | | both (flush left and right) |
| | bn | | normal (flush left and right) |

| | |
|---|---|
| **.na** | No adjustment |

| | |
|---|---|
| **.hy n** | Hyphenation |
| **.nh** | No hyphenation |
| **.br** | Break to a new line |
| **.ls n** | Line spacing n |
| **.ce n** | Center next n lines |
| **.ul** | Underline |
| **.cu** | Continuous underline |
| **%** | Page number |
| **u** | Superscript |
| **d** | Subscript |

# 19

# Formatting with **troff**

Chapter 18 covered the features common to both **nroff** and **troff**. This chapter covers those features that are unique to **troff**, which is used to drive laser printers and phototypesetters. Before describing these features themselves, we'll begin with some background information.

## 19.1 Printing and typesetting

### THE TRADITIONAL METHODS

Around 1436 Johann Gutenberg revolutionized the process of producing books when he invented movable type. His invention replaced hand lettering and illustrating with a new technology based on the use of pieces of metal type. Typesetters selected characters one at a time from wooden cases and inserted them into metal frames. When a line of type approached the side of the frame, the typesetter either hyphenated the final word or inserted extra spacers between the existing words until the line filled the space available.

From our point of view, the distinguishing characteristic of these pieces of type was that the width of each letter varied according to the formation of the letter. For example, i and j were very thin, while m and w were very wide. In this century, this tradition has been maintained with the introduction of phototypesetting machines. Although these machines have replaced individual pieces of metal type with photographic techniques, different letters have continued to occupy different widths on the page.

### TYPEWRITERS AND THE EARLY COMPUTERS

About a century ago, the invention of the typewriter took printing on a detour from traditional printing. Because of its mechanical limitations, the typewriter has required all characters to occupy identical widths on the page. On a typewriter, the hammer that contains `i` is the same width as the hammer that contains `w`. In addition, most typewriters produce similar-looking characters—characters that have a common "typewriter" appearance. The result is printed text that is much less attractive than typeset text.

This detour from true printing technology continued into the early decades of the computer age. Until the mid-1980s, printers that were connected to computers (whether they were line printers, daisy-wheel character printers, or dot-matrix character printers) produced results that resembled typewritten output. Characters were of uniform width and appearance. In addition, these limitations extended to the video display screens. On the screen and on the printed page, the rule was uniform width and uniform appearance.

This began to change in the 1980s with the introduction of two technical innovations: the graphical video display and the laser printer. Graphical video displays transformed the screen from a collection of character cells to a blank canvas, on which shapes of any description could be drawn. Laser printers similarly transformed the printed page. For the first time since the introduction of computers, it was possible to print characters that varied in height, width, and appearance. These two devices have restored the methods of traditional printing, uniting Gutenberg's methods with computer technology.

## Laser printers and **troff**

The daisy-wheel printers that were widely used in the 1970s and early 1980s have been rapidly superseded by laser printers. Besides being faster and quieter, laser printers have a number of advantages over their predecessors, which are intertwined with current computer technology. The two main advantages are their ability to print text and graphics on the same page and their ability to print text in more than one size and appearance on the same page.

While the daisy-wheel printer was modeled after the typewriter, the laser printer is modeled after the phototypesetting machine. Like a full-scale typesetter, the laser printer allows you to print headlines, titles, and paragraph headings larger than body text. The main difference between a laser printer and a true typesetter is that the typesetter offers higher resolution and a larger selection of character designs to choose from. A typical laser printer today offers resolution of 300 dots per inch (dpi), while a typesetter offers 1200–2400 dpi.

The following examples contrast the output of a daisy-wheel printer driven by **nroff** and a laser printer driven by **troff**:

**nroff Output**

```
This is an example of printing
with constant spacing, where an
i and a w are the same width.
```

**troff Output**

This is an example of printing
with proportional spacing, where
an i is much narrower than a w.

## Typefaces and fonts

A *typeface* is a family of characters designed to share a common appearance. Two of the most popular typefaces in use include Times Roman and

Helvetica. Times Roman is an example of a *serif* typeface, in which each letter includes small decorations called serifs. The body text of this book is set in Times Roman; you can see the serifs at the top and bottom of the letter l. Helvetica is an example of a *sans-serif* typeface, which does not use serifs. Each typeface includes a variety of sizes, measured in points, as well as **bold** and *italic* versions. (The "plain" version of a typeface, whether serif or sans-serif, is known as *roman*—with a lowercase r.)

A specific size and style of a given typeface is called a *font.* For instance 18-point Times Roman bold and 12-point Helvetica italic are examples of fonts. A phototypesetter typically provides four fonts for each typeface in a given size: roman, italic, bold, and special. The special font contains the Greek alphabet, plus a variety of mathematical and logical symbols. The four fonts look like this:

Times Roman

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz

Times Italic

*ABCDEFGHIJKLMNOPQRSTUVWXYZ*
*abcdefghijklmnopqrstuvwxyz*

Times Bold

**ABCDEFGHIJKLMNOPQRSTUVWXYZ**
**abcdefghijklmnopqrstuvwxyz**

Special

αβγδεζηθικλμνξοπρστυφχψω
ΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΦΧΨΩ

## Laser printer fonts

In a phototypesetter, fonts are often stored on a wheel—usually four to a wheel. In a computer system, fonts are stored in memory in files. It may be computer memory or printer memory. We'll discuss two types of laser printers to elaborate a little.

The most popular laser printers used today belong to the Hewlett-Packard LaserJet series. The LaserJet can access fonts that are stored in any of three different places:

- in the LaserJet's internal storage
- in a cartridge inserted into the LaserJet
- in a computer file that can be downloaded to the LaserJet

The internal fonts include 12-point Courier roman and bold, which look like typewriter output, and 8.5 Helvetica, which is too small to serve any useful purpose. You can supplement these with font cartridges sold by Hewlett-Packard and downloadable fonts, which you can buy from other vendors.

Another popular laser printer is the Apple LaserWriter. The LaserWriter has many of the same features of the LaserJet, plus one major feature not found on the LaserJet: Adobe PostScript. PostScript is an example of a page description language, which provides a common printer interface for a variety of different laser printers and phototypesetters. The primary advantage of PostScript is that you can send the same text to a PostScript laser printer and a PostScript typesetter without having to make any changes. (Usually, you have to insert extensive coding into a file before you can send it to a typesetting machine.)

## 19.2 Introduction to **troff**

### **troff** AND **ditroff**

The **troff** formatter was originally designed to drive a single typesetter, the Wang C/A/T. Since then, a number of printer drivers, or post-processors, have been developed to allow you to direct output to other machines. In addition, AT&T has developed an enhanced version of **troff**, which is known as device-independent **troff**, or **ditroff**. As its name implies, **ditroff** can drive a number of different printers.

So far, **ditroff** has reached only a limited number of users because it isn't part of the standard System V package. Instead, it comes separately with the Documenter's Workbench. Also, system administrators sometimes change the name of the program to **troff**. If you aren't sure whether you have **troff** or **ditroff**, take a look at directory /usr/lib/font. If you see nothing but fonts in the directory (with names like ftR, ftI, ftB, ftS), then you have **troff**; if you also find the names of subdirectories for device drivers (with names like devlj for LaserJet or devps for PostScript), then you have **ditroff**.

### EXECUTING A COMMAND LINE

With either formatter, you have to pipe text files through a post-processor en route to a laser printer. Here are typical command lines for sending text (in a file called text) to a laser printer:

```
$ troff -t text | laserjet | lp
```

This **troff** command line includes a **-t** flag to indicate that the output is going to the standard output, instead of to a Wang C/A/T, and a post-processor called **laserjet**.

```
$ ditroff -Tlj text | devlj | lp
```

This **ditroff** command line includes a **-T** flag to identify the output device and the name of a post-processor (**devlj**).

All the command line options described in Chapter 18, "Formatting with **nroff**," can also be used with **troff** and **ditroff**.

## 19.3 Working with **troff**

To accommodate laser printers and typesetting machines, **troff** offers all the features of **nroff**, plus these additional capabilities:

- changing point size
- changing spacing
- selecting fonts
- printing special characters

We'll discuss these capabilities in the paragraphs that follow.

### CHANGING POINT SIZE **.ps** *n*

The original **troff** allows the following 15 point sizes from 6 to 36: 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36. The point sizes supported by **ditroff** depend on the printer being used. In either case, the default is 10 points. To change the size, use the **.ps** request, followed by the desired point size, as shown here:

| **Input** | **Output** |
|---|---|
| `This is normal 10-point text.` | This is normal 10-point text. |
| `.ps 9` | This is smaller 9-point text. |
| `This is smaller 9-point text.` | This is normal text again. |
| `.ps` | |
| `This is normal text again.` | |

Without an argument, **.ps** tells **troff** to revert to the size before the most recent change (that is, two sizes back). If you select a point size that is not available, **troff** will use the nearest available value instead. The **.ps** request also allows relative changes, as shown below (the results are identical to those in the previous example above).

```
Input                              Output

This is normal 10-point text.      This is normal 10-point text.
.ps -1                             This is smaller 9-point text.
This is smaller 9-point text.      This is normal text again.
.ps +1
This is normal text again.
```

To make changes within a line, you can use the **s** escape sequence. It works the same way as the **.ps** request, except that you use zero (0) to revert to the previous size. Here is an example, in which we use relative numbers to reduce the word UNIX by one point:

```
Input                              Output

The \s-1UNIX\s0 system is good.The UNIX system is good.
```

## CHANGING VERTICAL SPACING

**.vs** *n*

Whenever you change point size, it's desirable to change the vertical spacing between lines with it. Otherwise, the lines will be printed either too close together or too far apart. As pointed out in Chapter 16, "Introduction to **mm**," vertical spacing should be about 20% larger than point size. Here is an example of increasing point size and vertical spacing together:

```
Input                              Output

This is normal 10-point text.      This is normal 10-point text.
.ps 12                             This is 12-point text.
.vs 14
This is 12-point text.             This is normal text again.
.ps
This is normal text again.
```

## CHANGING HORIZONTAL SPACING

**.ss** *n*
**.cs** *f n*

Horizontal spacing is usually specified in *ems* (where one em is the width of the letter m in the current font. The default amount of space between words of unjustified **troff** output is one-third em. (For justified text, the default space becomes the minimum space between words.) If you need to change this for any reason, you can use the **.ss** (spacing size) request to increase or decrease the word spacing in 1/36 ems. In the following example, we increase the word spacing to 2/3 em (24/36), then restore it to the default 1/3 em (no argument).

```
Input                              Output

This is normal word spacing.       This is normal word spacing.
.ss 24                             This  is  twice  as  wide.
```

```
This is twice as wide.
.ss
This is back to normal again.
```

This is back to normal again.

For certain types of output, such as tables of numbers or symbols, it's important to maintain vertical alignment of columns. Another **troff** request, **.cs** (constant spacing), allows you to achieve this result. Like **.ss**, the **.cs** request also uses a numeric argument that is measured in 1/36 ems; in addition, it also requires a font code. The font code identifies the font; the numeric argument indicates the constant width to be allocated to each character. Again, **.cs** without a numeric argument means to revert to the previous character spacing.

**Input**

```
Here is a small matrix:
.br
.cs R 36
A B C D E
.br
F G H I J
.br
K L M N O
.br
.cs R
Now we return to text again.
```

**Output**

Here is a small matrix:

```
A B C D E
F G H I J
K L M N O
```

Now we return to text again.

## Selecting fonts **.ft** *n*

Fonts for the Wang C/A/T typesetter were mounted on a rotating wheel, each in a separate numbered quadrant, or font position, on the wheel. By convention, the roman font occupied the first quadrant (font position 1), with the other fonts as shown below. Consequently, the original **troff** allows you to use only four fonts at a time.

| Font Position | Font | Letter Code |
|---|---|---|
| 1 | Roman | R |
| 2 | Italic | I |
| 3 | Bold | B |
| 4 | Special | S |

Unless otherwise instructed, **troff** prints text in roman. To select another font, you can use the **.ft** request with either a literal or a numeric argument. Here is an example of switching to bold and then back to roman again, using literal arguments:

**Input**

```
This is roman text.
.ft B
```

**Output**

This is roman text. **This is bold text.** This is roman again.

```
This is bold text.
.ft R
This is roman again.
```

The following example is equivalent to the previous example:

**Input**

```
This is roman text.
.ft 3
This is bold text.
.ft P
This is roman again.
```

**Output**

This is roman text. **This is bold text.** This is roman again.

To change to fonts within a line, you can use the **f** escape sequence, as shown in the following example:

**Input**

```
This is a \fBbold\fP word.
This is an \fIitalic\fP word.
```

**Output**

This is a **bold** word. This is an *italic* word.

The following example is equivalent to the previous example:

**Input**

```
This is a \f3bold\f1 word.
This is an \f2italic\f1 word.
```

**Output**

This is a **bold** word. This is an *italic* word.

With **ditroff**, you have an unlimited number of fonts available. Suppose there are ten fonts—the Times Roman fonts (plus bold italic), three Helvetica fonts, and two constant width fonts—arranged as shown here:

| Font Position | Font | Letter Code |
|---|---|---|
| 1 | Times roman | R |
| 2 | Times italic | I |
| 3 | Times bold | B |
| 4 | Times bold italic | BI |
| 5 | Helvetica roman | H |
| 6 | Helvetica italic | HI |
| 7 | Helvetica bold | HB |
| 8 | Constant width roman | CW |
| 9 | Constant width bold | CB |
| 10 | Special | S |

With **ditroff**, **.ft** and **f** are the same. However, if you're specifying a font with a two-letter code after **f**, you must precede the code with **(**. Here is an example of changing Helvetica fonts with the **.ft** request:

**Input**

```
This is roman text.
.ft HB
This is bold text.
.ft H
```

**Output**

This is roman text. **This is bold text.** This is roman again.

```
This is roman again.
```

The following example is equivalent to the previous example:

```
Input                           Output

This is roman text.             This is roman text. This is bold
.ft 7                           text. This is roman again.
This is bold text.
.ft P
This is roman again.
```

Here is an example of changing Helvetica fonts with the **f** escape sequence:

```
Input                           Output

This is a \f(HBbold\fP word.    This is a bold word. This is an
This is an \f(HIitalic\fP word. italic word.
```

The following example is equivalent to the previous example:

```
Input                           Output

This is a \f7bold\f1 word.      This is a bold word. This is an
This is an \f6italic\f1 word.   italic word.
```

## Changing font positions **.fp** *n f*

If it should every become necessary to change typefaces (say from Times Roman to Helvetica), you can assign font positions manually with the **.fp** (font position) request, as shown here:

| | |
|---|---|
| **1** | Assign Helvetica roman to font position 1 |
| **2 H** | Assign Helvetica italic to font position 2 |
| **3 H** | Assign Helvetica bold to font position 3 |

As long as you keep the new fonts in the same order (roman, italic, bold), you can now print your files in Helvetica without changing any codes as long as you use numeric codes after **.ft** requests and **f** escape sequences. Then Times italic will become Helvetica italic, and so on.

## Printing special characters

Many of the special characters provided in **troff** fonts, such as the degree symbol, can't be typed with a single keystroke. However, you can enter these characters using escape sequences of the form \**(***xx*, as shown in Tables 19.1 and 19.2.

TABLE 19.1. Special Character Codes

| Character | Name | Escape Sequence |
|---|---|---|
| • | Bullet | `\(bu` |
| □ | Square | `\(sq` |
| — | Em dash | `\(em` |
| ▬ | Baseline rule | `\(ru` |
| _ | Underline | `\(ul` |
| © | Copyright symbol | `\(co` |
| ® | Registered symbol | `\(rg` |
| § | Section symbol | `\(sc` |
| † | Dagger | `\(dg` |
| ‡ | Double dagger | `\(dd` |
| ↑ | Up arrow | `\(ua` |
| ↓ | Down arrow | `\(da` |
| ← | Left arrow | `\(<-` |
| → | Right arrow | `\(->` |
| 1/4 | One quarter | `\(14` |
| 1/2 | One half | `\(12` |
| 3/4 | Three quarters | `\(34` |
| ¢ | Cent symbol | `\(ct` |
| ° | Degree symbol | `\(de` |
| ± | Plus or minus symbol | `\(+-` |
| × | Multiplication symbol | `\(mu` |
| ÷ | Division symbol | `\(di` |
| √‾ | Square root symbol | `\(sq` |
| ≥ | Greater than or equal to | `\(>=` |
| ≠ | Not equal to | `\(!=` |
| ≤ | Less than or equal to | `\(<=` |
| ﬁ | ﬁ ligature | `\(fi` |
| ﬂ | ﬂ ligature | `\(fl` |
| ﬀ | ﬀ ligature | `\(ff` |
| ﬃ | ﬃ ligature | `\(Fi` |
| ﬄ | ﬄ ligature | `\(Fl` |

## 19.4 Summary

In this chapter you learned how to execute a **troff** command line and how to use the formatting features that are unique to **troff**.

TABLE 19.2. Greek Letter Codes

| Letter | Name | Sequence | Letter | Name | Sequence |
|---|---|---|---|---|---|
| α | alpha | \(*a | A | ALPHA | \(*A |
| β | beta | \(*b | B | BETA | \(*B |
| γ | gamma | \(*g | Γ | GAMMA | \(*G |
| δ | delta | \(*d | Δ | DELTA | \(*D |
| ϵ | epsilon | \(*e | E | EPSILON | \(*E |
| ζ | zeta | \(*z | Z | ZETA | \(*Z |
| η | eta | \(*y | E | ETA | \(*Y |
| θ | theta | \(*h | Θ | THETA | \(*H |
| ι | iota | \(*i | I | IOTA | \(*I |
| κ | kappa | \(*k | K | KAPPA | \(*K |
| λ | lambda | \(*l | Λ | LAMBDA | \(*L |
| μ | mu | \(*m | M | MU | \(*M |
| ν | nu | \(*n | N | NU | \(*N |
| ξ | xi | \(*c | Ξ | XI | \(*C |
| ο | omicron | \(*o | *O* | OMICRON | \(*O |
| π | pi | \(*p | Π | PI | \(*P |
| ρ | rho | \(*r | R | RHO | \(*R |
| σ | sigma | \(*s<br>\(ts | Σ | SIGMA | \(*S |
| τ | tau | \(*t | T | TAU | \(*T |
| υ | upsilon | \(*u | Υ | UPSILON | \(*U |
| ϕ | phi | \(*f | Φ | PHI | \(*F |
| χ | chi | \(*x | X | CHI | \(*X |
| ψ | psi | \(*q | Ψ | PSI | \(*Q |
| ω | omega | \(*w | Ω | OMEGA | \(*W |

## Printing and typesetting

This chapter opened with some background information on typesetting, laser printers, typefaces, fonts, **troff** and device-independent **troff** (**ditroff**).

## Working with troff

The following requests and escape sequences allow you to make the adjustments indicated:

| | |
|---|---|
| **.ps** *n* | Change the point size |
| \f*n* | |
| **.vs** *n* | Change the vertical spacing |
| **.ss** *n* | Change the word spacing |
| **.cs** *f n* | Change to constant character spacing |
| **.ft** *n* | Select a font |
| \f*n* | |
| **.fp** *n f* | Change font positions |

With escape sequences of the form \(*xx* you can enter a number of special characters and symbols not found on your keyboard.

# 20

# More on Formatting

In the first four chapters of Part IV, you learned how to use **mm**, **nroff**, and **troff** to format text. In this chapter, we'll discuss methods for producing double-column printing, tables, equations, your own macro requests, and custom modifications to the formatting programs.

## 20.1 Using double-column format

### SETTING UP THE TEXT **.2C**

If you would like to see your text in narrower columns, you can have **mm** print your document in double-column format with the **.2C** command. Then use the **.1C** command to revert to single-column format.

```
Input                           Output

.2C
.P
2222222222222222222             222222222    2222222222
2222222222222222                222222222    2222222222
22222222222222222222222.        222222222    2222222.
.1C
.P                              11111111111111111111111
111111111111111111111111        11111111111111111111111
11111111111111111111111111111.  1111111111.
```

### PREPARING THE OUTPUT FOR YOUR PRINTER **col**

Ordinarily, a printer will be able to print a document in more than one column only if the printer is capable of reversing the motion of the paper. Since most printers do *not* have this capability, you usually have to pipe the text from **nroff** to a *filter program* called **col** (columns) and then on to **lp**. The **col** program rearranges the text in such a way that the printer is not required to reverse the motion of the paper. (This also applies to text that you plan to preview on your screen.)

Whenever you use **col**, it is a good idea to specify the type of printer to which you are sending the text. You can do this by including the **-T** (type) option with the **nroff** command. For example, to specify a line printer, type ***-T***, followed by **lp** (line printer), or **-Tlp** on the command line. For example, to print a file called **story** in double-column format, you could use this:

```
$ nroff -cm -Tlp story | col | lp
```

or

```
$ mm -c -Tlp story | lp
```

You must also use **col** whenever you use either the **box** or **allbox** option with **tbl** (see later in this chapter).

## A TWO-COLUMN EXAMPLE

To see an example of two-column printing very quickly, make a copy of file **wall** called **wall.2C**, then begin an editing session with **wall.2C** and add the six formatting requests shown below. Now the first seven lines of the text will look like this:

```
.pl 33
.DS C
Request for Wall
.DE
.2C
.P 1
          Beijing (Peking).  The Mayor of West Berlin
```

With these formatting requests in place (plus **.P 1** on the blank line above each of the other three paragraphs), execute the following command line:

```
$ mm wall | col


                              - 1 -

                         Request for Wall

     Beijing (Peking).  The        Some 25 feet  high,  15
  Mayor  of West Berlin  stood   to 30 feet wide at the base,
  yesterday  on  top  of  the    and 12 feet wide at the top,
  Great  Wall of China next to   the wall is over 1,500 miles
  Deng  Xiao-ping,  leader  of   (2,400  km)  long,   greater
  the  nation  of  one billion   than  the  distance  between
  people.                        New York and Dallas.

     Begun during the  Ch'in        The purpose of the wall
  dynasty (about the time Rome   was to protect China against
  fought the first Punic War),   invaders from the north. The
  the  wall  was not completed   mayor  told  his host, "This
  until    the   Ming  dynasty   wall was here to keep people
  (about    the    time    the   out.  We have a wall that is
  Mayflower     arrived     at   there to keep people in."
  Plymouth Rock).
```

### Handling footnotes and displays

If you plan to intersperse double-column text with either footnotes or displays (or both), you can use the **.WC** (wide column) request to specify how the footnotes or displays are to be formatted. Here are the options for the **.WC** request:

| | |
|---|---|
| WF | Wide footnotes (margin to margin) |
| -WF | Footnotes follow column width of text (default) |
| FF | First footnote determines style for all footnotes |
| -FF | Footnotes follow **WF** or **-WF** (default) |
| **.WC WD** | Wide displays (margin to margin) |
| -WD | Displays follow column width of text (default) |
| FB | Each floating display causes a break (default) |
| -FB | Floating displays do not cause breaks |
| N | Default mode (pg -WF -FF -WD FB) |

For example, to specify narrow footnotes and displays and to cause an **nroff** break for each floating display with double-column text, you could use this request:

```
.WC N
```

To specify wide footnotes and displays with double-column text and to cause an **nroff** break for each floating display, you could use this request:

```
.WC WF -FF WD FB
```

Note that the term *display* includes tables and equations, which are described later in this chapter.

## 20.2 Formatting tables

### Introductory example

To get a quick glimpse at the basic features of the **tbl** preprocessor for producing tables, let's format the letter we used in Chapter 16 once more.

This time we'll use **tbl** instead of **mm** to format the tabular material. This means replacing the two display commands (**.DS** and **.DE**) with table commands (**.TS** and **.TE**) and adding a few more commands to improve the appearance of the text.

1. Prepare the text for formatting again:

   - Start another editing session with **pfl.deal**.
   - Change **.DS** to **.TS** (line 8) and **.DE** to **.TE** (line 12), then add the five extra lines shown below and insert the colons, leaving the rest of the text unchanged:

```
.TS                           [Start of table]
center ;                      [Center the table]
c c c c                       [Center each heading]
l n n l .                     [Allow four columns*]
Name Height Weight Team       [These are the headings, separated
                               by tabs                          ]
.sp 1                         [Leave a space after the headings]
Emerson, Ezekiel R.           6'5"    273     Rochester
Robinson, Charles F.          6'3"    287     Des Moines
Peterson, Paul N.             5'9"    178     Tupelo
.TE                           [End of table]

                              [* Columns 1 and 4: left-justified;
                               columns 2 and 3: numeric          ]
```

   - Store the text and return to the UNIX shell prompt.

2. Format the text on your screen with **mm**:

   - Type a command line that invokes the **tbl** preprocessor:

```
$ mm -t pfl.deal | more
```

   or

```
$ tbl pfl.deal | nroff -cm | more
```

```
                          - 1 -

          Dear Mr. Madison:

               The purpose of this letter is to confirm
          Monday's  agreement.  Tupelo  gets "Porkchop"
          Peterson,  you  get "Earthquake" Emerson, and
          we  get  "Rotunda"  Robinson.   Here  are the
          players'  names,  heights,  weights,  and new
          teams:

           Name                   Height  Weight    Team

           Emerson, Ezekiel R.     6'5"      273  Rochester
           Robinson, Charles F.    6'3"      287  Des Moines
           Peterson, Paul N.       5'9"      178  Tupelo

               You  will  be  getting  one of the finest
           players  in  the People's Football League.

               Your Friend and Mine, Bill
```

- This time **tbl** has taken over the task of formatting the tabular material in the middle of the letter, with the results shown.

## USING THE **tbl** PREPROCESSOR **tbl**

In the previous example, you used one **nroff** dot command (**.sp 1**), two **tbl** requests (**.TS** and **.TE**), and a number of symbols that describe the desired appearance of the table. Here is an instant replay of the commands and symbols you just used in the example:

| | |
|---|---|
| ***.TS*** | Start of table |
| ***center*** | Center the entire table between the side margins |
| ***;*** | End of layout for the entire table; start of layout for individual lines |
| ***c c c c*** | Center each of four headings over their columns (there could also be two, three, or a larger number of headings) |
| ***l n n l*** | Position the columns of information as indicated (in this case, left-justified, numeric, numeric, left-justified) |
| ***.*** | End of layout of individual lines; start of information to appear in the table |
| | [Column headings within the table, separated by tabs] |
| ***.sp 1*** | Leave one space (blank line) between the headings and the columns below (**nroff** command) |
| | [Actual contents of the table entered here, separated by tabs] |

```
.TE          End of table
```

## SURROUNDING THE TABLE WITH LINES **box**

Another feature of **tbl** is the ability to surround a table with lines. to do this, all you have to do is add **box** to the second line:

```
.TS                              [Start of table]
box center ;                     [Box and center the table]
c c c c                          [Center each heading]
l n n l .                        [Allow four columns*]
Name Height Weight Team          [These are the headings, separated by
                                  tabs                                ]
.sp 1                            [Leave a space after the headings]
Emerson, Ezekiel R.      6'5"    273     New Rochester
Roginson, Charles F.     6'3"    287     Des Moines
Peterson, Paul N.        5'9"    178     Tupelo
.TE                              [End of table]
```

Now you can execute another command line, with the results shown:

```
$ mm -t pfl.deal | col | more
```

```
                                - 1 -

          Dear Mr. Madison:

               The purpose of this letter is to confirm
          Monday's  agreement.  Tupelo  gets  "Porkchop"
          Peterson,  you  get  "Earthquake" Emerson, and
          we  get  "Rotunda"  Robinson.   Here  are the
          players'  names,  heights,  weights,  and new
          teams:

          ------------------------------------------------
         | Name                 Height  Weight    Team     |
         |                                                 |
         | Emerson, Ezekiel R.   6'5"     273  Rochester   |
         | Robinson, Charles F.  6'3"     287  Des Moines  |
         | Peterson, Paul N.     5'9"     178  Tupelo      |
          ------------------------------------------------

               You  will  be  getting  one of the finest
           players  in  the People's Football League.

               Your Friend and Mine, Bill
```

If you would like to surround not only the table as a whole but also each individual item with lines, use **allbox** instead of **box** on the second line:

```
.TS                                       [Start of table]
allbox center ;                           [Box all and center the table]
c c c c                                   [Center each heading]
l n n l .                                 [Allow four columns*]
Name Height Weight Team                   [These are the headings, separated by
                                           tabs]
.sp 1                                     [Leave a space after the headings]
Emerson, Ezekiel R.     6'5"    273     New Rochester
Roginson, Charles F.    6'3"    287     Des Moines
Peterson, Paul N.       5'9"    178     Tupelo
.TE                                       [End of table]
```

Now you can execute the command line again, with the results shown:

```
$ mm -t pfl.deal | col | more
```

```
                              - 1 -

        Dear Mr. Madison:

             The purpose of this letter is to confirm
        Monday's  agreement.  Tupelo  gets "Porkchop"
        Peterson,  you  get "Earthquake" Emerson, and
        we  get  "Rotunda"  Robinson.   Here  are the
        players'  names,  heights,  weights,  and new
        teams:

        ------------------------------------------------
       | Name                 |Height |Weight|   Team     |
       |----------------------|-------|------|------------|
       | Emerson, Ezekiel R.  | 6'5"  |  273 | Rochester  |
       |----------------------|-------|------|------------|
       | Robinson, Charles F. | 6'3"  |  287 | Des Moines |
       |----------------------|-------|------|------------|
       | Peterson, Paul N.    | 5'9"  |  178 | Tupelo     |
        ------------------------------------------------

             You  will  be  getting  one of the finest
         players  in  the People's Football League.

             Your Friend and Mine, Bill
```

The **allbox** option has at least three drawbacks: 1) it draws too many lines; 2) it draws the lines too tightly around the individual items; and 3) it has a tendency to misalign the sides of the table.

## Using blocks in a table

Another useful feature of **tbl** is the ability to include a block of text within a table, as you would do in presenting definitions or descriptions. (You can use this feature with or without surrounding lines, of course). All you have to do is type the item to be defined or described, press the TAB key, then bracket each block like this:

```
Item  TAB  T\{
Type your block of text between a pair
of braces, each preceded by a capital T.
T\}
```

The following is an example in a complete table:

1. Begin a session with **vi**:

   - Start with a file called **programs**:

     ```
     $ vi programs
     ```

   - Type ***a*** and enter the following text:

     ```
     .TS                       [Start of table]
     center;                   [Center the table on the page]
     c s                       [Center and span the title across all
                                columns]
     c c                       [Center each heading over its col-
                                umn]
     l lw(4i).                 [Display each column flush left; al-
                                low four inches for the column on
                                the right (the description column)]
     Formatting Programs            [Title]
     .sp 2
     Program  TAB   Description [Column headings]
     .sp

     tbl   TAB   T{
     Use this preprocessor to produce tabular material
     in a variety of formats, with or without
     surrounding lines.
     T}
     .sp
     neqn  TAB   T}
     Use this nroff preprocessor to produce mathematical
     equations either on a single line or with vertical
     spacing.
     T}
     .TE            ESC
     ```

2. Execute an **mm** command line:

```
$ mm -t programs | more

                        - 1 -

                 Formatting Programs


Program                         Description

tbl         Use this preprocessor to produce tabular
            material  in a variety of formats,  with
            or without surrounding line.

neqn        Use  this nroff preprocessor to  produce
            mathematical   equations  either  on   a
            single line or with vertical spacing.
```

3. If you prefer surrounding lines, you can insert **box** in front of **center** on the second line, then execute another **mm** command line:

```
$ mm -t programs | col | more

                          - 1 -
 ----------------------------------------------------
|                Formatting Programs                 |
|                                                    |
|                                                    |
|  Program                         Description       |
|                                                    |
|  tbl        Use this preprocessor to produce tabular |
|             material  in a variety of formats,  with |
|             or without surrounding line.             |
|                                                    |
|  neqn       Use  this nroff preprocessor to  produce |
|             mathematical   equations  either  on   a |
|             single line or with vertical spacing.    |
 ----------------------------------------------------
```

The problem here is that **tbl** surrounds the title as well as the table itself, which makes the whole thing look a little odd.

## 20.3 Formatting equations

Subject to the limitations of your printer, you can produce equations of considerable complexity with the **neqn** preprocessor. (The laser printer and typesetting version for **troff** is called **eqn**, which is also capable of producing mathematical symbols and Greek letters.)

## Introductory example

The following example shows you what is possible with **neqn**:

1. Begin an editing session with **vi**:

□ Start with a new document called math:

```
$ vi math
```

□ Type ***a*** to append, then enter the following:

```
.P
Here is an equation that contains a fraction:
.EQ          [Start of equation]
a - b over a + b  =  c - d over c + d
.EN          [End of equation]

.P
This equation includes an exponent:
.EQ I
y  =  ax sup 2 + b
.EN

.P
This equation contains three subscripts:
.EQ I
a sub 2 x + b sub 2 y ^ = ^ c sub 2
.EN

.P
This equation contains radicals:
.EQ L
sqrt ab ~ = ~ sqrt a sqrt b
.EN
```

□ Store the text (***:w***) and return to the shell prompt (***:q***).

2. Format the text on your screen with **nroff**:

```
$ mm -e math | col | more
    or
$ neqn math | nroff -cm | col | more
```

Here is an equation that contains a fraction:

$$\frac{a-b}{a+b}=\frac{c-d}{c+d}$$

This equation includes an exponent:

$$y = ax^2 + b$$

This equation contains three subscripts:

$$a_2x + b_2y = c_2$$

This equation contains radicals:

$$\sqrt{ab} = \sqrt{a}\sqrt{b}$$

## Using the **neqn** preprocessor **neqn**

Without getting too involved, these examples illustrate many of the most commonly used mathematical expressions (fractions, exponents, subscripts, and radicals). Here is a quick summary of the words used in the examples:

**.EQ** Start of equation (centered)

**.EQ I** Start of equation (indented)

**.EQ L** Start of equation (flush left)

**over** Form a fraction, using the preceding expression as the numerator and the following expression as the denominator

**sup** Make the expression that follows an exponent

**sub** Make the expression that follows a subscript

**sqrt** Place the expression that follows inside a radical

**^** Leave an extra half space

**~** Leave an extra full space

**.EN** End of equation

## Bracketing expressions

To clarify ambiguity, bracket expressions with braces. For example, consider the following two expressions, one with braces and one without. The results on the right show the different ways they are interpreted:

```
a sup x + y                     a^x + y

a sup {x + y}                   a^{x + y}
```

$a^{x} + y$

$a^{x + y}$

The second expression clearly indicates that the exponent is **x + y**, not just **x**. (To have braces printed as braces, surround them with double quotes.)

## Defining expressions

You can define an expression with the **define** statement, then recall the definition in subsequent equations. For example, the following sequence shows how you could use **X** to represent a much longer expression:

```
.EQ
define X '{a sup 2 - b sup 2} over {a sup 2 + b sup 2}'
f(x) = X
.EN
```

With the definition in place, **neqn** will substitute the longer expression for **X** in the equation that follows. Note that the expression that provides the definition must be typed between single quotes. You can also use this feature to redefine keywords. For example, suppose you prefer to type ** instead of **sup** to begin exponents. Then you could enter a sequence like this:

```
.EQ
define ** 'sup'
a ** 2 + b ** 2 = c ** 2
.EN
```

## Aligning equations

If you have a series of related equations, you can line up all the equal signs on the same column, provided that (1) the first equation not be shorter than the others and (2) all equations be either flush left or indented. Use **mark** to locate the column in the first equation; then use **lineup** in subsequent equations, as shown here:

**Input**

```
.EQ I
y + a sup 2 + 2ab + b sup 2 mark = x
.EN
.EQ I
y + (a + b) sup 2  lineup = x
.EN
.EQ I
y  lineup = x - (a + b) sup 2
.EN
```

**Output**

```
      2
y + a  + 2ab + b   = x

                2
y + (a + b)        = x
                               2
y                  = x - (a + b)
```

## Other features

The **eqn** preprocessor is used with **troff** for laser printers and phototypesetters. It can handle all the features of **neqn**, as well as limits, summation and integral signs, vectors, matrices, tall brackets, and a variety of diacritical marks. If you use any of these things, refer to the *UNIX User's Manual.*

## 20.4 Defining your own requests

Although **mm** gives you a lot of flexibility in formatting text, there may be times when you will want to custom-design some formatting requests of your own. You can type a definition for your request, then enter that request just the way you would enter a standard request. Before discussing definitions, let's go over the units of measure available in **nroff** and **troff**.

### Units of measure

The **nroff** formatter has a resolution of 240 dots per inch. So the basic unit of measure for **nroff** is 1/240 of an inch, denoted **u**. From this basic unit, all other units are derived. When entering **nroff** requests, you are allowed to specify distances on the page either in common units (centimeters, inches) or printer's units (points, ens, ems, picas). See Table 20.1.

TABLE 20.1. Units of Measure in **nroff** and **troff**

| Unit | Abbre-<br>viation | **nroff** Distance<br>(Basic Units) | **troff** Distance<br>(if Different) |
|---|---|---|---|
| Basic unit | u | 1 | |
| Vertical line space | v | v | Current line spacing |
| Point (1/72 inch) | p | 6 | |
| en | n | C | Width of letter n |
| em | m | C | Width of letter m |
| Pica (1/6 inch) | P | 40 | |
| Centimeter | c | 94.5 | |
| Inch | i | 240 | |

C is either 20 (1/12 inch) or 24 (1/10 inch), depending on the output device; v is the height of each line, usually 1/6 inch. In **nroff**, m and n are equivalent, and indicate one character column (C). Using this information, you could use expressions like the following in **nroff** and **troff** requests:

| | | | |
|---|---|---|---|
| 4c | Four centimeters | 2i | Two inches |
| 9p | Nine points | 3P | Three picas |
| 2m | Two ems | 2n | Two ens |

### Defining a simple macro

To define your own macro request, begin near the top of the file with the **.de** request, followed by a one- or two-character name. Then type the lines of the definition, ended by a pair of dots on a separate line. To avoid confusion with **nroff** requests, it's usually best to use capitals letters and avoid names already used by **mm** or **nroff**. Finally, you can use \" to enter comments. Here's an example:

```
.de PA          \" Define a macro called .PA to begin a
                \"   new paragraph
.sp             \" Leave a blank line above the paragraph
.ti +5m         \" Indent 5 ems (columns) for this par.
..              \" This is the end of the definition
```

With this definition near the beginning of your document (before any text), you can start using **.PA** in your document as a request. When you enter **.PA** above a paragraph, **nroff** will leave a blank and indent 5 columns before printing the text. To **nroff**, **.PA** is just as valid as the **mm .P** request. Note that there is no dot in front of "PA" in the definition itself.

## Defining a pair of macros

Many of the **mm** macros occur in pairs, such as **.DS** and **.DE**, which are used to begin and end a display. You can define your macros this way, too. Here's an example:

Start of Verse

```
.de VS          \" Define a macro called .VS to display
                \"    lines of verse
.sp             \" Leave a blank line before first line
.nf             \" Turn off filling of text
.in +0.2i       \" Indent 1/5 inch from the left margin
.ll -0.2i       \" Indent 1/5 inch from the right margin
..              \" End of the definition of macro .VS
```

End of Verse

```
.de VE          \" Define a macro called .VE to end .VS
.sp             \" Leave a blank line after the last line
.fi             \" Restore filling of text
.in -0.2i       \" Restore the previous left margin
.ll +0.2i       \" Restore the previous right margin
..              \" End of the definition of macro .VE
```

With these definitions placed near the beginning of your document, you could then use this pair of macros to produce results like the following:

**Input** **Output**

```
.P
This is what is written        This is what is written in
in the book:                   the book:
.VS
When they saw the star,          When they saw the star,
they rejoiced with                 they rejoiced with
exceeding great joy.               exceeding great joy.
.VE
.P                               These are the words that are
These are the words              recorded.
that are recorded.
```

## Handling page transitions **.wh**

There is no mechanism built into **nroff** or **troff** to handle the top or bottom of a page. You have to provide for these manually, using one macro to take care of the top of the page, one to take care of the bottom, and a pair of **.wh** (where) requests to activate them.

The **.wh** request, which indicates where a given macro is to be executed on each page, is said to set a *trap.* Each **.wh** request provides a vertical distance in any suitable units and the name of a macro. On **.wh**'s scale, zero is the top of the page, positive numbers are measured from the top, and negative numbers are measured from the bottom. Here is an example of a pair of **.wh** requests:

```
.wh 0 HE       \" Activate macro HE at the top of each page
.wh -1i FO     \" Activate macro FO one inch from the bot-
tom of each page
```

The two macros, in this example named HE (header) and FO (footer), can be as simple or as elaborate as you choose to make them. Here is a pair of simple definitions for HE and FO that do no more than leave one inch at the top and bottom of each page:

```
.de HE         \" Define a macro called .HE for the top
               \"     of each page
.sp 1i         \" Leave one inch of blank space at the
               \"     top of the page
..             \" End of the definition of macro .HE

.de FO         \" Define a macro called .FO for the
               \"     bottom of each page
.bp            \" Break to a new page
..             \" End of the definition of macro .FO
```

Another pair of macros is given below.

## Inserting header and footer text **.tl**

To enter text into a header or footer, you can use the **.tl** (title line) request, which allows you to place something in the left, center, and right position. Here is the general form of the **.tl** request:

```
.tl 'left'center'right'
```

You can either enter or omit text for any of the three positions, depending on where you want the text to appear on the page. Within a **.tl** request, you can use a single percent symbol (**%**) to represent the current page number. Here are examples of title lines in **nroff** and **troff**:

```
.tl 'Company Use''Confidential'
.tl ''%''
```

The first will print "Company Use" on the left and "Confidential" on the right. The second will print the current page number in the center. Now we'll use these title lines in our header and footer macros:

```
.de HE          \" Define a macro called .HE for the top
                \"     of each page
.sp 2           \" Leave two lines of blank space above
                \" the header
.tl 'Company Use''Confidential'
.sp 3           \" Leave three lines of blank space below
                  the header
..              \" End of the definition of macro .HE

.de FO          \" Define a macro called .FO for the
                \"     bottom of each page
.sp 3           \" Leave three lines of blank space above
                \" the footer
.tl ''%''
.bp             \" Break to a new page
..              \" End of the definition of macro .FO
```

## 20.5 Modifying formatting options

The **mm** macro package is designed to offer you a complete formatting tool that is ready to use. However, if its preset choices don't meet your requirements, you can alter the operation of **mm** by changing its *number registers* and *strings*. For example, paragraph indentation is five spaces by default; if you would like ten spaces instead, you can change the appropriate number register (**Pi**) with a **.nr** request like this:

***.nr Pi 10*** [Change paragraph indentation to ten spaces]

In Appendix G, "Summary of Formatting Options," you will find most of the number registers and strings available to you, listed in approximately the order in which the topics were discussed in Part IV. The default setting is shown in *italic*, with alternate settings shown in a few cases.

## Modifying number registers

You have three **nroff** requests for dealing with number registers: **.nr** (number register), **.af** (assign format), and **.rr** (remove register). Use **.nr** to assign a value to a number register. For example, to set the **mm** number register **Pi** (paragraph indentation) to ten spaces, enter

| | |
|---|---|
| ***.nr Pi 10*** | [Set **mm** paragraph indentation to ten spaces] |

To specify the numbering sequence for a number register, use **.af**. For example, to use upper case letters to number **mm** figures, enter

| | |
|---|---|
| ***.af Fg A*** | [Use upper case letters to number **mm** figures] |

The options for **.af** are shown here:

```
        ⎧ 1     Plain arabic: 0, 1, 2, 3,... (default)
        ⎪ 001   Zero arabic: 000, 001, 002, 003,...
.af NR  ⎨ i     Lower case roman: 0, i, ii, iii,...
        ⎪ I     Upper case roman: 0, I, II, III,...
        ⎪ a     Lower case alphabetic: 0, a, b, c,...
        ⎩ A     Upper case alphabetic: 0, A, B, C,...
```

To remove a number register, thereby allowing more internal memory space for existing number registers, use the **.rr** request. For example, to remove the **mm** exhibit counter, enter this:

| | |
|---|---|
| ***.nr Ex*** | [Remove the **mm** exhibit counter] |

To name a number register in **nroff**, precede its name with **n** (single-letter name) or **n(** (two-letter name). For example, to name **mm**'s table counter in **nroff**, type ***\n (Ec***.

To name a number register in **mm**, you must use extra escape characters (\). For example, here is how to name two strings in **mm**:

| | |
|---|---|
| `\\\\*W` | [Name the width of the page in **mm**] |
| `\\\\*(DT` | [Name the date in **mm**] |

## Modifying strings

To assign a value to a string, use the **.ds** (define string) request. Since the **nroff** strings shown in this appendix are all predefined, this request can be used only for **mm** strings. For example, to change the bullet character to an asterisk, enter this:

| | |
|---|---|
| ***.ds BU **** | [Change the bullet character to an asterisk] |

To change the heading for the list of figures from `LIST OF FIGURES` to `Illustrations`, enter this:

```
.ds Lf Illustrations    [Use this heading for mm's list of figures]
```

To name a string in **nroff**, precede its name with * (single-letter name) or *( (two-letter name). For example, to name the **mm** footnote numberer in **nroff**, type ***F**; but to name the **mm** indent list, type ***(Ci**.

To name a string in **mm**, you must use extra escape characters (\). For example, here is how to name the two strings just mentioned above in **mm**:

```
\\\\*F             [Name the footnote number in mm]
\\\\*(Ci           [Name the indent list in mm]
```

### Additional information

For further information on modifying the operation of the formatting programs, refer to Appendix G, "Summary of Formatting Options," which lists most of the number registers and strings used by **mm** and **nroff**.

## 20.6 Summary

In this chapter you learned techniques for printing in double-column format with **mm**, formatting tables with **tbl**, formatting equations with **neqn**, and defining your own macro requests.

### Using double-column format

To print text in two columns side by side, precede the text with **.2C**. Then execute a command line like one of these:

```
$ mm -c -Tlp file | lp
$ nroff -cm -Tlp file | col | lp
```

To specify formatting for footnotes and displays, including tables and equations, use the **.WC** request.

### Formatting tables

To format a table, place **.TS** above the headings and information and **.TE** below, with the appropriate descriptions typed between **.TS** and the body of the table. Then execute a command like one of these:

```
$ mm -t -c -Tlp | lp
$ tbl file | nroff -cm | col | lp
```

## FORMATTING EQUATIONS

To format an equation, place **.EQ** above the mathematical expressions and **.EN** below. Then execute a command like one of these:

```
$ mm -e -c -Tlp | lp
$ neqn /usr/pub/eqnchar file | nroff -cm | col | lp
```

## WRITING MACROS

To write a custom macro request, begin the definition with **.de** (followed by the name of the macro), type the **nroff** requests that perform the desired functions (using \" for comments), then end the definition with **..** on a line by itself. Now you can begin using the macro in your document.

## ADDING HEADERS AND FOOTERS

To include headers and footers to a document, you must have a macro for the header, a macro for the footer, and a pair of **.wh** (where) requests to activate them. The macros can contain **.tl** (title line) requests to place text left, center, or right on the page, with apostrophes separating them. Use a percent symbol (**%**) for the page number in a title line.

## MODIFYING THE FORMATTING PROGRAMS

To change the value of one of the **mm** or **nroff** number registers, use the **.nr** request; to change the value of one of the **mm** strings, use the **.ds** request. Number registers and strings are listed in Appendix G, "Summary of Formatting Options."

## FOR FURTHER READING

If you'd like to study UNIX text-formatting in greater depth, refer to the following:

Birns, Peter; Patrick Brown, and John C. C. Muster, *UNIX for People*, Englewood Cliffs, NJ: Prentice Hall, 1985

Dougherty, Dale and O'Reilly, Tim, *UNIX Text Processing*, Indianapolis, IN: Hayden Books, 1987

Krieger, Morris, *Word Processing on the UNIX System*, New York: McGraw-Hill, 1985

Roddy, Kevin P., *UNIX NROFF/TROFF: A User's Guide*, New York: Holt, Rhinehart, Winston, 1987

Emerson, Sandra L., and Karen Paulsell, *troff Typesetting for UNIX Systems*, Englewood Cliffs, NJ: Prentice-Hall, 1987

Ossanna, Joseph F., *Nroff/Troff User's Manual*, Murray Hill, NJ: Bell Laboratories, 1976

# Part V

# Shell Programming

In Part V, you will learn some short-cuts for using UNIX, and you will learn how to use the UNIX shell as a programming language. Most System V installations actually include two shells, the Bourne shell and the C shell. The Bourne shell, which is the original UNIX shell, is the faster of the two; the C shell, a later contribution from Berkeley (and not officially part of System V), is the more versatile. The C shell offers a feature that allows you to recall a previous command line, modify it, then execute it again. Borrowing from the C language, the C shell also offers string and numeric arrays, with access to individual elements. Historically, the Bourne shell, the C shell, and the C language all have their roots in ALGOL, which accounts for the many similarities that all three share. A newer development, the Korn shell, combines many of the best features of both the Bourne shell and the C shell, and may supersede them some time in the next few years. (While a few of the sample programs presented in Part V can be put to use in a working environment, the main purpose of these programs is to illustrate features of the shell.)

# 21

# Introduction to the Bourne Shell

## 21.1 Introductory example

### Making a command easier to use

People who are new to UNIX often complain that many of the commands have cryptic names and are difficult to use. For example, to change the name of a file in your current directory from **past** to **future**, you have to enter the following command line:

```
$ mv past future
$ _
```

One problem with this is that the name **mv** suggests "move" not "rename." Another problem is remembering which comes first, the old name or the new name. For a new UNIX user, it would be better to have a command called **rename** that leads the user through the process with an interactive dialogue, like this:

```
$ rename
Old name: past
New name: future
File past is now called future
$ _
```

### Writing a shell program

Shell programming allows you to do this. By writing a simple six-line *shell program* (also known as a *shell script* or a *shell file*), you can create a new command called **rename** that works just like this. All you need is one command to place prompts on the screen (**echo**), a command to receive input from the user (**read**), and the UNIX command that will carry out the user's request (**mv**), as shown here as the contents of a file called **rename**:

```
$ cat rename
echo "Old name: \c"                [Display prompt—no line feed]
```

```
read old                            [Assign response to variable old]
echo "New name: \c"                 [Display prompt—no line feed]
read new                            [Assign response to variable new]
mv $old $new                        [Rename, using the names entered]
echo "File $old is now called $new \n"
                                    [Inform the user]
$ _
```

## Shell programming features

This simple example illustrates a few of the things you can do with a shell program:

- Display prompts on the screen.
- Receive input from the user.
- Assign strings to variables.
- Use assigned variables in UNIX commands.

In addition, there are a number of other things you can also do with a shell program, which you will learn in Part V:

- Redirect input and output.
- Connect processes with pipelines.
- Execute commands conditionally.
- Assign numbers to variables.
- Perform arithmetic on numeric variables.
- Substitute variables conditionally.
- Manipulate a command's arguments.
- Construct programming loops.
- Test files, compare strings, and compare quantities.
- Perform multi-way branching.
- Handle system interrupts.

### Allowing execution

One final note before beginning these topics: To make the file **rename** an executable command, you have to grant read and execute permission to all users with a command like this:

```
$ chmod +rx rename     [Add read and execute permision for everyone]
$ _
```

## 21.2 Controlling the environment

Until now you have known the UNIX shell as the command interpreter. As the command interpreter, the shell, which itself is not part of the operating system, provides an interface between you and the kernel by executing commands entered at the terminal. As you have just seen from the introductory example, the shell also serves as a programming language, which allows you to control and reshape the user interface. We'll begin our discussion of shell programming with those variables that control the user's working environment.

### The shell start-up file *.profile*

Your home directory probably contains a file that provides the shell with default values for initialization. This file, which must be called **.profile**, contains information used by the shell to set up your working environment, such as your terminal type, the prompts that will appear on your screen, the names of the UNIX directories that contain the commands you use, the name of the file that contains your electronic mail, the protection that a new file will receive when created, and settings for your terminal.

Whenever you log in, if your home directory has a file named **.profile**, the shell will execute this file before issuing you a prompt. If there is no **.profile** file, the shell will use its own default values for variables. Here are the contents of a typical **.profile** file:

```
PATH=/bin:/usr/bin:$HOME/bin::.            [Set the command search path]
MAIL=/usr/spool/mail/`basename $HOME`      [Set your mail box's path name]
TERM=vt100                                 [Set the terminal type]
PS1='% '                                   [Set the primary prompt]
PS2='> '                                   [Set the secondary prompt]
umask 022                                  [Set file creation mask]
: set tty                                  [Set information about your terminal:
stty erase "^H" -tabs                       (CTRL-H) to erase; replace tabs with
stty cr0 ff0                                spaces when printing; no delay for car-
export HOME MAIL PATH TERM                  riage return or form feed; export four
                                            shell variables—see below]
```

### Shell variables

A system administrator typically creates **.profile** when adding a new user's account to the system, and in the process defines variables such as **HOME**, **MAIL**, **PATH**, **TERM**, **PS1**, and **PS2**. However, you can edit **.profile** with a text editor and redefine any of these variables as needed. You will find a summary of the variables in Tables 21.1 and 21.2, with detailed descriptions following.

TABLE 21.1. Shell variables

| Variable | Description |
|---|---|
| HOME | *Login Directory*—The login program initializes HOME to the name of the directory that becomes your current directory after you log in. Then any time you execute the **cd** command without an argument, the shell automatically returns you to this directory. |
| MAIL | *Mail File*—When this variable is defined, it tells the shell the name of the file that serves as your mail file. |
| PATH | *Command File Search Path*—This variable is used by the shell to search through a sequence of directories for a command whenever you supply a partial pathname. **PATH** is assigned by entering a list of filenames, separated by colons. Its default value is<br>`: /bin : /usr/bin`<br>which means, "First look in /bin in your current directory, then look in /usr/bin." |
| TERM | *Terminal Type*—Use this variable to assign the type of terminal you are using (required by any program that employs a full-screen interface). |

## 21.3 Setting variables

### Home directory **HOME**

Generally, the **HOME** variable should not be changed. However, in some instances after login, you may want to reinitialize it so you that you can operate from a given working directory without retyping a long pathname, as shown here:

```
$ HOME=/usr/jim/project/planning/report
$ _
```

Recall that the **cd** command without arguments returns you to your home directory (the value of variable **HOME**). So if you set **HOME** as shown in

TABLE 21.2. Variables Unique to the Bourne Shell

| Variable | Description |
|---|---|
| PS1 | *Primary Prompt*—Use this variable to assign the prompt to be used by the shell when it is ready to accept input. |
| PS2 | *Secondary Prompt*—Use this variable to assign the prompt to be used by a subshell to receive additional input. |
| IFS | *Internal Field Separator*—Use this variable to assign a character (in addition to SPACE and TAB) to be used for separating fields from each other. |
| TZ | *Time Zone*—This variable, mainly for the benefit of the system administrator, indicates the geographic time zone of the computer installation. |
| LOGNAME | *Login Name*—This variable identifies the user currently running the shell. |

the previous example, this will have the same effect as if you had typed this:

```
$ cd /usr/jim/project/planning/report
$ _
```

## Terminal type **TERM**

Set the terminal variable **TERM** to obtain the proper display on your terminal. The value assigned to **TERM** must be one of the terminal names given in either /etc/termcap or /etc/terminfo. You can set the **TERM**, **PATH**, and **MAIL** variables the same way you set the prompts, as shown here:

```
$ PATH=:/bin:/usr/bin:$HOME/bin
$ TERM=vt100
$ MAIL=/usr/spool/mail/'basename $HOME'
$ _
```

## The primary prompt **PS1**

When the shell is ready to accept a command from your terminal, it issues a prompt. The dollar sign ($) is the default prompt. You can make your own prompt by changing the value of the **PS1** variable. This would be handy if you are accustomed to using several UNIX systems from the same terminal. Here's an example:

```
$ PS1='V: '          [Set the primary prompt to V: —with a space]
V: _                 [This is your new primary prompt]
```

Note the space that follows `V:`. This separates the prompt from your command line. From now on until you sign off from the system or reassign the value of **PS1**, your primary prompt will be `V:` instead of the dollar sign (`$`). For the sake of clarity, we will use the dollar sign as the primary prompt throughout this chapter.

## The secondary prompt **PS2**

The **PS2** variable is used by the shell to inform you that it expects further input. The default value for **PS2** is the greater than sign (>). Here is an example:

```
$ (
> cd /bin
> ls -l
> )
$ _
```

The shell interprets **(** as the command grouping symbol. Then the shell creates a subshell, which issues the > prompt to indicate that it is reading and will not execute any commands until you inform the shell that you finished your entry by typing **)**. If you accidentally press the wrong key and get the secondary prompt, the DEL key will get you out, and you will see the primary prompt again. The above example is used to illustrate one of the cases when the shell issues its secondary prompt **PS2**. Command grouping is discussed in greater detail in Chapter 22, "Bourne Shell Processes." You can change the secondary prompt the same way you change the primary prompt:

```
$ PS2='+ '          [Set the secondary prompt to +]
$ _
```

Your shell secondary prompt now is set to + .

## Internal field separator **IFS**

You may recall from Chapter 14, "Programming with **awk**," that the **awk** program allows you to set your own field separator (in addition to SPACE and TAB). The Bourne shell also allows this. The following example allows you to use a vertical bar ( | ) to separate fields:

```
$ IFS=|
$ _
```

Once you've issued this command, the following two command lines are now identical:

```
$ cp file_1 file_2 store
```

```
$ cp file_1|file_2|store
```

## Time zone **TZ**

With this variable you can assign the three-letter name of your time zone, the number of hours by which your time zone differs from Greenwich Mean Time (GMT), and (optionally) the three-letter name of your daylight saving time zone. The following command could be used to set **TZ** for the states along the Pacific Coast of the United States:

```
$ TZ=PST8PDT
$ _
```

This sets the standard time to PST (Pacific Standard Time), the time differential to 8 hours, and daylight saving time to PDT (Pacific Daylight Time).

## Exporting shell variables

As shown earlier in this chapter, the **TERM, PATH, MAIL, PS1**, and **PS2** variables, along with the **stty** command are defined in **.profile**, making it unnecessary for you to have to set them every time you log in. Normally, these variables should be *exported*, so that any *child processes* can inherit them. (Child processes will be explained later in this chapter.)

Should the exported variable be modified in a child process, these new values will take effect within this child process only, and the previous value will be automatically restored when the child process returns to its parent process. Place the **export** command at the end of your **.profile** file, as shown earlier in this chapter. You can ask the shell to display all the exported variables by typing **export**, as in this example:

```
$ export
MAIL HOME PATH TERM
$ _
```

You can also ask the shell to display the values of shell variables with the **echo** command (**echo** is explained in Chapter 23, "Bourne Shell Variables"):

```
$ echo $HOME
/usr/joe/workdir
$ _
```

## 21.4 Commands and arguments

As the command interpreter, the shell reads command lines entered at your terminal or from a file, interprets the first word as the command name, and any remaining words as arguments. Then, using the command name, the shell searches for the command file, invokes it, and passes all pertinent arguments to it. Arguments that have special meaning to the shell (such as &, |, >>, >, $, *) will not be passed to the command program, but will be retained for interpretation by the shell itself. These special symbols, called *metacharacters*, are discussed later in this chapter. Let's examine the following example:

```
$ mv file.1 file.2 &
$ _
```

The shell seaches for a command file named **mv**, invokes **mv**, passes **file.1** and **file.2** to **mv** as arguments, but retains & for its own use, interpreting & as a request for execution of this command line as a background process. (Background processes are explained in detail later in this section.) Some commands require only one argument, some require more (as in the previous example), and some do not require any arguments, as in this example:

```
$ pwd
/usr/jim/zebra
$ _
```

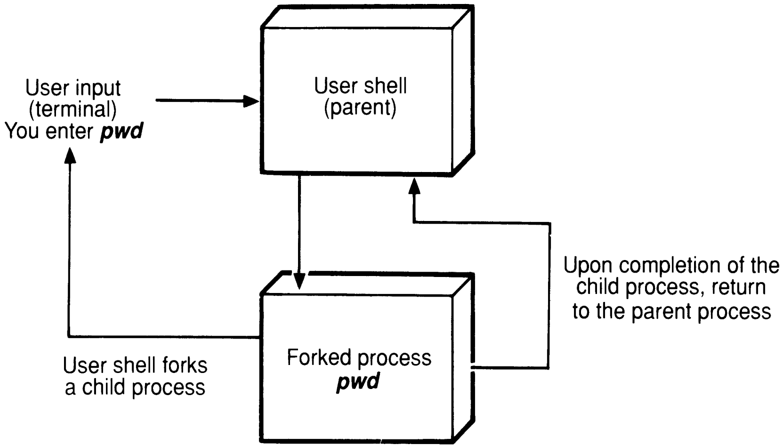
The **pwd** (print working directory) command displays the full pathname of your current directory. (Refer to the *UNIX User's Manual* for details about each command.) At any rate, even the simplest command requires at least one character, as in the case of the **ls** command, which lists all filenames in the current directory.

### How the shell invokes commands

When you log into UNIX, you are assigned a copy of the shell as the new *parent process.* This allows you to use all the resources of the operating system for which you have permission. From this parent shell process, the shell invokes a command file by creating a new duplicate shell process called a *child process.* The parent shell process then waits for this child process to complete its execution, provided that the command file is actually a compiled program. On the other hand, if the command file is a shell procedure, the shell will simply read the file and invoke the commands inside it. (Shell procedures are described in Chapter 23, "Bourne Shell Processes.") Figure 21.1 shows the relationship between parent and child processes.

The act of creating a duplicate process is called *forking*, and the child process is also called a *forked process.* A child process can fork to create another new process, thereby becoming a parent shell process of its own child, while its parent process becomes a *grandparent.*

FIGURE 21.1. An example of forking.



## How the shell finds commands

If you know the exact location of the command file, you can invoke it by giving the full pathname. If you do this, the shell will not search anywhere else to find it. For example, suppose there is a command named **greeting** in directory /usr/jim/print that displays this message `WELCOME TO UNIX`. You can then identify the location of this command to the shell explicitly:

```
$ /usr/jim/print/greeting
WELCOME TO UNIX
$ _
```

If you don't provide the shell with a full pathname, the shell will use either the value of the **PATH** variable defined in your .profile file or its own default value to perform a search for the command. As explained earlier in this chapter, **PATH** may be set with colons (:) separating the directories to be searched, like this:

```
$ PATH=/bin:/usr/bin::$HOME/bin:
```

If **PATH** in your .profile file is set like this, the shell will first attempt to locate **greeting** in the /bin directory. If the shell doesn't find **greeting** there, it will next search through /usr/bin. If the shell doesn't find **greeting** there, it will try your current directory (the extra colon). Finally, if the shell doesn't find **greeting** in your current directory, it will try the subdirectory of your home directory (**$HOME**) called /bin.

For example, if you want to find out who is currently logged on the system, and you are logged in as Jim, type

```
$ who
jim
$ _
```

In this case, the shell found **who** in directory /bin. By rearranging the names listed after the **PATH** variable, you can have the shell search from your current directory first by placing the **:** right after the equal sign:

```
$ PATH=:$HOME/bin:/bin:/usr/bin
```

With this arrangement, you can keep private copies of public commands or custom commands in your directory, even if your commands have the same names as public commands. It is best to keep all of your commands in one directory, so that you don't have to memorize many search paths when you want to invoke one of them. UNIX users usually keep private commands in their /bin (binary) subdirectory.

The name /bin is used because most files in this directory are object files, stored in binary form. The major bin directory (/bin) contains system commands, while the /usr/bin directory contains user utility commands. Any user can usually use commands in either of these directories. If your **PATH** is set as shown in the example above, the shell will first search in your current directory, then your subdirectory bin before searching through another bin.

When setting the **PATH** variable, be careful about the order in which you arrange the directories. If you aren't careful about this, you will cause the system to waste valuable time making inefficient and unnecessary searches. For example, consider a **PATH** setting like this:

```
$ PATH=:/usr/diag/bin:$HOME/bin:/usr/bin:/bin:
```

Every time you enter a command residing in /bin, the shell is forced to go through these six steps before it can find and execute the command:

(Your current directory)
/usr/diag/bin
/usr/$HOME/bin
/usr/bin
/bin

If your most frequently used commands reside in /bin and /user/bin, you place an unnecessary burden on the system unless you rearrange the setting for **PATH** more efficiently, as in this example:

```
$ PATH=/bin:/usr/bin::$HOME/bin:/usr/diag/bin:
```

Now the shell will search for commands in the following order, and the shell can locate your commands faster:

/bin
/usr/bin
(Your current directory)
$HOME/bin
/usr/diag/bin

## 21.5 Standard input, output, and diagnostics

Whenever a command begins execution, three files are opened, each associated with a number called a *file descriptor*:

- Standard input (file descriptor 0) from terminal or argument file
- Standard output (file descriptor 1) to terminal or file
- Standard diagnostic output (file descriptor 2) to terminal or file

In each case, the terminal is the default standard input or output. For example, suppose there is a file named **message** in your current directory that contains WELCOME TO UNIX. When you type

```
$ cat message
WELCOME TO UNIX
$ _
```

the **cat** command reads from its standard input (that is, the contents of file **message**), then prints out the contents to its standard output (that is, your terminal).

Any diagnostic output of **cat** is also directed to your terminal. If the file **message** is not in your current directory or you do not have read permission for **message**, an error message will be written to **cat**'s diagnostic output (that is, your terminal), as in this example:

```
$ cat message
cat: can't open message
$ _
```

Furthermore, If you don't provide a file argument for **cat**, **cat** will read what you type as its input, and then display the output on your screen:

```
$ cat
this is a test (RETURN)          [cat takes this input...]
(CTRL-D)                         [end of input]
this is a test                   [then displays this output on the
$ _                              screen]
```

You can also use the **cat** command to enter text into a file by directing its output to a file:

```
$ cat > text.file
I am using cat to create a little file. (RETURN)
This is the second line of the file. (RETURN)
(CTRL-D)
$ _
```

Now you have a file called **text.file**, whose contents become whatever you type at the keyboard. (Then (CTRL-D) terminates the text.)

## 21.6 Redirection of input and output

In the previous section you learned about standard input and output. There will be times when you may want to change the input or output of a command to something other than its standard input or output. You can do this with redirection, using one of these symbols:

| | |
|---|---|
| <*file* | Redirect command input from *file* |
| >*file* | Redirect command output to *file* |
| >>*file* | Append command output to *file* |

### Redirection of input

The default standard input of the **mail** command is what you type at your terminal. You can send an existing file instead of keystrokes as your message with the **mail** command by redirecting the input from that file, as in

```
$ mail jim < letter
$ _
```

Now the **mail** command will read file **letter** as its input and send it to Jim.

### Redirection of output

If you have permission to write to a device or a file, you can use one of them as the output for any command that sends its output to the standard output, as in this example:

```
$ cat message > file_1
$ _
```

In this example, the **cat** command reads the contents of file **message**, and writes its output to **file_1**. If **file_1** does not exist, it will be created by the shell; if **file_1** already exists, its contents will be overwritten. If you do not have permission to write to **file_1**, you will see a message like this:

```
$ cat message > file_1
cat: can't create file_1
$ _
```

You can also use **cat** to send a file to the printer, as in this example:

```
$ cat message > /dev/lp
$ _
```

Or you may want to send your file to another output port, as in this example:

```
$ cat message > /dev/tty05
$ _
```

In a shell procedure, you may need to create a empty file. You can do so with the redirection symbol, as shown in this example:

```
$ > empty.file    [Create a file called empty.file]
$ _
```

## Redirection of output with file descriptors

You can direct standard output or diagnostic output to different files or devices by adding the appropriate file descriptors, as in this example:

```
$ cat message 1>& file.out 2>& file.error
$ _
```

The contents of file **message** will go to **file.out**, while any error message that may arise will be sent to **file.error**. You may want to redirect both standard output and standard error output to the same file in this way:

```
$ cat message 1>&file.all 2>&1
$ _
```

You can append to the end of a file without overwriting its contents by using the double greater than symbol (>>) to add to the end of the file:

```
$ cat message > file.both
$ cat greeting >> file.both
$ _
```

Now **file.both** contains all the contents of file **message** plus all the contents of file **greeting**.

### A NOTE ON REDIRECTION

Only standard input and output of a command can be redirected. For example, the **date** command always consults the system, then displays the current date and time on your screen (that is, its input is not standard, and therefore it cannot be redirected). As another example, the output of the **lp** command is always directed to the printer port (that is, its output is not standard, and therefore it cannot be redirected). If you are not certain about a particular command, consult the *UNIX User Manual* before attempting to redirect its input or output.

## 21.7 Summary

In this chapter you learned some basic things about the Bourne shell. This included discussions of the shell start-up file (**.profile**); shell variables; commands and arguments; standard input, output, and diagnostics; and redirection of input and output.

### INTRODUCTION

This chapter begins with a discussion of the Bourne shell as a programming language. The shell start-up file (**.profile**), which resides in your home directory, is executed immediately after you log in. Its contents provide UNIX information about your terminal and sets your primary prompt, secondary prompt, command search path, the path name of your mail box, file creation mode, and various things that relate to the operation of your terminal.

Here are the shell variables commonly defined in **.profile**:

| Bourne and C Shells | | Bourne Shell Only | |
|---|---|---|---|
| **HOME** | Login directory | **PS1** | Primary prompt |
| **MAIL** | Mail file | **PS2** | Secondary prompt |
| **PATH** | Command file search path | **IFS** | Internal field separator |
| **TERM** | Terminal type | **TZ** | Time zone |
| **LOGNAME** | Login name | | |

To obtain the value of a variable, precede its name with a dollar sign.

## Setting shell variables

You can execute a command line after the UNIX shell prompt to set the value of any shell variable. This has the same effect as placing the same line in **.profile**. The process of extending the settings of shell variables to all *child processes* executed at your terminal is called *exporting* the variables. This is accomplished with the **export** command.

## Commands and arguments

A command line to be processed by the shell must contain a command name, and may contain arguments and special symbols called *metacharacters* that provide the shell with special instructions.

Whenever you log into UNIX, you are assigned a copy of the shell as the new *parent process.* When the shell invokes a compiled program, the shell creates a new process called a *child process* to execute the program. This is called *forking*, and the child process is also referred to as a *forked process.* When the shell executes a shell procedure, it simply reads the file and invokes the commands found in the file.

If you do not provide the shell with a full pathname, the shell will use the value of the **PATH** variable to search for the command. The value of **PATH** is a list of pathnames where commands may be found.

## Standard input, output, and diagnostics

Whenever the shell begins to execute a command, the shell opens three files, each associated with a number called a *file descriptor*:

| | |
|---|---|
| Standard input | (0) |
| Standard output | (1) |
| Standard diagnostic output | (2) |

You can use redirection to change input or output, using one of these symbols:

| | |
|---|---|
| < | Redirect command input from a file |
| > | Redirect command output to a file |
| >> | Append command output to a file |

To redirect selected command output, precede the redirection symbol with a file descriptor. Redirection is not possible with every UNIX command. Some cannot accept input and some do not produce output. Redirection is possible only with standard input or standard output.

# 22

# Bourne Shell Processes

In the previous chapter, you learned about some basic things in the Bourne shell: the start-up file (**.profile**); shell variables; command lines; standard input, output, and diagnostics; and redirection of input and output. In this chapter you will learn about shell functions, background commands, connecting processes, and giving directives to the shell.

## 22.1 Shell functions

At the beginning of Chapter 21 you learned a simple shell program named **rename**, which gave you a way to rename a file using screen prompts. In System V, Release 2.0 and later versions of UNIX, you can use a *shell function* to accomplish much the same result—but with better performance. Because the shell stores a function in main memory instead of in a disk file, executes a function without spawning a new shell, and performs parsing for a function in advance, it can execute a function faster than it can a shell script. One function can call another, but not itself.

### Command line execution

The simplest way to execute a function is to enter it and invoke it directly from the command line. To convert our example from Chapter 21 into a function, we could type the function itself, followed immediately by an invocation of the function, as follows:

```
$ rename ()               [Call the function rename]
  {                       [Use an opening brace to begin]
  echo "Old name: \c"
  read old
  echo "New name: \c"
  read new
  mv $old $new
  echo "File $old is now called $new \n"
  }                       [Use a closing brace to end]
$ rename                  [Invoke the function here]
Old name: past
New name: future
```

```
File past is now called future
$ _
```

The procedure is similar to the one you followed in Chapter 21, with two differences: this time you didn't have to enter the statements into a file and (since there was no file) you didn't have to make the file executable.

### LOGIN FILE DECLARATION

To make a function available to you each time you log in, you can place it in your login file (.profile) with your other initialization statements. Once you've done this, the function is ready to invoke as soon as you log in.

```
$ cat .profile
TERM=wyse50
stty erase "^H" -tabs
export TERM
  {
  echo "Old name: \c"
  read old
  echo "New name: \c"
  read new
  mv $old $new
  echo "File $old is now called $new \n"
  }
$ _
```

## 22.2 Background commands

To make it possible to run more than one process at your terminal, add an ampersand (&) after the last argument of your command string. This allows you to proceed immediately to another command without having to wait for the first one to be completed. For example:

```
$ cc program.c &
308
$ _
```

calls up the C compiler to compile the source code in file program.c in the background. The shell initiates this process, then the kernel assigns it a process number (308 in this example). The number of processes that you can have running at the same time is limited by the system. If you exceed this limit, a warning message will be sent to your terminal. The larger the number of processes running at any given time (foreground or background), the slower the system is able to respond.

Executing a command as a background process does not alter the command's inputs and outputs. So if the standard input of the background process is from your terminal, this input may be disrupted by anything you type for the current foreground process; if the standard output of the background process is to your terminal, this output will disturb the display of the current foreground process. To avoid this, it is best to redirect the input and output of background processes. To change the standard output of the background process, use the **nohup** (no hangup) command, as in this example:

```
$ nohup cmp file1 file2 &
320
Sending output to 'nohup.out'
$ _
```

The output of the **cmp** command will be sent to nohup.out. If nohup.out does not exist, it will be created by the shell; if it does exist, output will be appended to the end of nohup.out. If nohup.out in your current directory doesn't have write permission, the shell will direct output to nohup.out in your home directory ($HOME/nohup.out).

The **nohup** command makes your process immune to hangup and quit signals that could be caused by hanging up on a dial-up line. For this reason, **nohup** is best for keeping a background process running after you log out from a dial-up line.

## BACKGROUND PROCESSES **ps**

At some point you may want to know if your background processes have completed (or failed). To view the status of background processes, use the **ps** (process status) command. The **ps** command displays information about active processes, with several different options. When **ps** is executed without any options, **ps** displays only active processes associated with the current terminal, as shown in this example:

```
$ cc screen.c &
283
$ ps
   PID   TTY   TIME    COMMAND
   283   02    0:02    cc screen.c
   037   02    0:11    -sh
   290   02    0:01    -ps
$ _
```

The number assigned to your background process (283) appears in the first display line under the heading PID (process identifier), indicating that the background process you just invoked has started. The TTY column displays your current log-in port. The TIME column displays the cumulative execution time for the process. And, finally, in the COMMAND column you see displayed the commands currently in process. (Here, -sh represents the shell.) If you don't find your background process number on the list, it may be completed already.

When invoked with the **a** option, **ps** displays information on all active processes on the system, as shown in this example:

```
$ ps -a
  PID    TTY   TIME   COMMAND
   36   sysc   0:07   -sh
  386     02   0:02   -ps -a
   37     02   0:12   -sh
$ _
```

As you can see, your current port (02) has only two processes, ps -a and -sh, while the system console (sysc) is idle (-sh).

To obtain a more detailed display of all active processes, include the **l** option, as shown here:

```
$ ps -al
F S UID PID PPID   C PRI NI ADDR SZ WCHAN  TTY TIME COMD
1 S   0  36    1   0  28 20   67 16  5a20 sysc 0:07 -sh
1 S 203  37    1   0  30 20   49 16  6fd4   00 0:13 -sh
1 S   0 367    1   0  28 20   9b 16  193e   01 0:01 - 2
1 S   0 368    1   0  28 20   80 16  1970   02 0:01 - 2
1 S 203 283   37   0  30 20   96 44  703a   00 0:01 cc screen.c
1 R  12 421   37 117  59 20   66 28         00 0:08 ps -al
$ _
```

This gives you detailed information on system processes. In the paragraphs that follow, we discuss only those items that are within the scope of this book. (For other items, refer to the *UNIX User's Manual.*)

1. The F column displays the processes flag:

   01 indicates that the process is in core.
   02 indicates that this is the system process.
   04 indicates that the process is locked in core.
   10 indicates that the process is being swapped.
   20 indicates that the process is being traced by another process.

2. The S column displays the state of the process:

   S indicates that the process is sleeping.
   R indicates that the process is running.
   W means that the process is waiting for another process to finish.

3. The UID column displays the user ID. Your user ID is displayed for all processes invoked at your terminal. In some systems, the user name is displayed instead.

4. The PPID column displays the process ID of the parent process. As you can see, process ID 37 is the parent of processes 283 and 421.

5. The PRI column displays the priority of the process based on the value at NICE. A *higher* number means a *lower* priority.

6. The NI column, as mentioned above, displays the value that is used to compute the priority of process. You can increase this value by using the **nice** command, explained in the next section.

7. The SZ (SIZE) column displays the number of blocks used by the core image of the current process

8. The WCHAN column, which is related to the F column, displays the event for which the process is sleeping, as indicated by the S flag in the F column. WCHAN is blank if the process is active.

9. The TIME column displays the cumulative execution time of the process.

## PRIORITY OF PROCESSES **nice**

UNIX is a multi-user, multi-tasking system. That is, more than one user can be logged at a given moment and each user can invoke more than one process at a time. Since computer time must be shared among the processes that are active, any process that has a higher priority will receive more computer time. When processes are first invoked, they all begin with the same priority level. However, since this implies that each process receives equal attention from the system's main processor, it can often mean slower operation for the system as a whole. To avoid this, you can use the **nice** command to assign less important processes a lower scheduling priority. Here is an example:

```
$ nice cc screen.c &
212
$ ps -al
F S UID PID PPID   C PRI NI ADDR SZ WCHAN TTY TIME COMD
1 S 203  30    1   0  30 20 2044 16 6f4c sysc 0:09 sh
1 S   0  31    1   0  28 20 3549 16 190c   00 0:01 - 2
1 S 203 212   30  50  30 30 4d65 44 6ff6 sysc 0:01 cc screen.c
1 R  12 217   30 161  62 20 518e 28      sysc 0:09 ps -al
$ _
```

In the absence of an argument, **nice** increases the priority number of the process by 10 (which decreases its priority). That is why you see 30 in the third display line under NICE (the line for cc screen.c, with process ID 212). If you prefer a number other than 10, you can provide any number in the range of 1 to 19 as an argument to the **nice** command. For example, to increase the priority number for cc program.c by 12, use this command line:

```
$ nice -12 cc program.c &
$ _
```

This sets the NICE value to 32 (20 + 12). Only the super-user can assign to a process a lower **nice** value than 20 (resulting in a higher priority for the process), since the **nice** command allows the super-user to use negative arguments, as shown in this example:

```
# nice --10 cc screen.c &
285
# _
```

This assigns to process 285 a priority number of 10 (a higher priority).

## TERMINATING BACKGROUND PROCESSES **kill**

If you should ever have to terminate a background process (and you know its process ID), use the **kill** command, followed by the process ID as an argument. As you have learned, the shell displays the process ID at the time you invoke the background process. If you should forget the number, use the **ps** command to display it. Suppose you have a background process whose PID is 283 and you want to terminate it. You can execute this:

```
$ kill 283
$ _
```

You can then use the **ps** command to verify that process 283 is no longer displayed. If it still appears, then it hasn't stopped running. To make sure that a process is terminated, use the 9 option, as shown here:

```
$ kill -9 283
$ _
```

The kill signal generated by **kill -9** cannot be caught or ignored by the process indicated. If you use kill with the 9 option and the argument is either 0 or your login process number, you will log yourself off the system. Unless you have super-user status, you can kill only processes associated with your terminal.

## USING JOB CONTROL **shl**

The **shl** (shell layer) command of Release 2.0 allows you to execute up to seven different processes at the same, with more control than you would have with background execution alone. Each process is executed in a separate shell, which is called a *layer*. During session with the **shl** command, you can create, name, list, and delete layers. While several layers are active, you can switch from to another, making the new layer the *current layer*, where you can enter and initiate commands. To begin a job control session, execute the **shl** command; to end a session, enter **quit**. (or **q**).

Here is a simple example of a job control session:

```
$ shl                           [Begin the session]
>>> _                           [This is the prompt of the shell layer manager]
>>> help                        [First we'll display the commands available]
block name [name ...]           [Block output display of the layer(s) named]
create [name]                   [Create a new layer called name]
delete name [name ...]          [Delete the layer(s) named]
help or ?                       [Display this list of commands]
layers [-l] [name]              [Display information about the layer(s) named]
quit                            [End this session with the shell layer manager]
toggle                          [Switch to the previous layer]
resume [name]                   [Make the layer named the current layer]
unblock name [name ...]         [Restore output display of the layer(s) named]
>>> c compile                   [Create (c) a new layer called compile]
compile _                       [The name of a layer is also its prompt]

compile cc get.c                [Begin compiling a C source program called get.c]
compile CTRL-Z                  [Press CTRL-Z to return to the shell layer manager]
compile >>> c format            [Create (c) a second layer called format]
format _                        [Again, the name of a layer is also its prompt]

format mm text                  [Begin formatting a file called text]
format CTRL-Z                   [Press CTRL-Z to return to the shell layer manager]
format >>> b format             [Keep formatting output off the screen]
>>> c edit                      [Create (c) a third layer called edit]
edit _                          [Once again, the name of a layer is also its prompt]
edit vi memo                    [Begin editing a file called memo]
   .
   .
   .
edit CTRL-Z                     [Press CTRL-Z to return to the shell layer manager]
edit >>> layers                 [Display information about active layers]
compile
format
edit
>>> _                           [The shell layer manager prompt returns]
>>> d edit                [Delete the layer for editing (edit)]
>>> _                     [Wait for the other two processes to complete]
>>> d compile format      [Delete the other two layers]
>>> quit                  [End the session with the shell layer manager]
$ _                       [The ordinary shell prompt returns]
```

This sample session, somewhat oversimplified, illustrates some of the things you can do with the **shl** command: create new shell layers, give them names, and let them run simultaneously, display information about the active layers, block output display, switch layers, delete layers, and end the session. Here are a few additional notes:

1. You can abbreviate the commands, usually to one letter.
2. You can type more than one name after several commands (**block**, **unblock**, **layers**, and **delete**).
3. You can type (CTRL-Z) to return to the shell layer manager.
4. If you don't give names to your layers, the shell layer manager will call them `(1)`, `(2)`, `(3)`, and so on; you can then refer to them simply as `1`, `2`, `3`, and so on.
5. For additional information about layers, type ***-l*** after the **layers** command, like this: ***layers -l*** (or ***l -l***).

## 22.3 Connecting processes

One of the advantages of using UNIX is that you have so many different ways of connecting processes. In this section, you will learn three of these methods.

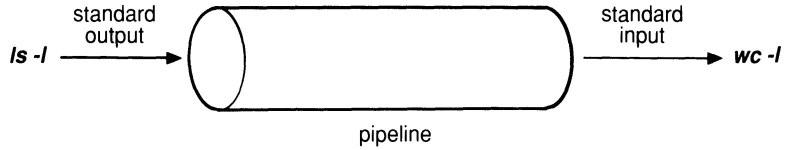
### Pipelines

A *pipeline* is a device that allows you to connect two processes by using the output of one as the input of another. The symbol that represents a pipeline on a command line is the vertical bar ( | ). For example, to count the number of files in the current directory, use this:

```
$ ls -l | wc -l
255
$ _
```

The **ls -l** command lists (in long format) the names of all the files in the current directory, each on a separate line. **wc -l** will count the number of lines in the output of **ls -l**, with the result displayed (255). The shell will invoke the **pipe** function to connect the two commands as pictured in Figure 22.1

FIGURE 22.1. A pipeline.



The output of the **ls -l** command becomes the input of the **wc** command, which is the same as if you typed the following string of commands:

```
$ ls -l > tempfile; wc -l < tempfile; rm tempfile
255
$ _
```

The advantage of using a pipe is that the temporary file **tempfile** becomes unnecessary. You can also connect more than two commands with pipes, as shown here:

```
$ ls -l | grep intro | lp
$ _
```

This sequence of piped commands sends the filenames that match the pattern *intro* to the printer.

A pipeline can connect only the standard output of the command on the left to the standard input of the command on the right. Just as it would make no sense to try to connect the closed ends of two lead pipes, it would make no sense to try to connect a process with no standard output to a process with no standard input. Some commands, such as **mail** and **lp**, do not use standard input or output the way other commands do. If you tried to use **lp** on the left-hand side of a pipeline, there would be no output to pipe.

## FILTER PROGRAMS

Most UNIX commands read from the standard input, and then process data before sending it out to the standard output. These commands, such as **grep**, **sort**, **spell**, and **diff,** are called *filter programs* (or just *filters*). As an example of a filter, the **grep** command searches an input file for lines matching a pattern, as illustrated in this example:

```
$ ls -l | grep intro | lp
$ _
```

Here **grep** takes the output of **ls -l** and selects only strings containing `intro` to be printed on the lineprinter. In this sense **grep** filters out strings of text, allowing some through and inhibiting others, like a mechanical or chemical filter. In UNIX, a command is a *filter* if it is free to accept its input via a pipe from another command and send its output via a pipe to another command. A command like **mail** can't be thought of as a filter because it cannot send its text anywhere except directly to the standard output. Two other commands that cannot be thought of as filters are **ls** and **kill**.

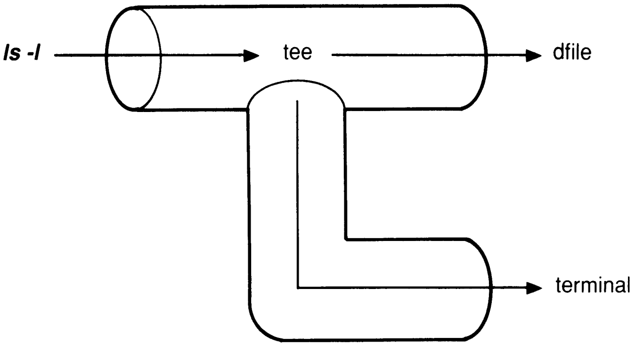
## Tees

The **tee** command allows you to view data as it is being written to a file. If you type,

```
$ ls -l | tee dfile
$ _
```

all the filenames in the current directory will be written to **dfile** and also displayed on screen, as illustrated in Figure 22.2.

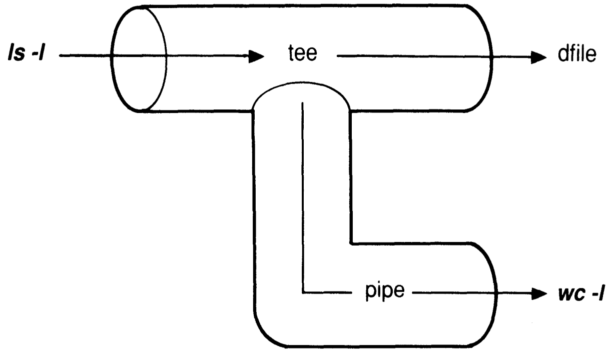
FIGURE 22.2. A tee.



Since the data is also sent to the terminal (standard output), you can use a pipeline to process this output, as shown here (see Figure 22.3).

```
$ ls -l | tee dfile | wc -l
$ _
```

FIGURE 22.3. A tee with a pipe.



## 22.4 Giving directives to the shell

### Command grouping

Grouping a list of commands within parentheses causes the shell to create a subshell to read and execute the enclosed commands. This subshell has the ability to run the enclosed command without affecting the values of variables in the current shell. This feature allows the user, for instance, to slip into another directory, run a few commands, and then be automatically returned to the current working directory. Assuming that your current directory is your home directory, you can execute this command line:

```
$ (cd /usr/james/man.spec; nohup nroff doc1 | lp &)
$ _
```

which is equivalent to

```
$ cd /usr/james/man.spec; nohup nroff doc1 | lp &
$ cd $HOME
$ _
```

Parentheses can appear anywhere on the command line. If the shell expects a right parenthesis but doesn't find it at the end of a command line, the shell will prompt with its secondary prompt (**PS2**) to let you know that it expects further input from you.

Grouping a list of commands within braces causes the shell to read and execute the enclosed commands, which produces the combined output. This method allows you to use the output of several commands as the input to another command, as in this example:

```
$ { nroff -cm text1; nroff -cm text2; } | col | lp
$ _
```

By using braces to enclose the commands in this way, you can let the shell queue the printing jobs for you, instead of having to wait for the first one to complete, then starting the second manually.

Unlike parentheses, braces must be separated from the commands they enclose. The exit status of a set of commands grouped by either parentheses or braces is the exit status of the last command executed from the set.

## CONDITIONAL EXECUTION

The shell provides a mechanism that allows you to invoke a command list in order with conditional execution. The command list is a sequence of one or more commands separated by semicolons (;), as in this example:

```
$ ls -l; who
-rw-r--r-- 1 john    27832 Jan 12 16:09 file1
-rwxr-xr-x 1 john     5358 Jan 15 14:16 file2
-rwxr-xr-x 1 john    13993 Feb 16 14:04 file3
john     tty20   Feb 16 11:34
$ _
```

The shell first invokes the **ls -l** command; then, when that has been completed, it invokes the **who** command.

You can have the shell perform additional execution by connecting two commands by either the AND-IF (**&&**) or the OR-IF (||) symbol. The format of an OR-IF statement is as follows:

```
$ command1 || command2
$ _
```

The shell invokes *command1* and examines its exit status. If *command1* fails, its exit status will be non-zero and the shell will invoke *command2*. Here is an example:

```
$ mkdir sample                    [Create directory sample]
$ rmdir sample > /dev/null || echo Failed
                                  [Remove directory sample]
$ rmdir sample > /dev/null || echo Failed
                                  [Remove directory sample]
Failed
$ _
```

The second command line yields no message because the command to remove the directory (**rmdir sample**) is successful. However, the third command line does produce a message. Since the directory has already been removed as a result of the second command line, the shell executes **echo failure** when **rmdir sample** fails (cannot be executed).

The format of an AND-IF statement is

```
$ command1 && command2
$ _
```

This will cause *command2* to be invoked only if the execution of *command1* succeeds. Here is an example:

```
$ test -f .profile && cp .profile backup
$ _
```

or

```
$ [ -f .profile ] && cp .profile backup
$ _
```

The command **test -f .profile** (or **[ -f .profile ]**) checks to determine whether file .profile exists in the current directory. (The **test** command is described in detail in Chapter 24, "Bourne Shell Program Control.") If you execute the above command line from your home directory where .profile is located, the shell will create a copy of it called backup. If you invoke the command line from one of your subdirectories (which doesn't contain .profile), nothing will happen.

## Pattern matching

The shell provides a means for generating its own list of filenames, based on a pattern-matching expression given as an argument. Such *regular expressions* were introduced in Chapter 13, "Searching and Sorting." For example,

```
$ ls -l *.c
```

will cause the shell to generate a list of filenames that end in .c to be used as arguments in this command (that is, to be listed). Pattern-matching characters like * are sometimes called *metacharacters*, which are listed here:

| | |
|---|---|
| ***** | Match any string of any length |
| **?** | Match any single character |
| **[ ]** | Match any of the characters enclosed |

Since the following metacharacters and other characters have special meanings to the shell, they should never be used in a name for any file or directory:

```
<    >    >>    *    ?    &    |
```

If you do attempt to use such characters, you may mislead the shell to interpret them rather than read them as arguments. For example, suppose you have a file named **files1**. If you try to change its name to **file?1**, this is what will happen:

```
$ mv files1 file?1
mv: files1 and files1 are identical
$ _
```

Why? Since **?** is a metacharacter, it will match any single character. Now try to change the above file to a different new name:

```
$ mv files1 file&1
148
1: Command not found.
$ _
```

As you can see, the shell interprets the ampersand (&) in its special meaning to invoke a background process to change the name **files1** to **file**, then reports that it can't find the **1** (one) command. (Having interpreted **&** as a request for a background process, the shell now thinks that the **1** that follows is another command.)

## 22.5 Summary

In this chapter you learned about Bourne shell processes, with discussions of shell functions; background commands; connecting processes with pipelines, filters, and tees; command grouping; conditional execution; and pattern matching.

### Shell functions

You can execute a sequence of shell commands more quickly from a shell function than from a shell script. You have the option of either typing the function and its invocation directly on the command line or entering the function into your login file (**.profile**) and making it available at all times.

### Background commands

The shell makes it possible for you to run more than one process at your terminal. If you add an ampersand (&) at the end of a command line, the shell will execute the command line as a *background process*, thereby allowing you to proceed immediately to another command without having to wait for the first to be completed. The shell will display the *process identification* number, which you can use to monitor the process. To make the process immune from hangup and quit signals, begin a command line with the **nohup** (no hangup) command.

Use the **ps** (process status) command to monitor background processes. If you execute **ps** without any arguments, the display will give four columns of information with these headings:

| | |
|---|---|
| PID | Process identifier |
| TTY | Terminal number |
| TIME | Cumulative execution time |
| COMMAND | Name of command |

When you add the **-l** argument to **ps**, you obtain ten additional columns, some of which are beyond the scope of this chapter. One of the columns displayed by **ps** is called `NI` (`NICE`), which indicates the priority level of each process. You can decrease the priority of a process by beginning the command line with the **nice** command. To terminate a background process, use the **kill** command followed by the process identifier.

To be able to run up to seven background processes with some control over each, use the **shl** command to begin a session with the shell layer manager. This allows you to create new shells (layers), monitor them, switch layers, and delete layers.

## Connecting processes

Here are three ways to connect processes:

1. A *pipeline* allows you to connect the standard output of one process to the standard input of another. The symbol for a pipeline is the vertical bar ( | ).

2. Some UNIX commands read from the standard input, and then select some of it to be sent to the standard output. These commands (like **grep**, **sort**, **spell**, and **diff**) are called *filter programs* (or *filters*). They often receive their input through a pipeline.

3. The **tee** command, which allows you to view data as it is being written to a file, is used in conjunction with a pipeline.

## Giving directives to the shell

Grouping a list of commands within *parentheses* will cause the shell to create a subshell to read and execute the enclosed commands. With this scheme, you can move to another directory, run some commands, and then have the shell return you to your current directory. Grouping a list of commands within *braces* will simply cause the shell to read and execute the enclosed commands. With this scheme, you can use the output of several commands as the input to another command.

A command list is a sequence of commands separated by semicolons (;). You can invoke a command list with conditional execution. An OR-IF construction (||) tells the shell to invoke the second command only if the first fails. An AND-IF construction (&&) tells the shell to invoke the second command only if the first succeeds.

The shell can use one of the following *metacharacters* to match patterns when selecting a filename:

* Match any string of any length
? Match any single character
[] Match any of the characters enclosed

# 23

# Bourne Shell Variables

In the first two chapters of Part V, you reviewed how the Bourne shell interprets commands entered from your terminal. We begin this chapter with a discussion of shell procedures and then go on to command substitution.

## 23.1 Shell procedures

A *shell file* (also known as a *shell procedure* or a *shell script*) is a file that contains one or more shell commands to be executed in sequence whenever the file is named as a command. This allows you to create your own commands. For example, here is a shell script that counts the number of characters in a file in the current directory. Here is how you use **ed** to create a file. (You could also use **vi** here instead of **ed**.)

```
$ ed counter
?counter
a
ls | wc -c          [List filenames and pipe them to the wc -c command]
.
w
11
q
$ _
```

Now you have a simple command file called counter that can be invoked with the **sh** (shell) command, which reads the contents of its argument file counter, and then executes these commands:

```
$ sh counter
20                  [Character count of counter]
$ _
```

At this point you have a file that contains a single command line, but you still can't execute it directly. The next step is to change the execution mode of the file with the **chmod** (change mode) command (see Chapter 3, "The UNIX File System"):

```
$ chmod u+x counter          [Make file counter executable]
$ _
```

Now, to invoke counter as a command in its own right, you no longer need to use the **sh** command. Just type ***counter***, the way you would type the name of any standard UNIX command.

The last step is to move counter into your bin directory, where most of your private commands should reside and add $HOME/bin to the value of your variable **PATH**. If you skip this step, you will be able to execute **counter** only when you are in the same directory with it. Otherwise, you must give the full pathname: /usr/john/newcomd/counter or $HOME/bin/counter.

## Preliminary commands

Before going on to the section on shell variables, let's discuss two commands that you'll be using as examples in that section and the sections that follow: **echo** and **read**.

## Displaying text on the screen **echo**

The **echo** command takes its argument string, which is terminated by a newline (\n), and writes it to the standard output (your terminal). The **echo** command is used mostly by the shell script to display messages on the screen, as in this example:

```
$ echo "This is a message \n"
This is a message
$ _
```

To suppress the newline, use \c instead of \n. This allows you to append one string to another without a line break between them, as in this example:

```
$ echo "This is a message \c"
This is a message$ _
```

Note the position of the prompt ($) at the end of the message instead of on the next line. The \c character is used often in shell programs to prompt for input from the user. For example, assume that there is a shell file called getkey that contains the following two lines:

```
$ cat getkey
echo "Press any key, then RETURN to continue: \c"
read inkey
$ _
```

To create such a file, follow the steps given in the preceding section, "Shell Procedures," page 000. Once you have created the file, you can invoke it as a UNIX command:

```
$ getkey
Press any key, then RETURN to continue: _
```

The program is now waiting for your input at the keyboard. Any keystrokes you enter will be displayed at the cursor location. As soon as you press (RETURN), the program will exit and the prompt will reappear on the next line—the **read** command expects the (RETURN) key as end of the input string.

### Receiving input from the user **read**

The **read** command takes one line from the standard input and assigns the first word to the first name, the second word to the second name, and so on, with any leftover words assigned to the last name. If there is only one name, as in this simple example, **read** assigns the entire input to this name.

In the file **getkey**, the **read** command assigns your keyboard input as the value of the variable **inkey**. If you want to see this input displayed after you've typed it, add another **echo** command to the end of the **getkey** file, like this:

```
$ cat getkey
echo "Press any key, then RETURN to continue: \c"
read inkey
echo $inkey        [Now add this line]
$ _
```

Now when you invoke the **getkey** program (assuming that you enter ***KEYBOARD*** and then press (RETURN) after the prompt), you will see this:

```
$ getkey
Press any key, then RETURN to continue:
                                   [You type KEYBOARD here]
KEYBOARD                           [This time getkey echoes it here]
$ _
```

You will see the **echo** and **read** commands used more in shell files in the next chapter. For now, you are ready to learn about shell variables.

## 23.2 Shell variables

A shell variable, which can be assigned either by the shell itself or by a user, provides the shell with basic information. You learned about several shell variables in Chapter 21, Introduction to the Bourne Shell." The shell usually assigns such variables default settings, which a user can then reassign for different applications. The name of the variable can be any sequence of letters, digits, and underscores, but must always begin with either a letter or an underscore. Use an equal sign (=) with no spaces on either side to assign a value to a shell variable. Additional rules and restrictions will be discussed later in this chapter.

## Assigning variables

The following example shows the value `STRING` being assigned to the variable called **VARIABLE**:

```
$ VARIABLE=STRING
```

To retrieve the value of the variable, precede the variable name with a dollar sign (**$**). The **echo** command provides a simple way to examine the value of variable:

```
$ echo $VARIABLE \n
STRING
$ _
```

You can assign more than one variable in an assignment statement, as shown in the example below. Note the spaces between assignments:

```
$ a=T b=H c=I d=S  [Assign more than one variable in a statement]
$ echo $a$b$c$d \n [Retrieve the values of the variables assigned]
THIS
$ _
```

In the following example, variable **b** is assigned the value of variable **a**, so that variables **a** and **b** have the same value:

```
$ a=123           [Assign 123 to variable a]
$ b=$a            [Assign variable b the value of variable a]
$ echo $a $b \n   [Echo both values, separated by a space]
123 123           [The values of a and b are displayed]
$ _
```

You can combine the two statements above into a single, more complicated statement to obtain the same result:

```
$ z=$y  y=123     [Set z equal to the value of variable y (123)]
$ echo $z $y \n   [Echo both values, with a space between them]
123 123           [Both values are displayed]
$ _
```

Since the shell assigns variables from right to left (not from left to right), you have to be careful in constructing multiple assignment statements, as in the example above. In the example that follows, the shell assigns **z** the previous value of **y** (123), not the new value abc, as the user expects:

```
$ y=abc z=$y          [The shell assigns 123, not abc, to z]
$ echo $z \n
123                   [Now the shell echoes 123 as the value of z]
$ echo $y \n
abc                   [The shell echoes 123 as the value of y, though]
$
```

Variable assignments are used often in shell procedures. However, you can still benefit from these assignments during ordinary interactive use of the shell. For example, you can use variables in place of lengthy command names, arguments, and filenames, thereby avoiding a lot of unnecessary typing. Here, the shell assigns a long pathname to **HERE**:

```
$ HERE=/usr/james/selection/car/camaro
```

Now you can type

```
$ cd $HERE
```

and obtain the same result as if you had typed

```
$ cd /usr/james/selection/car/camaro
```

In the following example we'll use a variable to hold a long command line with many options: **nroff -h -rT450 -mm** *file*. Let's assign a variable **form** to this command string (single quotation marks are required here because of the spaces between the individual arguments):

```
$ form='nroff -h -rTl -T450 -mm'
```

So now, instead of typing the entire command every time you want to use it, you only have to type something like this:

```
$ $form file_15
```

which is equivalent to

```
$ nroff -h -rTl -T450 -mm file_15
```

## Using quotation marks

So far we have discussed simple variable assignments, without spaces, tabs, semicolons, and newlines. Any time one of these appears in the assigned value string, you have to enclose this assigned value (right-hand side) of the statement in single quotes (′) as shown in the following example:

```
$ string='This has both; it has a semicolon and a space'
$ echo $string \n
This has both; it has a semicolon and a space
$ _
```

If you do not enclose the assigned value above in single quotes, an error message will appear on the screen.

However, if you want to include a variable subtitution within an assigned value string, you must use double quotation marks (" "). This will force the shell to scan and substitute the values of the enclosed variables:

```
$ S=space M=semicolon
$ string=" This includes both; it has a $S and a $M "
$ echo $string \n
This includes both; it has a space and a semicolon
$ _
```

Besides, the three special characters ($, `, ") will retain their special meaning when enclosed within double quotes, while the shell will take them literally when they are enclosed within single quotes, like this:

```
$ string=' This string has three: $S, ", ` '
$ echo $string \n
This string has three: $S, "[No`substitutions are made ]
$ _
```

## Evaluating a command string **eval**

You can force the shell to rescan the string using the **eval** command. The eval command rescans the command arguments either to perform commands or to substitute variables as specified. Here is an example:

```
$ a=1234             [Assign a the value 1234]
$ b='$a'             [Assign b the value of variable a]
$ eval echo $a $b    [Force the shell to rescan and substitute $b]
1234 1234
$ echo $a $b \n      [Echo without evaluation]
1234 $a
$ _
```

The advantage of having the shell rescan your command line is that you can have all substitutions made before execution, as shown in here:

```
$ speed='stty 9600'    [Assign variable speed the value stty 9600]
$ port=' > /dev/tty01' [Assign variable port the value in quotes]
$ eval $speed $port    [Evaluate both variables]
stty 9600 > /dev/tty01
$ _
```

The **eval** command makes the above statement appear to the shell as if you had typed this:

```
$ stty 9600 > /dev/tty01
```

Of course, you could also set the whole command string in one variable, as explained earlier, without the **eval** command, but **eval** gives you a little more flexibility when you need it.

## Distinguishing a variable's name

As you have already seen, the value that is assigned to a variable can be the value of another variable. You can also use a variable to help define itself, as shown here:

```
$ word=character [Variable word is assigned the value character]
$ word=meta$word [Variable word is used in its own definition]
$ echo $word \n
metacharacter
$ _
```

If you try to append extra characters to the end of a variable's name, as shown here:

```
$ far=tele
$ tv=$farvision
$ echo $tv \n
$ _
```

the shell will assign a null string to **tv** since it can't find a variable called **farvision**. The solution to this problem is to enclose the variable's name within braces, as shown here:

```
$ tv={$far}vision
$ echo $tv \n
television
$ _
```

For those variables that you may want to be automatically initialized every time you log in, you can have them set in your **.profile** file.

## 23.3 Command substitution

The shell allows you to assign the results of an invoked command string as the value of a variable, by enclosing this command string in grave accents (‘command string‘), as illustrated in this example:

```
$ fnames=`ls`      [Assign fnames the results of the ls command]
$ _
```

This will force the shell to invoke the **ls** command, assigning the result as the value of the variable **fnames.** Assume that there are four files in the current directory called **file_1**, **file_2**, **file_3**, **file_4**. Now, using the **echo** command to display the value of **fnames**, you will see the result as follows:

```
$ echo $fnames \n
file_1 file_2 file_3 file_4
$ _
```

You can use command substitution only with commands that have standard output. Since the **lp** command gives no standard output, avoid doing this:

```
$ VAL=`lp file_1`             [Not recommended]
$ _
```

The output of **lp file_1** is sent to the printer, and the shell assigns a null value to the variable **VAL**.

Command substitutions are often useful in shell files, as in the following example:

```
$ CURRENT=`pwd`               [Get the current directory]
$ cd /usr/dennis/plan/goal    [Move to a different directory]
$ ...                         [Do something in that directory]
$ cd $CURRENT                 [Return to the previous directory]
```

The variable **CURRENT** is used to hold the pathname of the current directory while the program goes to a different directory to perform some task. The last statement allows you to return to the previous directory without having to remember its name.

## Evaluating arithmetic expressions **expr**

The **expr** command takes its arguments as an expression (logical or arithmetic). The result of this expression is sent to the standard output. These arithmetic operators can be used with the **expr** command:

| | |
|---|---|
| + | addition |
| - | subtraction |
| | multiplication |
| / | division |
| % | remainder |

Because the **expr** command treats each operator or value as a separate argument, you have to separate operators and values with spaces. To prevent metacharacters like the following from being interpreted by the shell itself

```
( * & ? ^ [ ]
```

you have to *escape* them (release them from their special meanings) either by by enclosing them within single quotes ('*') or double quotes ("*") or by preceding them with backslashes (\). In the fourth example below, the asterisk (*), which could be interpreted by the shell as a wild card character, is escaped with single quotes.

```
$ a=2                          [Initialize a with value 2]
$ a=`expr $a + 7`              [Add 7 to the value of a and assign to a]
$ b=`expr $a / 3`              [Divide a by 3 (9 / 3 = 3) and assign to
                                b                                       ]
$ c=`expr $a - 1 '*' $b`       [Multiply a-1 by b (8 x 3 = 24) and assign
                                to c                                      ]
$ d=`expr $c % 5`              [Divide c by 5 (24 % 5) and assign the
                                remainder to d                        ]
$ e=`expr $d - $b`             [Subtract b from d (4 - 3) and assign to
                                e                                       ]
```

The result is assigned to variable **e**. Let's check all the values:

```
$ echo $a $b $c $d $e \n
9 3 24 4 1
$ _
```

## Comparing strings **expr**

The **expr** command also allows you to compare strings with its matching operator (:). The output is the number of characters in the two strings that match. An output of zero (0) indicates failure of the comparison. Here is an example:

```
$ R=`expr 'abcdefg' : 'abcd' `   [Match the two strings and as-
                                  sign to R]
$ echo $R \n                     [Recall the matching number R]
4
$ _
```

The string comparator of the **expr** command uses the same scheme used by **ed**. That is, the second string is compared against the first string, starting with the first character. Because of this, even one non-matching character in the second string will cause a failure, resulting in an output of zero (0), as illustrated in the three examples below:

```
$ R=`expr 'abcdefg' : 'abce' `   [Missing d in the second string]
$ T=`expr 'abcdefg' : 'bcde' `   [Missing a in the second string]
$ V=`expr 'abcd' : 'abcdefg' `   [Second string is longer than the
$ echo $R $T $V \n                first]
0 0 0                            [All three comparisons fail]
$ _
```

You could perform the same string comparison using variables, as in this example:

```
$ C=`expr $M : $N`
```

By taking advantage of the string-comparison property of the **expr** command and the character matching that the shell provides, we can also use a string comparison to determine the length of a string. Let's assign to variable **A** a character string, then count the number of characters in this string:

```
$ A='This is an experiment'  [Assign a string to variable A]
```

Since variable **A** is a character string with spaces in it, A must be enclosed in *double quotes* in the following example; otherwise, the statement will produce a syntax error. The second argument is enclosed in *single quotes* to preserve the special meaning of the asterisk (*) to the shell:

```
$ count=`expr "$A" : '.*' `  [Note the asterisk (*) on the right]
$ echo $count \n             [This retrieves the value of count]
21                           [String length ]
```

Another feature of the **expr** command is the ability to extract only a portion of a character string, as in

```
$ B=`expr "$A" : '...\(.*\)' `
```

The five dots in front of the backslash (\) represent the five characters to be skipped (including the space), starting at the beginning of the string. Now, echoing variable **B**, we have

```
$ echo $B \n
is an experiment      [The first five characters have been omitted]
$ _
```

The **expr** command checks statements for errors before processing. Here are some possible errors:

1. A syntax error will occur any time an operator or operand is missing or illegal.
2. A "non-numeric argument" message will be displayed any time you try to use an *arithmetic* operation on a character or string.

## 23.4 Conditional substitution of variables

You can test a variable to determine whether or not it has been assigned a value. Then, if it has, you can either use this value in an expression or replace the value with another. This is known as *conditional substitution.* The shell provides four variations of conditional substitution, using four symbols to indicate comparison (**-**, **=**, **+**, **?**):

### Use value or switch **${*var-other*}**

If *var* has been assigned a value, use this value; otherwise, use string *other* instead, leaving *var* unassigned. In the example that follows, we assume that variable **VAL** is *not* set, with the result shown:

```
$ echo ${VAL-123} \n          [Test whether VAL has been set]
123                           [Since it has not, string 123 is used]
```

Here is a similar example with a variable used instead of an actual string on the right as the replacement:

```
$ SUB=456 \n         [Assign 456 to SUB]
$ echo ${VAL-$SUB}   [Again we assume that VAL is unset]
456                  [So SUB's value is used instead]
```

### Use value or assign other **${*var=other*}**

If *var* has been assigned a value, use this value; otherwise, assign the value *other* to *var* before using *var*, as shown in the following examples:

```
$ echo ${VAL=xyz} \n          [VAL has no assigned value]
xyz                           [So xyz becomes the value of VAL]
$ echo $VAL \n                [Instead of remaining unassigned, VAL
xyz                            now has the value xyz]
```

### Switch or use null ${*var*+*other*}

If *var* has been assigned a value, use the value of *other.* If *var* is not set, use a null string as its value, leaving *var* unassigned.

```
$ THIS=145                          [Assign 145 to THIS]
$ THAT=abc                          [Assign abc to THAT]
$ echo ${THIS+$THAT} \n             [Perform the test]
abc                                 [The value of THAT appears]
$ echo $THIS
123                                 [THIS retains its value]
$ _
```

### Use value or abort ${*var*?*empty*}

If *var* has an assigned value, use this value; otherwise, display the message *var: empty* and abort the procedure. Once again, assume that **VAR** is unassigned:

```
$ error='Not set--abort'
$ echo ${VAR?$error} \n
VAR: Not set--abort
$ _
```

The default for a missing message on the right side of the question mark is a standard message:

```
$ echo ${VAR?} \n
VAR: parameter not set
$ _
```

## 23.5 Positional parameters

The shell itself sets positional parameters to identify the positions of items (or *arguments*) on the command line. Arguments must be separated by spaces so that the shell can distinguish them from each other. The shell identifies items on a command line with numbers beginning with zero. The first item (the name of a shell procedure) is always denoted by **$0**; then the first argument is denoted by **$1**, the second by **$2**, and so on up to **$9**, as illustrated here:

```
   $0  $1   $2     $9                     [positional parameter]
 $ diff file1 file2 ... file9             [command line]
```

## Assigning positional parameters **set**

Before going further into the positional parameter, let's consider the **set** command as it relates to positional parameters. The **set** command is often used to assign a positional parameter to an argument string as in

```
$ set arg1 arg2 arg3 arg4 arg5 arg6 arg7 arg8 arg9
$ _
```

This **set** command forces the shell to assign the first positional parameter **$1** to **arg1**, the second **$2** to **arg2**, the third to **arg3**, and so on. You can verify these assigments by echoing them back:

```
$ echo $1 $3 $4 $2 $4 $5 $6 $7 $8 $9 \n
arg1 arg2 arg3 arg4 arg5 arg6 arg7 arg8 arg9
$ _
```

## Accessing specific arguments

When the **set** command is used with command substitution, each element of the output of the command substitution is assigned a positional parameter. In the following example, the **date** command is enclosed within grave accent marks (‘ ‘) to form a command substitution:

```
$ set 'date'
$ _
```

The output of the **date** command becomes a sequence of arguments for the **set** command, and no output will show on your screen. However, if you invoke the **date** command by itself, you should see this:

```
$ date
Wed Feb 15 21:49:20 1984
$ _
```

After you invoke **set ‘date‘**, each item in this string is assigned a positional parameter $1, $2, $3, $4, $5, as explained in the next section. If you are interested in the time only, select it this way:

```
$ echo $4 \n
21:49:20
$ _
```

You can also rearrange these items in a different format:

```
$ echo 'DATE: $1 $2 $3 $5  TIME: $4 \n'
DATE: Wed Feb 15 1984  TIME: 21:49:20
$ _
```

## Accessing more positional parameters **shift**

To process command-line options, use the **shift** command, which shifts each argument one position to the left. After the **shift** command, the argument at positional parameter $4 is moved to position $3, $3 is moved to position $2, $2 is moved to position $1, and $1 is discarded (the command name in position $0 remains unchanged, however). Here is an example:

```
$ echo $1 $2 $3 $4 $5 $6 $7 $8 $9 \n
arg1 arg2 arg3 arg4 arg5 arg6 arg7 arg8 arg9
$ shift              [Shift arguments to the left]

$ echo $1 $2 $3 $4 $5 $6 $7 $8 $9 \n
arg2 arg3 arg4 arg5 arg6 arg7 arg8 arg9 arg10
$ _
```

After this shift, **arg2** is in the first position and **arg10** is at the end.

## Matching arguments with a wildcard

The shell also provides the asterisk (*) as a wild card to match any positional parameter. You can access all positional parameters by typing

```
$ echo $* \n
arg2 arg3 arg4 arg5 arg6 arg7 arg8 arg9 arg10 arg11
$ _
```

This **echo** command line and the one used in the example above are not exactly the same. The previous **echo** command displayed nine positions; this one displayed up to the last current positional parameter.

## Passing arguments to a command

The shell always assigns arguments to positional parameters before passing them to an invoked command. For example, suppose you have an executable command file named odd_even, which contains **echo $1 $3 $5 $2 $4 $6**. If you invoke **odd_even** with six arguments, this is what happens:

```
$ odd_even one two three four five six
one three five two four six
$ _
```

As you can see, the echoed line reflects the order set by the **odd_even** file.

## 23.6 Reserved variables

The special variables listed here are set by the shell itself, rather than by the user:

**#** *Number of Arguments*—The number of positional parameters (not counting $0) that have been used to record arguments. For example, let's see how many arguments were saved by the shell as the result of the **set** command in this example:

```
$ set 'date'
$ echo $# \n
5                [Five positional parameters have been set]
$ _
```

**?** *Return Code*—The exit status code returned to the shell from the last executed command. In general, successful execution of the command returns a zero; any other execution returns a non-zero value.

```
$ echo $? \n
0                [Exit status of the echo command: successful]
$ _
```

**$** *Process Identification Number of the Current Process*—Since this number is unique, it is sometimes used as a temporary filename, as in this example:

```
$ tempfile=file$$              [Set tempfile variable ]
$ sort file1 -o $tempfile      [Output to variable tempfile]
$ mv $tempfile file1
```

**!** *Process Identification Number of the Background Process Most Recently Invoked*—This number remains in effect even after the process has been completed.

```
$ echo $! \n
243              [The most recently invoked background process]
```

**-** *Status of Shell Flags*—This variable identifies all current shell flags that can be turned on or off by the **set** command. To examine the current settings of shell flags, type

```
$ echo $- \n
s                [The s flag is on; all others are off]
$ _
```

These variables can be set only by the shell. If you attempt to assign them any value, an error message will be displayed, as shown here:

```
$ #=256                        [You try to set variable # manually]
#=256: Command not found.      [This is what happens]
$ _
```

## 23.7 Summary

In this chapter, you learned about shell procedures and shell variables. You learned about assigning shell variables, using quotation marks, naming variables, command substitution, conditional substitution, positional parameters, and pre-defined variables.

To create a shell procedure, enter the desired shell commands in a file and make the file executable with the **chmod** command. The **echo** command takes its argument string and writes it to the standard output (displays it on your screen). The **read** command takes standard input and assigns it to a variable (or to several variables).

### Shell variables

Shell variables, like other variables used in programming, can be assigned values. To retrieve this value, use the **echo** command, preceding the variable's name with a dollar sign ($). You can use shell variables to spare yourself extra typing.

To allow a string to contain spaces, tabs, semicolons, and newlines (but not to allow substitution of variables), use single quotation marks (' '). To allow a string to contain spaces, tabs, semicolons, and newlines, along with substitution of variables, use double quotation marks (" ").

The **eval** command works like the **echo** command, except that **eval** can take an entire command line, evaluate variables, and substitute their assigned values. To distinguish a variable's name from other characters, enclose the name in braces.

### Command substitution

To assign to a variable the standard output of an invoked command string, enclose the command string in grave accents.

To evaluate arithmetic expressions, use the **expr** command, along with the symbols for addition (+), subtraction (-), multiplication (*), division(/), and remainder (%). To compare two strings, use the **expr** command with a comparator (:). The result will be the number of matching characters in the two strings. You can use string comparison to determine the length of a string (compare the string to .*) or to extract a portion of the string (use one dot for each character to be deleted, then enclose .* in substring brackets \\( and \\)).

## Conditional substitution of variables

To test a variable to determine whether or not it has been assigned a value and then base your next action on the result of the text, use one of the following four types of expressions:

**$**{*var*-*other*} If *var* has a value, use it; if not, use *other* and leave *var* unassigned.

**$**{*var*=*other*} If *var* has a value, use it; if not, assign the value of *other* to *var*.

**$**{*var*+*other*} If *var* has a value, use the value of *other* instead; if not, use a null string and leave *var* unassigned.

**$**{*var*?*empty*} If *var* has a value, use it; if not, display the message *var: empty* and abort the process.

## Positional parameters

The shell sets positional parameters to keep track of a command and its arguments on a command line, using **$0** for the command, **$1** for the first argument, **$2** for the second, and so on. To assign values to positional parameters, use the **set** command.

To select a specific argument, use the **set** command with a positional parameter. To access arguments numbered higher than 9, use the **shift** command. To match any positional parameter, use the shell wild card symbol (*) after the dollar sign. You can rearrange the order of the arguments by placing the positional parameters in the desired order, then executing the command.

## Reserved shell variables

The following special variables can be set only by the shell, but can be accessed by the user:

| | |
|---|---|
| **#** | Number of arguments in a command line |
| **?** | Return code for the command |
| **$** | PID of the current process |
| **!** | PID of the most recent background process |
| **-** | Status of shell flags |

# 24

# Bourne Shell Program Control

Within a *shell file* (also known as a *shell procedure* or *shell script*), the shell acts as a programming language. Like other languages, the shell provides features like these:

- Flow control commands (**if**...**then**...**else**, **while**, **for**, and so on)
- Interrupt handling (**trap**), variable setting, and parameter passing (refer to Chapter 23, "Bourne Shell Variables")
- Calls to other commands or shell procedures as subroutines

Because a shell procedure is written in a high-level language, it is easy to modify. Also, a shell procedure doesn't need to be compiled. Since there is no object file to recreate, it is easier to maintain your shell program.

The shell also provides devices for debugging a shell procedure. These allow you to watch each step of your program execute, with errors displayed whenever the shell finds illegal expressions.

In this chapter, you will learn about flow-control commands, interrupt handling, and many other features of shell programming. Early examples will be very simple, with more involved examples shown later.

## 24.1 Constructing loops

In any kind of programming, it is common to have a segment of a program executed repeatedly, usually with a few changes made just before each repeated execution. Such a segment is called a program *loop*. This section deals with the different ways you can construct a loop when programming the UNIX shell. Before going into looping itself, let's take a look at the **true** and **false** commands, which we'll use to simplify our discussions of the looping commands.

### THE **true** AND **false** COMMANDS

**true**
**false**

When any UNIX command completes execution, it passes a value to its parent process (the calling program) to inform the parent of the outcome.

This value, called the *exit status* of the command, is a small integer. A value of zero is agreed to mean *true* (the command ran successfully); a value other than zero, to mean *false* (the command did not run successfully). Furthermore, since there are any number of values that can mean *false*, the particular value often explains *why* the process failed.

To allow you to force one exit status or the other, the shell provides you with a pair of commands called **true** and **false**. The **true** command returns an exit status of *true*; the **false** command returns an exit status of *false.* In a moment we'll use these commands to show clearly how some of the commands used to form loops (**while** and **until**) work. For right now, let's look at the **true** and **false** commands themselves:

```
$ true;A=$?        [Obtain the exit status of the true command ]
$ false;B=$?       [Obtain the exit status of the false command ]
$ echo $A $B
0 1                [The status of each is displayed]
$ _
```

## Looping while true **while**

The **while** command sets up a loop that continues to be executed as long as the **while** command detects a *true* condition, as shown here:

```
while  command list
do  [execute commands
     between do and done]
done
```

The **while** command examines the exit value of the *command list*, which can be either a single command or a group of commands. Then, based on this return-value, takes appropriate action: if a zero is returned, all commands between **do** and **done** are executed. This process is repeated until the **while** command detects a non-zero return-status (or a **break** command is executed). In the example below, the **while** command nevers see a non-zero value. Since the output of the **true** command is always zero, this program will loop forever!

```
while true
do echo THIS IS AN ENDLESS LOOP
done
```

To stop this program, press the DEL key.

## Looping while false **until**

The format of the **until** command and its command execution processes are the same as for the **while** command, except that the looping condition is a non-zero-exit value (false). The following program gives the same result as the one that uses the **while** command in the section above:

```
until false
do echo THIS IS AN ENDLESS LOOP
done
```

The DEL key stops this program the same way it stops the **while** loop. The **done** statement closes the loop.

## NESTED LOOPS

It is possible to place one program loop inside another, producing a number of repetitions of the inner loop with each single repetition of the outer loop. This arrangement is called a *nested loop*, and is illustrated in this example:

```
$ cat loop
out=1
in=3
until out=`expr $out - 1`  -----------------+
do echo outer $out                          |
   while in=`expr $in - 1` ---+             |
   do echo inner $in          | Inside loop | Outside loop
   done                    ---+             |
done                       -----------------+
$ _
```

Since the variable **out** has been initialized to 1, the expression **$out - 1** decreases the value of **out** to zero (0), which gives the **until** statement a return value of *false*. This causes the **do** statement of the outside loop to be executed.

Since the variable **in** has been initialized to 3, the inside loop will be executed twice—until the expression **$in - 1** reduces the value of **in** to zero (0), which ends the inside loop, and return program control to the outside loop.

This time the expression **$out - 1** at the top of the outside loop reduces the value of **out** to **-1**, a non-zero value that ends the outside loop (and the program). If you execute **loop**, this is what you will see:

```
$ loop
outer loop 0
inner loop 2
inner loop 1
$ _
```

## Breaking out of a loop **break**

The **break** command allows you to exit from an enclosing loop involving one of the commands already discussed (**while** and **until**), as well as two commands to be discussed later in this chapter (**for** and **case**). To break from more than one loop level, you must specify the number of levels to break. If you execute the program on the left below, you will see the result displayed on the right. Without **break**, the innermost loop (loop 3) would keep repeating indefinitely. But **break** forces execution back out to the middle loop (loop 2), so that loops 2 and 3 alternate single turns at executing. (This example illustrates how the **echo** command comes in handy when you want to trace the execution of a program.)

```
$ cat multiloop                             $ multiloop
while true     --------------------+         LOOP 1
do echo LOOP 1                     |         LOOP 2
   until false    ----------+      |         LOOP 3
   do echo LOOP 2           |      |         LOOP 2
      while true   --+      |      | loop 1  LOOP 3
      do echo LOOP 3 |      | loop 2         LOOP 2
         break       | loop 3      |         LOOP 3
      done         --+      |      |         LOOP 2
   done           ----------+      |         LOOP 3
done           --------------------+         . . .
```

The shell procedure's syntax requires that for every **while**, **for**, or **until** loop command, there must be a matching **done** statement to terminate it. After the LOOP 3 message is displayed, the innermost loop is broken by the **break** command, and the **until** loop is again executed. This process will repeat endlessly, displaying LOOP 2 and LOOP 3 alternately.

Replacing **break** by **break 2** will force the program back to the outermost loop (loop 1), with LOOP 1, LOOP 2, and LOOP 3 messages displayed alternately:

```
$ cat multiloop                               $ multiloop
while true     ---------------------+         LOOP 1
do echo LOOP 1                      |         LOOP 2
   until false   ---------+         |         LOOP 3
   do echo LOOP 2         |         |         LOOP 1
      while true   ---+   |         | loop 1  LOOP 2
      do echo LOOP 3  |   | loop 2  |         LOOP 3
         break 2      | loop 3      |         LOOP 1
      done         ---+   |         |         LOOP 2
   done          ---------+         |         LOOP 3
done           ---------------------+         . . .
```

A **break 3** statement would cause the program to exit all three loops and the program to be terminated.

```
$ cat multiloop                          $ multiloop
while true ---------------------+        LOOP 1
do echo LOOP 1                  |        LOOP 2
   until false  -----------+    |        LOOP 3
   do echo LOOP 2          |    |        $ _
      while true   ---+    |    | loop 1
      do echo LOOP 3  |    | loop 2
         break 3      | loop 3  |
      done         ---+    |    |
   done         -----------+    |
done       ---------------------+
```

## Resuming execution of a loop **continue**

The **continue** command resumes program execution at the enclosing **for** or **while** loop, the closest enclosing loop being the default when the command is invoked alone. If the level number is explicitly given, the enclosing **for** or **while** loop at the level specified is where the program resumes execution, as illustrated here:

```
while true          ----------------------------+
do command list 3                               |
   until false         ----------------+        |
   do command list 2                   |        |
      while true         -----+        |        | Level 3
      do command list 1       |        | Level 2
         continue $level      | Level 1         |
      done               -----+        |        |
   done                ----------------+        |
done                ----------------------------+
```

If variable **level**'s value is 2, the **until** loop will be executed. If the value of **level** is 3 the outside **while** loop will be executed. If **level** is not set (or set to 1), the program will remain in the innermost **while** loop.

## Exiting a loop **exit**

The **break** and **continue** commands allow your program to leave its current loop. The **exit** command, however, allows your program to abort the entire procedure, not just the current loop. The **exit** command allows you to stop processing in the event of an error. When invoked by itself, **exit** returns an exit status of zero (true) to its parent program. In the following example, **main** is the parent program, which calls procedure **subpro**. By using the name of the procedure as a Boolean variable, this program actually relies on the exit status of the procedure to govern the conditional statement. Then, based on the exit status returned, **main** will display one of two messages.

```
$ cat main
```

```
if subpro
   then echo ' Exit status is zero '
   else echo ' Exit status is non-zero '
fi
$ cat subpro
HERE='pwd'
status=0
. . .
exit $status
$ _
```

In procedure **subpro**, variable **status** is initialized to a value of zero (0), which causes the message “Exit status is zero” to be displayed. If the value of **status** were non-zero, the message “Exit status is non-zero” would be displayed.

## LOOPING ON A VARIABLE **for**

The **for** command sets up a conditional loop using the following format:

```
for   variable in  list
do  [execute command(s)
    between do and done]
done
```

The **for** command assigns the next word in the list as a new value of *variable*, then examines this new value. If the variable has a valid value, commands between the **do** and **done** statements are executed; if the variable has a null value (indicating that there are no more words left in the list), the command following the **done** statement is executed. Here is an example:

```
$ cat example                                $ example
list='word1 word2 word3 word4'               word1
for VAL in $list                             word2
   do echo $VAL                              word3
   done                                      word4
echo 'END OF LIST'                           END OF LIST
$ _                                          $ _
```

When this shell program is invoked, word1 is assigned to variable **VAL** first. Then the **for** command examines the value of **VAL**. Since **VAL** has a valid value (word1), **echo $VAL** is executed. The program resumes execution at **for**, this time the next argument (word2) is assigned to **VAL**, and the process is repeated. The **for** loop continues through two more iterations (one for word3 and one for word4). Then, in the next iteration after the last argument (word4), **VAL** assumes a null value, indicating that the end of the list has been reached. At this point, the **for** loop is broken, and the program by-passes the **do** ... **done** list and executes **echo 'END OF LIST'**.

In the shell procedure that follows, variable **VAL** assumes the procedure's arguments, which are passed by the shell as its value:

```
$ cat rebounce                  $ rebounce one two three
for VAL                         one
do echo $VAL                    two
done                            three
$ _                             $ _
```

The **rebounce** program is invoked with three arguments, which are passed in the positional variables and assigned one by one as values of variable **VAL** each time the **for** command is executed. If the **rebounce** program is invoked with the asterisk (*) as its argument, the result is that the names of files in the current directory are passed in the positional variables to the **rebounce** program. The two shell procedures below produce the same results, as explained below:

```
$ cat rebounce.1                $ cat rebounce.2
set `ls`                        for VAL in *
for VAL                            do echo $VAL
   do echo $VAL                    done
   done                         $ _
$ _
```

In the example on the left (**rebounce.1**), the **set** command assigns positional parameters to the elements of the output of the **ls** command, which form the list of filenames and directory names in the current directory. In the example on the right (**rebounce.2**), the shell matches the names of files and directories of the current directory, then assigns them one by one to variable **VAL**.

## 24.2 The conditional statement

The general format of the **if** command in its simplest form is as follows:

```
if  expression
   then  command list
fi
```

The **if** command evaluates the return-value of the expression that follows and then takes action: if the return-value is zero (true), the commands between **then** and **fi** are executed; if the return value is non-zero (false), the command following **fi** is executed. Here is a simple example:

```
$ cat check                       $ check
if true                           TRUE VALUE
   then echo 'TRUE VALUE'         $ _
fi
$ _
```

The **true** command passes a zero value to **if**, which causes the **echo** command to be invoked. As an experiment, modify the program by replacing **true** with **false**, then invoke the program again.

```
$ cat check                       $ check
if false                          $ _
   then echo 'TRUE VALUE'
fi
$ _
```

No message is displayed this time. Because **false** exits with a non-zero value, the **echo** command is skipped.

## AN ALTERNATE PATH **else**

The **else** statement goes with the **if** command to provide an alternate execution path for the program when a condition doesn't satisfy the **if** command. The command format is as follows:

```
if  expression
   then  command-list 1
   else  command-list 2
fi
```

If the return value of *expression* is zero (true), the commands between **then** and **else** are executed. If the return value is non-zero (false), commands between the **else** and **fi** statements are invoked. Here is an example

```
$ cat match
if a=`expr "information" : "$1"`
   then echo " $a characters matched "
   else echo 'No character matched'
fi
$ _
```

The expression **‘expr ”information” : ”$1”‘** compares the argument string **$1** with the string `information`. If the expression's return status is zero (indicating that there are some matches between two strings), the statement between **then** and **else** is executed. Otherwise, the statement between **else** and **fi** is executed and then the program exits. Let's invoke program **match** with arguments and see the results:

```
$ match info
4 characters matched
$ match format
```

```
No character matched
$ _
```

## Combining **else** and **if**

**elif**

The **elif** statement is a combination of the **else** and **if** statements, allowing the shell program with nested **if** commands to be written in a shorter form. For example, here is a program **checkname** that attempts to match the argument being passed with strings `Mark`, `John`, and `Dennis`. Here is the program's contents:

```
$ cat checkname
if expr "$1" : "Mark" > /dev/null
then echo   Matching word is Mark
else if expr "$1" : "John" > /dev/null
     then echo   Matching word is John
     else if expr "$1" : "Dennis" > /dev/null
          then echo   Matching word is Dennis
          else echo * * * No match * * *
          fi
     fi
fi
$ _
```

This program can be rewritten with the **elif** statement as follows:

```
$ cat new_file
if expr "$1" : "Mark" > /dev/null
     then echo   Matching word is Mark
elif expr "$1" : "John" > /dev/null
     then echo   Matching word is John
elif expr "$1" : "Dennis" > /dev/null
     then echo   Matching word is Dennis
else echo * * * No match * * *
fi
$ _
```

Note that with **elif** only one **fi** statement is required. The output of the **expr** command is redirected to **/dev/null**, which can be considered as a dumping site for the unwanted output.

## Testing files, quantities, and strings

**test**
**[ ]**

The **test** command is used in conjunction with commands such as **while**, **if**, and **until**. These commands evaluate the **test** command and take action based on the return-value of the **test** command. The arguments of the **test** command form an expression, which is evaluated. If the result is true, the shell returns a value of zero; if the result is false, the shell returns a non-zero value. The **test** command may be used to evaluate any of the following: files; quantities; strings.

In versions of UNIX earlier than System V, the word **test** is used in front of the expression to be evaluated; beginning with System V, the expression is surrounded by a pair of brackets.

## Testing files

You can evaluate files in any of five ways:

| | | | |
|---|---|---|---|
| | `-r` | | File exists and can be read |
| | `-w` | | File exists and can be written to |
| **[** | `-f` | *filename* **]** | File exists and is *not* a directory |
| | `-d` | | File exists and *is* a directory |
| | `-s` | | File is not empty—its size is non-zero |

If you are concerned about maintaining portability with scripts written in earlier versions of UNIX, use the word **test** instead of brackets (for example, **test -f** *name*). This construction is still supported by System V.

In the example below, you can see that **if** commands are nested and that **test** commands are used to check an argument file's status.

```
$ cat copyall
num=0                              [Set the number of unreadable files to
                                    zero]

if [ -d $HOME/$1 ]                 [Check for the existence of the direc-
                                    tory]

then :                             [If the directory exists, do nothing]
else mkdir $HOME/$1                [If it doesn't exist, create it in your
fi                                  home directory]
for eachone in *                   [Check filenames in the current direc-
                                    tory]

do  if [ -f $eachone ]             [Is this a file?]
 then if [ -r $eachone ]           [If so, is it readable?]
  then cp $eachone $HOME/$1        [If so, copy it to the directory typed]
  else num=`expr $num + 1`         [Otherwise, add 1 to the total number
  fi                                of unreadable files in the current di-
 fi                                 rectory]
done
echo "$num non-accessible files"
$ _
```

This program (**copyall**) allows you to copy each accessible, non-directory ordinary file from your current directory to a directory named by you in your home directory. For example, if you invoke **copyall** with the command line

```
$ copyall newdir
```

the program will copy each readable file in your current directory to subdirectory **newdir** in your home directory. If **newdir** doesn't yet exist, it will be created. If it does exist, then each file in your current directory is checked (first for an ordinary file, then for being readable). If it is both (that is, if it is a readable file), it will be copied to **newdir**; if is a non-readable file, **copyall** increment the total of unreadable files (**num**) by one. After all files have been checked, **copyall** will display the number of unreadable files.

## Comparing quantities

You have a different set of expressions to check the relationship between two numbers:

$$[\ \texttt{A} \left\{\begin{array}{l}\texttt{-eq}\\ \texttt{-ne}\\ \texttt{-ge}\\ \texttt{-le}\\ \texttt{-gt}\\ \texttt{-lt}\end{array}\right\} ]\ \texttt{B}$$

| | |
|---|---|
| Is A equal to B? | $A = B$? |
| Is A not equal to B? | $A \neq B$? |
| Is A greater than or equal to B? | $A \geq B$? |
| Is A less than or equal to B? | $A \leq B$? |
| Is A greater than B? | $A > B$? |
| Is A less than B? | $A < B$? |

Once again, in each case, the shell returns a zero if the statement is true; a non-zero if it is false.

The following shell program tests the size of all files in the current directory, then displays the names of all files that contain at least 10,000 words:

```
$ cat bigfiles
count=10000
for i in *
     do size=`wc -c < $i`
     if [ $size -ge $count ]
          then echo "$i size: $size"
     fi
done
$ _
```

This is how the program works: first variable **count** is set to 10,000. The **for** command checks for the existence of the next file in the current directory before assigning it to variable **i**: if there are no more files, the program will exit; otherwise, the next file is selected. The expression **size=‘wc -c <$i‘** is a command substitution, which counts the number of characters in the file currently assigned to **i**. The value of the variable **size** is then tested against the value of **count**. The expression **$size -ge $count** will be true only if **$size** is greater than or equal to 10,000. If so, the name of the file will be displayed on the screen, along with its length (**echo "$i size: $size"**).

In some cases, the program may want to check arguments before processing, as in this example:

```
if [ $# -eq 2 ]
     then echo PROCESSING
     elif [ $# -lt 2 ]
          then echo 'missing second filename'
     else echo 'too many arguments'
fi
```

The expression **$# -eq 2** will be true only if the shell passes exactly two arguments to this program. The expression **$# -lt 2** will be true only if the number of arguments is less than two.

## Comparing strings

Expressions that are used to test strings of characters are as follows:

| | |
|---|---|
| **[ -n** *string* **]** | The string exists |
| **[ -z** *string* **]** | The string doesn't exist |
| **[** *string_1* **=** *string_2* **]** | The two strings are the same |
| **[** *string_1* **!=** *string_2* **]** | The two strings are not the same |

This example checks the existence of the second argument, which is passed by the shell to this program:

```
if [ -n "$2" ]
     then echo PROCESSING
     else echo 'missing second filename'
fi
```

Note that variable **$2** must be enclosed in double quotes so that the **test** command will recognize it as a character string. The following expression also checks the existence of the second argument string:

```
if [ "$2" ]
```

Let's assign two variables for comparing strings in an example:

```
Y='        abcd'
X=abcd
```

As you may recall from Chapter 21, "Introduction to the Bourne Shell," whenever a string of characters is enclosed in double quotes, special characters retain their meanings and spaces are treated as characters. If you don't enclose the string in quotes, spaces will be omitted from its string. That is why the following program finds strings X and Y to be the same:

```
if [ $Y = $X ]
then echo 'they are the same'
else echo 'they are different'
fi
```

However, when variables **X** and **Y** are enclosed in *double quotes*, **[** will recognize the spaces in string **Y**, and the following program will find strings **X** and **Y** to be different:

```
if [ "$Y" = "$X" ]
then echo 'they are the same'
else echo 'they are different'
fi
```

If variables **X** and **Y** are enclosed in *single quotes*, the dollar sign ($) loses its meaning as a special character. Therefore, in the following statement, **test** compares the two-character strings $Y and $X, rather than the assigned values of the strings **$Y** and **$X**:

```
[ '$Y' = '$X' ]
```

Note the following points about string comparisons:

1. You must surround an equal sign (=) or a not-equal sign (!=) with spaces.

2. The two symbols in the not-equal sign (!=) must not be separated from each other.

3. The expressions **-z** and **!-n** are equivalent.

## Compound testing

The primitives described above can be combined with the **not**, **and**, and **or** operators to obtain more flexible flow-control in shell programming:

**!** Unary **not** operator inverts the truth-value of an expression.
**-o** Binary **or** operator returns *true* if either of the surrounding statements is true.
**-a** Binary **and** operator returns *true* only if both of the surrounding statements are true.

In the following example, the **not** operator inverts a true value (zero) to a false value (non-zero):

```
if [ ! true ]
```

which provides the same result as the following expression:

```
if [ false ]
```

Therefore, the following expression is true if string $2 exists:

```
if [ "$2" ]
```

The expression below is true if string $2 doesn't exist:

```
if [ ! "$2" ]
```

The **and** and **or** operators can also be used to test several expressions in a single test statement, like this:

```
$ cat rw.file
 if [ -f $1 -a -r $1 -o -w $1 ]
    then echo "File $1 is accessible"
    elif [ -d $1 ]
      then echo "$1 is a directory"
      else echo "File $1 is not accessible"
 fi
$ _
```

The message "file $1 is accessible" is displayed only if the argument file (**$1**) is a normal file that is either readable or writable. The expression **[ -d $1 ]** will be true if the argument file is a directory.

## 24.3 Other programming techniques

### MULTI-WAY BRANCHING **case**

The **case** command provides multi-way branching for a shell program. The general format is as follows, with **esac** (**case** spelled backwards as the final delimiter):

```
case < string> in
       S1) < command list> ;;
       S2) < command list> ;;
                  .
                  .
                  .
       Sn) < command list> ;;
esac
```

The **case** command compares the pattern *string* to each pattern $S_1$, $S_2$,..., $S_n$. If it finds a match, it will execute the list of commands that follows the matching pattern, with execution ending at the double semicolon (;;). Program syntax requires a double semicolon at the end of each command list and **esac** at the end of the entire **case** command. Here is an example:

```
echo -n 'Please enter your selection (a, b, c, or d) : '
read entry
case $entry in
     A|a) echo 'Average' ;;
     B|b) echo 'Better' ;;
     C|c) echo 'Conditional' ;;
     D|d) echo 'Definite' ;;
     *) echo 'Please type A, B, C, or D' ;;
   esac
```

The **read** command accepts input from your keyboard, then assigns this input to variable **entry**. The **read** command stops reading when you press the (RETURN) key. If you type ***A*** (or ***a***), then **case** will execute

```
echo 'Average'
```

by displaying `Average` on the screen. If you type ***B*** (or ***b***), **case** will display `Better`, and so on. The asterisk (*) (used here to indicate any other entry) must be placed at the end of the list. Otherwise, **case** will not be able to select one of the valid entries (A, B, C, or D).

## PERFORMING TERMINAL CONTROL **tput**

Suppose you are using an ADM-3A terminal and the following entry has been compiled and stored in **terminfo/a/adm3a** (see Chapter 33, "Terminals and Printers," for details):

```
adm3a|3a|lsi adm3a,
        lines#24, cols#80, am,
        cup=\E=\%p1\%' '\%+\%c\%p2\%' '\%+\%c,
        cr=^M, home=^^, bel=^G, clear=^Z$<1>,
        cub1=^H, cud1=^J, ind=^J, cuu1=^K, cuf1=^L,
```

A System V command called **tput** allows you to activate most of the terminal functions named in this entry. You can either type **tput** on a command line or insert it into a shell script. As an example, since the function for clearing the screen and homing the cursor (**clear**) has been defined for your terminal in **terminfo**, you can enter the following command line at any time to clear your screen and home the cursor:

```
$ tput clear
```

Since the functions for turning high intensity on (**smso**) and off (**rsmo**) have been defined, you can include the following lines in a shell script to display a message in high intensity and then return the display to normal intensity:

```
tput smso
     echo "You have entered the correct value"
tput rmso
```

You can use the **tput** command to activate only functions included in the **terminfo** entry for the terminal you are using at the time.

## INTERRUPT-HANDLING **trap**

Interrupt signals are generated by various abnormal events. While the shell handles most of these signals itself, it passes the following signals to the shell procedure that is active at the time the interrupt takes place:

- Signal 1—Generated by a hangup
- Signal 2—Generated by the DEL key
- Signal 9—Generated by **kill -9** *pppp* (kill interrupt)
- Signal 15—Generated by the **kill** command itself with no option (software termination interrupt)

The shell procedure may choose to catch or ignore one of these signals (except the kill signal). If your program does not catch interrupt signals, upon receiving one of those interrupts, the shell will terminate the shell procedure currently invoked. Sometimes, this might cause inconvenience, such as leaving some temporary files that were created by this shell procedure. To avoid this inconvenience, use the **trap** command to catch interrupt signals, like this:

```
trap ' command'list'   signals
```

The command list must be enclosed in quotes. If there is more than one command, individual commands must be separated by semicolons (;). More than one signal can be specified (separated by spaces). Here is an example:

```
$ cat zap_temp
trap 'rm temp; exit' 1 2 15    [interrupt handling]
for i in $HOME/zfile/*
     do
          if expr "$i" : "$HOME/zfile/za" > /dev/null
          then cat $i >> temp
          fi
     done
if [ -f temp ]
     then nroff -50 temp > newfile
     rm temp
fi
$ _
```

In this program, whenever one of the three interrupts 1, 2, or 15 occurs, the shell removes file **temp** before exiting from the program.

### Debugging a shell procedure

As mentioned earlier in this chapter, the shell allows you to monitor your shell program so that you can see where your mistakes are. To monitor a shell program, add one of these two options to the the **sh** command when you invoke it:

**-x** Execution option: To see commands and arguments as they are executed.

**-v** Verbose option: To display input lines while they are being read.

For example, to debug the shell procedure **track**, type this:

```
$ sh -x track
```

You can also bundle the options, as shown here:

```
$ sh -vx track
```

## 24.4 Summary

In this chapter you learned about the **true**, **false**, and **exit** commands, and the commands for looping (**while**, **until**, **break**, **continue**, and **for**); the commands used to form a conditional statement (**if**, **then**, **else**, and **elif**); the command used to test files, strings, and quantities (**test**, along with a number of comparators and operators); and a few other tools for writing shell procedures (the **case** command, the **trap** command, and the options **-x** and **-v** for monitoring a shell procedure being executed).

A *loop* is a segment of a program that executes repeatedly until it is stopped by some indication that a condition has been met. The **true** command always returns an exit status of true; the **false** command always returns an exit status of false. Use the **while** command to continue executing a loop as long as the exit status is true; use the **until** command to continue executing as long as the exit status is false. When one loop is placed inside another, the loops are said to be *nested.*

To terminate execution of an inner loop and return to an outer loop, use the **break** command, giving an optional nesting level number if desired. To continue execution at an enclosing **for** or **while** loop, use the **continue** command, giving an optional nesting level number if desired. To abort the entire procedure, use the **exit** command. To construct a loop on a set of assigned values to a given variable, use the **for** loop construction. The **if** command allows you to set up a number of branching options in a shell procedure.

## Testing files, strings, and quantities

You can use the **test** command (in System V denoted by brackets) with one of five different options to determine whether a file can be read from or written to, whether it is a directory, or whether it is empty. You can also use the **test** command with one of six two-letter options to compare two quantities (equal, not equal, and so on). In addition, you can use the **test** command to determine whether a single string exists or whether two strings are the same or not. Finally, you can form compound tests with symbols for **and** (**-a**), **or** (**-o**), and **not** (**!**).

## Other programming techniques

You can form a multi-way branching list with the **case** command, using pairs of semicolons to terminate individual command lists and the **esac** command to terminate the entire list.

You can use the **tput** command to activate terminal functions that have been defined for your terminal in the terminal information database **terminfo**. You can use the **trap** command to handle various system interrupts. Two **sh** command options allow you to display commands and arguments as they are being executed (**-x**) or display input lines while they are being read (**-v**).

# 25

# Introduction to the C Shell

In Chapters 21 to 24, you learned about the *Bourne shell* (the original shell developed by Stephen R. Bourne at Bell Laboratories) and the **sh** command, which is used to invoke Bourne shell procedures. In this chapter we'll talk about a newer shell called the *C shell.* The C shell, developed by William N. Joy at the University of California, is one of the major Berkeley enhancements to UNIX.

Officially, System V supports only the Bourne Shell. However, many System V installations also support the C shell. If yours is one of them, you can use the **csh** to invoke C shell procedures. Unless you are already in the C shell, you should ask the system administrator for your UNIX system to set up your account so that the C shell is your login shell. To do this, the administrator must change an item inside a file called /etc/passwd. Somewhere in this file, there will be a line with your **login** name; the last item of this line must be changed from from /bin/sh to /bin/csh.

## 25.1 Initialization files

To make the C shell work properly with your terminal, you must have two files in your home directory: .login .cshrc

These two files will be executed by the C shell every time you log into the system (.cshrc first). These files will now be explained in detail.

### The C shell read command file *.cshrc*

The .cshrc file is executed whenever a new C shell is created—when you first log into the system or when you invoke a C shell procedure. For this reason, this file contains mostly variables that can be reinitialized for the newly created shell. Here is a typical .cshrc file:

```
set prompt = "<\!> "
set history = 11
alias   h      history
alias   cdl    cd ’\!*; l | more’
alias   ch     cd /usr/robin/book/\!
alias   ch8    cd /usr/robin/book/chap8
alias   ch9    cd /usr/robin/book/chap9
alias   ch10   cd /usr/robin/book/chap10
alias   ch11   cd /usr/robin/book/chap11
```

These lines will be explained in detail in the sections that follow.

### THE LOGIN FILE **.login**

The only time the **.login** file is executed by the C shell is when you log into the system. Therefore, this file should contain only those commands that need to be executed one time only (such as the ones for setup, command search path, and terminal type). Here is a typical **.login** file with the lines spread apart to leave room for explanations:

```
stty ff0 cr0 -tabs erase "^H"
```

This line is to set up your tty line: no delay after a formfeed or carriage return, replace tabs with spaces, use CTRL-H to erase.

```
set path=(/test/bin /bin /usr/bin $HOME/bin .)
```

This line is to set up the command search path. In this case, the shell will search (in this order) **/test/bin**, **/bin**, **/usr/bin**, and subdirectory **/bin** of your home directory for commands.

```
setenv TERM vt52
```

This line sets the *terminal* variable **TERM**. In this case, your terminal will be known by the system as a VT50, with all of the characteristics described under `vt50` in **/etc/termcap** (or **/etc/terminfo/v**).

```
echo  "today is `date`"
```

This line displays the date each time you login.

The *set environment* command (**setenv**) is explained in Chapter 26, "C Shell Variables," in the section on "Variables as Arrays," page 402.

## 25.2 Explanations of individual items

### SETTING YOUR OWN PROMPT **prompt**

The C shell, like the Bourne shell, allows you to set the prompt that appears for each command line. However, the C shell allows a *dynamic* prompt, which changes for each new command line. You can create line-numbering in your prompts by using the line number of the command line, known as the *event*, and indicated by an exclamation point ( ! ). This is what has been done in the sample **.cshrc** file above, which calls for displaying the current event within a pair of angle brackets: < >. (You have to type a backslash in front of the exclamation mark ( ! ) to prevent it from being interpreted as an event specifier by the shell.) Then, with your prompt set this way, this is how your prompt will look immediately after you log in:

```
<1> _
```

This event number is incremented each time you execute another command, thereby counting your command lines for you. If you do not assign anything to the variable **prompt**, the C shell will assign a percent sign (%) by default.

## SETTING THE NUMBER OF EVENTS TO BE SAVED **history**

On the second line of the sample **.cshrc** file you find this entry:

```
set history = 11
```

This assigns a value of 11 to the variable **history**. This number determines the number of events to be saved for later reference (that is, the size of the **history** list). Thus, the C shell keeps on this list the command lines of the last 11 events, which can be recalled at your request. You can set the variable **history** to any number, as long as it does not exceed the limit for your system (typically 25). You can change the setting of **history** at any time with the **set** command:

```
<2> set history = 5
<3> _
```

After this, until variable **history** is reassigned again, it will record only the last five events. You can also invoke **history** as a command to display a *history list*, as illustrated here:

```
<10> history
     5  cd book/XENIX
     6  ls -l
     7  cp chapter23 /usr/dennis/backup
     8  vi chapter23
     9  ls /usr/dennis/backup
<11> _
```

The five most recent commands are shown after their event numbers. These event numbers can now be used to reinvoke command lines in the history list. Furthermore, by using the history substitution command (explained in the next section), you can modify a recent command before executing it.

## 25.3 Reinvoking previous commands

Being able to reinvoke the past event means that you don't have to retype the whole command string, which may be very long and involved. To reinvoke the past event, construct a **redo** command by typing an exclamation mark ( ! ), followed by a reference to an event. The exclamation mark tells the C shell to look for the command line on the history list. There are four ways to refer to an event:

- Double exclamation marks (the previous event)
- The actual event number
- A relative event number
- A command name

### Reinvoking the most recent event **!!**

The quickest way to reinvoke the most recent event is to type double exclamation marks ( ***! !***) as the current command. (Since you entered **history** as event number 10 in the previous section, the next event number will be 11.) After you enter the two exclamation marks, the shell will first display the command string being replayed, then the results, as shown here:

```
<11> !!
history              [The previous event: number 10]
      6  ls -l
      7  cp chapter23 /usr/dennis/backup
      8  vi chapter23
      9  ls /usr/dennis/backup
     10  history
<12> _
```

### Reinvoking an event by number **!n**

In the following example, you reinvoke an event by typing its number. Let's invoke the command identified as event number 9:

```
<12> !9                             [Reinvoke event number 9]
ls /usr/dennis/backup               [Event number 9 is recalled: ls]
micro.back      entry.back          [Results of the ls command]
changes.back    files.back
```

```
<13> _
```

## Reinvoking an event relative to the current event **!-***i*

In **ed** and **vi** you used relative line numbers (for example, **.-5** means five lines back from the current line). The concept is exactly the same with the C shell (**!-5** means five events back from the current event). Here are two examples with relative event numbers:

```
<13> !-6
cp chapter 23 /usr/dennis/backup
<14> !-3
history
      8  vi chapter23
      9  ls /usr/dennis/backup
     10  history
     11  history
     12  ls /usr/dennis/backup
     13  cp chapter 23 /usr/dennis/backup
<15> _
```

## Reinvoking an event with a search string **!***string*

You can also type a search string after the exclamation mark, indicating that the C shell is to search for a matching command line. If the C shell finds a match, the event is reinvoked; if it doesn't, the message "*string*: Event not found" will appear on your screen. Here are two examples:

```
<15> !l
ls /usr/dennis/backup
micro.back     entry.back
changes.back   files.back
<16> !s
s: Event not found
```

Since **ls** begins with **l**, the reexecution command **!l** resulted in a successful match (event number 12), so the C shell displayed the results. The next attempt (**!s**) failed because no command string started with **s**. However, the C shell provides a way to search for a match anywhere on the command line. Type a question mark after the exclamation point, as shown here:

```
<17> !?s
ls /usr/dennis/backup
micro.back     entry.back
changes.back   files.back
<18> _
```

The question mark tells the C shell to look for a match with **s** anywhere on the command line.

## 25.4 Selecting individual arguments

Like the Bourne shell, the C shell splits every input line into words at blanks and tabs, assigning each word a number. Again, the command name is numbered zero, then the first word following is numbered one, the second word two, and so on. For example, the following command line contains a command (word 0), a first argument **-l** (word 1), and a second argument **/usr/dennis/backup** (word 2):

```
ls -l /usr/dennis/backup
```

The C shell's numbering scheme allows you to select a specific word on a command line. Simply append a colon, followed by a number, to a reinvocation command (a command that begins with an exclamation point). In the following example, we first set up a command (event number 18), then select word 1 from this command line (event number 19):

```
<18> echo word-1 word-2 word-3 word-4 word-5 word-6
word-1 word-2 word-3 word-4 word-5 word-6
```

Now we select the second word from the input line of event 18:

```
<19> echo !18:1
echo word-1        [The C shell first displays the command]
word-1             [then it displays the results of the command]
```

### Selecting the first argument ^

Another way to obtain the same result is by replacing **1** with a caret (^), as shown here:

```
<20> echo !18:^
echo word-1
word-1
```

### Selecting the last argument $

The dollar sign (**$**) can be used to address the last word of a command line, as shown here:

```
<21> echo !18:$
echo word-6
```

```
word-6
```

### SELECTING A RANGE OF ARGUMENTS *n-m*

To address a range of words, use the hyphen, as shown in this example:

```
<22> echo !18:2-5
echo word-2 word-3 word-4 word-5
word-2 word-3 word4 word-5
```

### SELECTING ALL ARGUMENTS *

To address every word of an input except zero (that is, to select every argument of the command, but not the command name itself), use an asterisk (*) to make a wild card selection, as shown here:

```
<23> echo !18:*
echo word-2 word-3 word-4 word-5 word-5 word-6
word-1 word-2 word-3 word-4 word-5 word-6
```

### OMITTING THE COLON WITH SYMBOLS

Whenever you use a caret (^), dollar sign ($), or asterisk (*) in place of an explicit word designator, you can omit the colon (:) between the event number and the word designator. For example, here is another way to type events 20, 21, and 23:

```
<20> echo !18^
echo word-1
word-1
<21> echo !18$
echo word-6
word-6
<23> echo !18*
echo word-2 word-3 word-4 word-5 word-5 word-6
word-1 word-2 word-3 word-4 word-5 word-6
```

## 25.5 Modifying a command line

The C shell allows you to reinvoke an *event* in the history list (a command line) with modifications, which can spare you a lot of extra typing. To modify an event, type a modifier after the colon (:) in place of a word number. You can substitute one string for another, repeat a string, remove prefixes and suffixes, preview an event, or protect an event from further changes.

## REPLACING A CHARACTER STRING **s**

Use the substitute modifier **s** the way you have used the **s** command with **ed** and **vi** to replace one character string with another. As an example, let's reinvoke event 23, then make a substitution:

```
<24> !23
echo word-1 word-2 word-3 word-4 word-5 word-6
word-1 word-2 word-3 word-4 word-5 word-6
```

Now we'll use the substitute modifier **s/w/W** to substitute `W` for `w` (first occurrence only):

```
<25> !!:s/w/W
echo Word-1 word-2 word-3 word-4 word-5 word-6
Word-1 word-2 word-3 word-4 word-5 word-6
```

As with the **s** (substitute) command in **ed**, only the first occurrence of `w` in `word-1` is affected. To convert all occurrences of `w` to `W`, use the global modifier **g**, as shown here:

```
<26> !23:gs/w/W
echo Word-1 Word-2 Word-3 Word-4 Word-5 Word-6
Word-1 Word-2 Word-3 Word-4 Word-5 Word-6
```

You can modify only a selected word by including its number. The following command affects only the second word:

```
<27> echo !23:2:s/word/WORD
echo WORD-2
WORD-2
```

If you attempt to modify a non-applicable word of a string (such as word-9, in this example), a "Modifier fails" message will be displayed.

## REPEATING THE PREVIOUS SUBSTITUTION **&**

The **&** modifier allows you to repeat the last substitution without having to retype the entire command. Suppose that event 25 was just invoked:

```
<25> !!:s/w/W
echo Word-1 word-2 word-3 word-4 word-5 word-6
Word-1 word-2 word-3 word-4 word-5 word-6
```

We could use the **&** modifier for the next event:

```
<26> !25:&
echo Word-1 Word-2 word-3 word-4 word-5 word-6
Word-1 Word-2 word-3 word-4 word-5 word-6
```

The **&** modifier affects only Word-2 this time.

### Removing the last name from a pathname **h**

Use the **h** modifier to remove the last name in a pathname. For example, let's create a pathname:

```
<28> echo /usr/dennis/backup.new
/usr/dennis/backup.new
```

Then we can use the **h** (head) modifier to remove **backup.new** from the pathname, like this:

```
<29> !!:h
echo /usr/dennis
/usr/dennis
```

### Removing the suffix from a pathname **r**

Use the **r** modifier to remove the suffix (in the form of *.sss*) from a string. Let's use the **r** modifier to remove `.new` from the pathname in event 28:

```
<30> !28:r
echo /usr/dennis/backup
/usr/dennis/backup
```

### Removing the prefix from a pathname **t**

Use the **t** (tail) modifier to remove the prefix from a pathname. Let's use the **t** modifier on the pathname in event 28:

```
<31> !28:t
echo backup.new
backup.new
```

### Previewing a command line **p**

To display a command line without executing it, use the **p** (print) modifier to preview it. For example, let's display the command string of event 28:

```
<32> !28:p
echo /usr/dennis/backup.new
<33> _
```

## PREVENTING FURTHER MODIFICATION **q**

Use the **q** modifier to *quote* a command line (that is, to protect it from further modification). After you execute event 33, any attempt to modify either event (28 or 33) will result in an error message ("modifier failed"):

```
<33> !28:q
echo /usr/dennis/backup.new
/usr/dennis/backup.new
```

## ALIASED COMMANDS **alias**

As explained earlier in this chapter, the C shell splits command input into words. Each of these words is checked for a match against the C shell's *alias list*, which provides a list of abbreviated commands, called *aliased commands*. If the C shell finds a match, it substitutes the full command line for the aliased command; if the C shell does not find a match, no substitution takes place. (You may recall that seven of the nine lines placed in the file **.cshrc** at the beginning of this chapter were intended for the "alias list.")

## CREATING A CUSTOM COMMAND

The alias list allows you to create either a new name for an existing or your own new command, which consists of a compound command string. For example, you can create a new **d** command to replace the **pwd** command, so that when you type ***d*** RETURN, the result will be the same as if you had typed ***pwd*** RETURN. Here's how to do this:

```
<34> alias d pwd
<35> d              [Invoke the new d command]
/usr/book/XENIX.1
```

Note that creating a new command name does not prevent you from continuing to use the old name. You can now use either **d** or **pwd**.

## REMOVING AN ALIASED COMMAND **unalias**

If you want to remove the **d** command above from the alias list, use the **unalias** command, as shown here:

```
<36> unalias d
```

```
<37> d
d: command not found
```

## Storing aliased commands

The most common use of the **alias** command is to allow you to type shorthand commands to replace frequently-used and excessively long commands. To preserve your aliased commands from one session to another, it's most convenient to store your own alias list in your **.cshrc** file, which is read every time a new subshell is created.

You can also use **alias** as a command to display your alias list, as shown here:

```
<38> alias
alias ch8   cd /usr/robin/book/chapter8
alias ch9   cd /usr/robin/book/chapter9
alias ch10  cd /usr/robin/book/chapter10
aias  ch11  cd /usr/robin/book/chapter11
```

## Invoking an aliased command

Once you have **ch8**, **ch9**, **ch10**, and **ch11** stored in your alias list, you can invoke them as your own private commands. For example, you can now invoke **ch10** to move to the directory called **/usr/robin/book/chap10**. Let's try it:

```
<39> ch10
<40> pwd
/usr/robin/book/chapter10
```

## Making a dynamic selection

You can set up a variable command that allows real-time selection at the time of invocation, like this:

```
alias  ch  cd /usr/robin/book/\!*
```

The aliased command **ch** allows you to pass an argument to be used as a part of the change directory (**cd**) command. The exclamation mark and asterisk must be quoted to prevent them from being interpreted by the shell. However, they are still passed to the alias. Let's invoke **ch** with arguments:

```
<39> ch chapter10
<40> pwd
```

```
/usr/robin/book/chapter10
<41> ch chapter9
<42> pwd
/usr/robin/book/chapter9
```

## Combining commands

You can combine more than one command into one aliased command, as shown here:

```
alias   cdl     cd '\!*; ls -l | more'
```

Because this command list contains spaces, an asterisk, and a pipeline symbol, it must be enclosed within single quotes ( ' ). The exclamation mark ( ! ) must be quoted (preceded by a backslash) to prevent the C shell from interpreting it and passing it to the alias. When invoked, **cdl** changes to the directory that you type after **cdl** as an argument; then all the files in this directory are piped to the **more** command to be displayed on the screen. By using **more** (or **pg**), instead of **cat**, you can control the display of a large directory like /etc, as illustrated here:

```
<43> cdl /etc
total 398
-rwx------ 1 bin      1346 Sep  4  1982 accton
-rwxr-x--- 1 bin       157 Apr 28  1981 asktime
-rw-r--r-- 1 bin        21 Dec 19 15:40 checklist
     .
     .
     .
-rwx--x--x 1 bin      5404 Feb 24 04:45 mount
--More-- (23%)
```

At this point, the display stops until you press the space bar.

## Recursive substitutions

You can alias commands recursively to as many levels as you like. If you do this, the C shell will substitute the aliased command over and over again until it reaches the last one in the list:

```
<44> alias com.1 com.2
<45> alias com.2 com.3
<46> alias com.3 echo last command
<47> com.1          [Invoke com.1 command.]
last command
```

Suppose you changed the alias procedure of event <46> so that **com.3** is aliased to **com.1**. Then, if you invoke **com.1**, you will see how the C shell deals with this, as follows:

```
<46> alias com.3 com.1
<47> com.1
Alias loop.          [error message]
```

The C shell can detect that this is a infinite loop, and it warns you with an error message.

## The logout file **.logout**

You can arrange for a custom logout to occur each time you sign off by placing commands in a special file called **.logout**. These commands may be standard UNIX commands or custom commands that you have set up with a shell procedure. The commands may simply place a message on the screen, or they may remove all the temporary files that you have created during your session with UNIX. The file must reside in your home directory, not a subdirectory. Here is an example of a **.logout** file already created:

```
<48> cat .logout
echo You are leaving the system
echo Goodbye \!
```

## Ordinary logout

With these two commands in your **.logout** file, pressing (CTRL-D) to log out not only allows you to exit from the system, but also to display a brief message on the screen:

```
<49>                          [You press (CTRL-D)]
<49>  You are leaving the system
      Goodbye !
```

## Explicit logout

You may recall that (CTRL-D) serves several functions in UNIX—one being a signal to log out, another being the end-of-file code. If a command that reads keyboard input is designed to exit upon receiving and end-of-file code (EOF), it will terminate the process. Occasionally, this results in an accidental logoff. To prevent this kind of mistake, the C shell provides a mechanism that disables (CTRL-D) as the signal to log out and forces you to type ***logout*** instead. The mechanism is the variable called **ignoreeof** (ignore end-of-file). Just type ***set ignoreeof*** to enable **logout**.

```
<49> set ignoreeof            [Set variable ignoreeof]
<50>                          [You press (CTRL-D)]
Use "logout" to logout.
<50> logout
```

## 25.6 Summary

This chapter introduces the C shell, an enhanced version of the Bourne shell. You learned about the two files frequently used with the C shell (.cshrc and .login), setting your own prompt, reinvoking previous commands, selecting individual arguments, modifying a command line, abbreviating a command string, and setting up a logout file (.logout).

### Initialization files

The .login file contains commands that are to be executed every time you log in, identifying your terminal, modifying your default command search path, and possibly providing a login message. The .cshrc file contains commands that are to be executed whenever a new C shell is created, and may be used to set your prompt, set the number of events to be saved, and set new names for command strings.

Use the **prompt** variable to set your own prompt, using either words or symbols. Use the **history** variable to set the number of command lines (events) to be saved for future reference.

### Reinvoking previous commands

To reinvoke the most recent event, type ***!!***. To reinvoke any previous event, type an exclamation point, then the number, as in ***!17***. To reinvoke an event a certain number of command lines from the current line, type an exclamation point, a minus sign, and the number, as in ***!-4***. To reinvoke an event in the history list that begins with a certain string of characters, type an exclamation point, then the characters, as in ***!ls***. To allow a match anywhere in the event, type a question mark between the exclamation mark and the string, as in ***!?s***.

### Selecting individual arguments

To select an individual argument in a command string, add a colon and the number of the argument to the end of a reinvocation command, as in ***echo !18:4***. To select the first argument, type one of the following alternate commands: ***echo !18:1***, ***echo !18:^***, ***echo !18^***. To select the last argument, type a command like the following: ***echo !18:$*** (or ***echo !18$***). To select arguments from a range of numbers, type a command like this: ***echo !18:2-5***. To select all arguments, use the wild card symbol (*), as in ***echo !18:**** (or ***echo !18****).

## Modifying a command line

To replace one character string with another, use the substitute modifier **s**, as in this example: **!!:s/w/W** (replace **w** with **W**). To repeat the previous substitution, use the repeat modifier **&**, as in **!25:&**. To remove the last name in a pathname, use the head modifier **h**, as in **!!:h**. To remove the prefix from a pathname, use the tail modifier **t**, as in **!28:t**. To remove the suffix from a pathname, use the remove modifier **r**, as in **!28:r**.

To display a command line (event) without executing it, use the preview modifier **p**, as in **!28:p**. To protect a command line (event) from further modification, use the quote modifier **q**, as in **!28:q**.

## Aliased commands

To create an abbreviated name for a longer command, assign the longer command to the shorter one with the **alias** command. To remove an alias already assigned, use the **unalias** command. To store an alias permanently, place it in your .cshrc file along with other aliases, thereby forming an *alias list.* To view this list, type ***alias*** as a command. To invoke an alias in your alias list, simply type the alias form as a command. To allow arguments to be passed with an aliased command, use an exclamation point followed by a wild card character (*), both preceded by a backslash. The C shell allows you to combine more than one command into a single aliased command.

## The logout file

The C shell allows you to set up a logout file to run commands at the time you sign off the system. These commands can either perform housekeeping tasks or just display a message. To force you to logout by typing **logout** instead of (CTRL-D), include the command **set ignoreeof** in your .login file.

# 26

# C Shell Variables

## 26.1 Assigning a string variable

The method for assigning a variable in the C shell is somewhat different from the method in the Bourne shell. In the following example, the string `table` is assigned as a value of variable **VAL**:

```
<1> set VAL = table
```

In the C shell, the **set** command is required, and the equal sign (=) must be surrounded by spaces in many UNIX systems. As in the Bourne, type a dollar sign ($) and variable name to recall the assigned value. Thus, `$VAL` will yield the value of variable **VAL**:

```
<2> echo $VAL
table
<3> _
```

If a value to be assigned to a variable contains spaces, it must be enclosed within single quotes, double quotes, or parentheses, as shown here:

```
<3> set var1 = '$VAL A B'
<4> set var2 = "$VAL A B"
<5> set var3 = ($VAL A B)
<6> _
```

Each pair of symbols—`' '`, `" "`, `( )`—will have a slightly different meaning, as summarized here:

- *Single quotes*—Metacharacters are interpreted literally.
- *Double quotes*—Metacharacters retain their special meaning.
- *Parentheses*—Metacharacters retain their special meaning, and the characters enclosed form an array (discussed in the next section).

The following **echo** commands illustrate these differences:

```
<6> echo $varl
$VAL A B
<7> echo $varl
table A B
<8> echo $varl
table A B
```

```
<9> echo $varl[2]
 Subscript out of range
<10> echo $varl[2]
 Subscript out of range
<11> echo $varl[2]
 A
```

### Displaying assigned values **set**

By typing the **set** command by itself, you can display a listing of declared variables and their assigned values, as illustrated here:

```
<12> set
argv    ()
bk      /usr/book/
history 23
home    /usr/robin
path    (/test/bin /bin /usr/bin /usr/robin/bin .)
prompt  <!>
shell   /bin/csh
status  0
unmask  022
<13> _
```

### Removing a variable **unset**

The C shell keeps a list of all declared variables. To remove a variable from the list, use the **unset** command, as shown here:

```
<13> unset bk       [Remove variable bk from the list]
<14> set            [Display the variable list]
argv    ()
istory 23
home    /usr/robin
path    (/test/bin /bin /usr/bin /usr/robin/bin .)
prompt  <!>
shell   /bin/csh
status  0
unmask  22          [Variable bk is no longer in the list]
<15> _
```

## 26.2 Variables as arrays

Improving on the Bourne shell, the C shell treats any string value of a variable as an array—provided that the string value is enclosed in parentheses. This means that, even though the entire string shares a common name, you can access individual words in the string. In the example that follows, the variable **path** is the name of an array whose value is a command search path. As in the C language, each item of an array can be addressed individually by pointer: the first item is refered to as **path[1]**, the second one is **path[2]**, the third one is **path[3]**, and so on. In the following example, we recall the third item:

```
<15> echo $path           [Recall all values of variable path]
/test/bin /bin /usr/bin /usr/robin/bin
<16> echo $path[3]        [Recall item 3]
/usr/bin                  [Third item in the search path]
<17> _
```

Since each item occupies a numbered position in the array, you can overwrite any item in a declared string variable. Type the **set** command and the array name, followed by the number enclosed in square brakets [ ]. In the following example, the first location of array **path** is overwritten with the string /usr/joe/bin (this becomes the first name in the path):

```
<17> set path[1] = /usr/joe/bin
<18> echo $path
/usr/joe/bin /bin /usr/bin /usr/robin/bin
<19> _
```

## Declaring an array

You must declare an array before you can use it. To declare an array, specify the array's name and size in a dummy **set** command. The C shell will reserve the number of locations that you have requested. You can then initialize the array with an actual string. In the example shown here, seven positions of array **temp** are set aside with null strings:

```
<19> set temp = ("  "  "  "  "  "  ")  [Seven pairs of single quotes]
<20> _
```

## Assigning positions

Once the positions in the array have been reserved, you can use **temp** as a storage location, to which strings can be written and from which strings can be read. Let's fill some of these locations with actual strings:

```
<20> set temp[1] = one
<21> set temp[3] = three
<22> echo $temp
one three
<23> _
```

You can also assign the value of an element in one array to an element in another, as shown here:

```
<23> set temp[2] = $path[1]
<24> echo $temp
one /usr/joe/bin three
<25> _
```

You can also retrieve any of these strings and assign them to another variable, as shown here:

```
<25> set VAL1 = $temp[1]
<26> echo $VAL1
one
<27> _
```

### DETERMINING WHETHER A VARIABLE HAS BEEN DECLARED ?

Special reserved variables are used to obtain information about assigned variables. The special variable **?** is zero (false) if a variable is undeclared and one (true) is it has been declared, as shown here:

```
<27> echo $?temp          [Test variable temp]
1                         [True: the variable has been declared]
<28> _
```

### DETERMINING THE SIZE OF AN ARRAY #

Use special variable **#** to determine the size of a variable (array). Let's check the size of array **temp**:

```
<28> echo $#temp [Test array temp]
7                [Array temp has seven positions]
<29> _
```

## 26.3 Assigning numeric variables

Just as the **set** command is used to assign *string* values to variables, the **@** command is used to assign *numeric* values to variables, as shown here:

```
<29> @ VAL = 3
<30> _
```

If you attempt to assign a non-numeric value to a variable with **@**, a syntax error message will be displayed, like this:

```
<30> @ VAL = abc
@: Expression syntax
<31> _
```

To list all declared numeric variables, use the **@** command alone, just as you did with the **set** command:

```
<31> @
val       3
num       5
integer   12
<32> _
```

## PERFORMING ARITHMETIC

Unlike the **set** command, the **@** command can be used to perform arithmetic and logical testing, like the **expr** command of the Bourne shell. You can use **@** with the following arithmetic operators:

| | |
|---|---|
| **+** | addition |
| **-** | subtraction |
| ***** | multiplication |
| **/** | division |
| **%** | remaind |

Here are some examples:

```
<32> @ A = 2 + 4              [Add 2 and 4 and assign the sum to A]
<33> @ S = $A - 3             [Subtract 3 from A and assign the differ-
                              ence to S                               ]
<34> @ M = $A * $S            [Multiply A by S and assign the product
                              to M                                    ]
<35> @ R = $M % 4             [Divide M by 4 and assign the remainder
                              to R                                    ]
<36> @ D = $M / 4             [Divide M by 4 and assign the quotient to
                              D                                       ]
<37> echo $A $S $M $R $D      [Display all five results]
6 3 18 2 4
<38> _
```

## INCREASING THE VALUE OF A VARIABLE

**++**
**+=*n***

Use the **++** operator to increase the value of a variable by one:

```
<38> @ D++             [Increase the value of D by one ]
<39> echo $D
```

```
5
<40> _
```

You can add any numeric value to an variable with the **+=** operator. For example, suppose variable **R** has a value of 2. The following example shows how you could add 5 to **R**:

```
<40> @ R += 5      [Add 5 to the value of R]
<41> echo $R
7
<42> _
```

## Decreasing the value of a variable

**--**

**-=***n*

Use the – operator to decrease the value of a variable by one:

```
<42> @ D--         [Decrease the value of D by one]
<43> echo $D
4
<44> _
```

You can subtract any numeric value from an variable with the **-=** operator. For example, using variable **D** with a value of **4**, the following example shows how you could subtract **3** from **D**:

```
<44> @ D -= 3      [Subtract 3 from the value of D]
<45> echo $D
1
<46> _
```

## Obtaining the one's complement of a variable

Use the tilde (~) to obtain the one's complement of the value of a variable (the opposite sign minus one), as illustrated here:

```
<46> R =~ $R             [Assign to R its own one's complement]
<47> echo $R
-8                       [This is the one's complement of 7 (−7 – 1)]
<48> _
```

## Logical testing

Logical tests work in the C shell like the **test** command in the Bourne shell to yield true (1) or false (0) as the exit-status of a test:

| | |
|---|---|
| **A <B** | True if A is less than B |
| **A >B** | True if A is greater than B |
| **A <= B** | True if A is less or equal to B |

**A >= B** True if A is greater or equal to B
**A == B** True if A is equal to B
**A != B** True if A is not equal to

Here is an example of a sequence of logical tests:

```
<48> @ A = 10; @ B = 15[Assign values to variables A and B]
<49> @ C = ($A <= $B)  [Test whether A is less than or equal to B]
<50> echo $C           [Display the result]
1                      [True: 10 is less than or equal to 15]
<51> _
```

The expression right of the equal sign must be enclosed in parentheses to prevent the shell from interpreting the less than sign (<). You can also construct more complex statements, as shown here:

```
<51> @ K = ($A + $B > $A * $B) [Is the sum greater than the prod-
<52> echo $K                    uct?                             ]
0                              [False: 25 is not greater than 150]
<53> _
```

As you can see, the return status of the expression of event 52 is a zero, which indicates that the expression is false.

## 26.4 Setting elements of a numeric array

Arrays were discussed earlier in this chapter in connection with *string* variables. Like the **set** command, the **@** command can also be used with pointers to select individual elements of a *numeric* array. Here is an example:

```
<53> @ scale = (1 2 3 4 5 6 7 8 9)
<54> @ scale[3] = 100     [Assign a value to the numeric array]
<55> echo $scale
1 2 100 4 5 6 7 8 9
<56> _
```

### Performing arithmetic

The **@** command allows you to perform arithmetic on an element of an array. In the following example, the values of positions 3 and 5 are added together, then the sum is stored in location 8:

```
<56> @ scale[8] = $scale[3] + $scale[5]
<57> echo $scale
1 2 100 4 5 6 7 105 9
<58> _
```

### Performing logical tests

You can also perform logical tests within a numeric array. Here, a logical test is performed on the values of positions 3 and 8 of array **scale**, with the logical result stored in position 9:

```
<58> @ scale[9] = ($scale[3] > $scale[8])
<59> echo $scale
1 2 100 4 5 6 7 105 0          [False: 100 is not greater than 105]
<60> _
```

Position 9 becomes 0, indicating *false*. Once again, the expression to the right of the equal sign in event 58 is enclosed in parentheses to prevent the *greater than* sign (>) from being interpreted by the shell.

## 26.5 Variables reserved by the C shell

The C shell uses the variables named in this section to record information and pass it to a child process. Some of the special variables are set by the C shell, and some can be changed by the user.

### Argument number *n* **$argv[*n*]**

The shell uses this variable to pass arguments to the C shell procedure in the same way that the Bourne shell passes positional parameters to the shell script. Therefore, **$argv[0]** is the name of the program being invoked, **$argv[1]** is the first argument, and so on. For example, suppose there is a C shell procedure **reflect** with the following contents:

```
<60> cat reflect
#                         [# indicates that this is a C shell procedure]
echo $argv[0] $argv[1] $argv[2] $argv[3]
set argv[3] = NEW
echo $argv[3]
<61> _
```

Let's invoke **reflect**:

```
<61> reflect one two three   [Invoke the program with three argu-
                              ments]
reflect one two three        [The command line just typed is echoed]
NEW                          [NEW replaces three as the third item]
<62> _
```

The C shell passes the positional parameters to **reflect**, and you can change any argument except the **echo** command argv[0]. However, you can still use **$1**, **$2**, ..., **$9** in place of **$argv[1]**, **$argv[2]**,..., **$argv[9]**.

### Number of arguments **$#argv**

The C shell uses this variable to store the number of arguments in the command just invoked. Let's change the **reflect** program so that it looks like this:

```
<62> cat reflect
#                                     [A C shell procedure]
echo $argv[1] $argv[2] $argv[3]
echo $#argv                           [Add this new line]
<63> reflect one two three
one two three                         [Only the arguments are echoed now]
3                                     [Number of arguments]
<64> _
```

The expression **argv[*]** refers to all of the positional parameters.

### Search path **$cdpath**

When you type the **cd** command with an argument as the target directory, the **cd** command uses the value of **cdpath** as the search path to search from the current directory for that target directory. If the subdirectory is found, this becomes your working directory. For example, suppose your current working directory is /usr/book and **cdpath** is set this way:

```
<64> set cdpath = $home/book/XENIX/chapter21
<65> _
```

and you type this:

```
<65> cd chapter21
<66> pwd                              [Verify your current directory]
/usr/book/XENIX/chapter21
<67> _
```

If **cdpath** is *not* set or the search fails, a message is displayed:

```
<67> cd XENIX
cd: No such file or directory
<68> _
```

The message appears because XENIX is not a subdirectory of your current directory (in fact, your current directory is a subdirectory of XENIX).

### Child process number **$child**

This variable carries the process number of a child process, which is forked in the background. The C shell unsets this variable when the child process terminates.

## Echo a command **$echo**

If this variable is set, any command you enter will be echoed (displayed on the screen) before being executed. This variable works like the -x option of the **sh** command (Bourne shell).

```
<68> set echo       [Declare variable echo]
<69> pwd            [You then type this]
pwd                 [Your command is echoed before being executed]
/usr/book/XENIX
<70> _
```

You can use the **unset** command to remove the effects of **echo**.

## Length of the history list **$history**

This variable was explained in Chapter 25, "Introduction to the C Shell." You can set **$history** to the size of the history list.

## Home directory **$home**

This variable, the same as the **$HOME** variable of the Bourne shell, can be abbreviated with a tilde (~). Its value is the pathname of your home directory, which is set by the shell. Here is an example:

```
<70> cp /usr/john/stats ~/saving
<71> _
```

This command causes file **stats** in directory **/usr/john** to be copied to directory **$home/saving**.

## Ignore end-of-file character **$ignoreeof**

Ordinarily, CTRL-D (or ^**D**) is defined as the end-of-file (EOF) character. If this variable is set, the shell will ignore the end-of-file character from your keyboard. This prevents you from being accidentally logged off. Then you can exit with the **logout** command (for a more detailed explanation, see the section "Explicit Logout," page 397, in Chapter 25, "Introduction to the C Shell").

## Prevent overwriting **$noclobber**

When this variable is set, the C shell prevents you from overwriting a file as the result of redirection of output. For example, suppose file **tempfile** resides in your current directory, and you type this:

```
<71> set noclobber              [Declare variable]
```

```
<72> echo I WRITE THIS LINE > tempfile
                                  [Redirection of output]
tempfile: File exists             [Overwriting is not allowed]
<73> _
```

Once **noclobber** has been set, the shell also will not create a new file when you append text to a nonexistent file, as shown below:

```
<73> echo I ADD NEW LINE >> newfile
newfile: No such file or dirctory
<74> _
```

## Remove special meanings automatically **$noglob**

When this variable is set, filename expansions such as *, ?, and ~ are treated as normal characters, allowing you to use them literally. Here is an example:

```
<74> echo *
file.1 file.2 file.3 tempfile [Files in this directory]
<75> set noglob               [Set variable noglob]
<76> echo *                   [The asterisk is now treated as an
*                              ordinary character]
<77> _
```

In event 74, the asterisk (*) is interpreted by the shell as a wild card character, allowing all filenames in the current directory to be displayed. The variable **noglob** is set in event 75 so that the shell treats the asterisk the same as any other character, as shown in event 76.

## Remove special meanings if no match **$nonomatch**

When this variable is set, the C shell first attempts to find a matching file with one of the filename expansion characters such as *, ?, and ~. If the shell finds a match, it passes that name to the calling program; if it cannot find a match, it passes the argument with the special character treated literally. If **$nonomatch** is *not* declared and the C shell fails to find a match, the C shell will display an error, as illustrated here:

```
<77> echo tempfile?
echo: No match
<78> set nonomatch        [Set variable nonomatch]
<79> echo tempfile?       [The echo command interprets the ques-
tempfile?                  tion mark literally]
<80> _
```

### Search path **$path**

The C shell uses the value of this variable as the command search path, assigning to it the default value **/usr/bin /bin**. Therefore, you can execute only files that reside in the current directory. This variable is explained in detail in Chapters 21, "Introduction to the Bourne Shell" and 25, "Introduction to the C Shell."

### Prompt symbol **$prompt**

The value of this variable is your prompt symbol. If you don't set **prompt**, the C shell will set `%` as the default value. This variable is explained in greater detail in Chapter 25, "Introduction to the C Shell."

### Pathname of the shell **$shell**

The value of this variable is the pathname of the shell. The variable **shell** reflects whatever is set in the file **/etc/passwd**, as shown here:

```
<80> echo $shell
/bin/csh
<81> _
```

The **/bin/csh** response indicates that you are using the C shell.

### Exit status of the last command **$status**

The value of this variable is the exit status returned by the command most recently invoked.

### Time to execute a command **$time**

When this variable is set, the shell displays the number of seconds to execute the most recent command (user time, system time, real time, and the percentage of real time spent on user time and system time combined):

```
<81> set time
<82> pwd
/usr/XENIX/book/chapter21
0.1u 0.4s 0:02 24%
<83> _
```

The numbers shown indicate user time (`0.1`), system time (`0.4`), and real time in seconds (`0:02`) required to process the **pwd** command, along with the percentage of real time spent on user time and system time $((0.1+0.4)/2)$.

### Terminal type **$TERM**

This variable, which is set with the **setenv** command, indicates the type of terminal you are using. You must set this variable before you can use **vi**, **more**, or one of several other programs.

### Process identification number **$$**

The value of this variable is the process identification number of the current process (discussed in greater detail in Chapter 21, "Introduction to the Bourne Shell").

### Complete messages **$verbose**

This variable works like the **-v** option of the Bourne shell when invoked with the **sh** command. When **verbose** is set, your commands will be echoed when read by the C shell.

### Permanent settings

If you want any of these variables set to a value of your choice every time you log in, place them in your .cshrc file, which is executed each time you log into the system and each time a new process is created.

## 26.6 Summary

In this chapter you learned about assigning values to string and numeric variables, working with variables as arrays, performing arithmetic and logical operations, and special C shell variables.

### Assigning string values

Use the **set** command to assign a string value to a variable. After the string value has been assigned, call the variable preceded by a dollar sign. Use the **set** command by itself by display all declared variables and their assigned values. Use the **unset** command to undeclare a variable.

The C shell treats the value of a variable as an array whenever the assigned value is enclosed in parentheses. This allows you to access individual elements of the array. Use null strings within single quotes to declare an array. Once declared, an array can be filled one position at a time.

Use a question mark between the dollar sign and the name of a variable in an **echo** command to determine whether or not a variable has been declared. The response will be either yes (1) or no (0). Use a pound sign

(#) between the dollar sign and the name of a variable in an **echo** command to obtain the number of elements in an array.

## Assigning numeric values

Use the **@** instead of the **set** command to assign a numeric value to a variable. After the string value has been assigned, call the variable preceded by a dollar sign. Use the **@** command by itself to display all declared numeric variables and their assigned values.

Use the operators **+**, **-**, *****, **/**, and **%** to perform arithmetic with numeric variables. Use the **++** and **+=n** operators with the **@** command to increase the value of a variable by one (or by n). Use the **–** and **-=n** operators to decrease the value of a variable by one (or by n). Use the ˜ operator to obtain the one's complement of a variable.

Use the operators <, >, <**=**, >**=**, **!=**, and **==** with the **@** command to perform logical test. The response will be either true (1) or false (0). Use an array selector in square brackets with the **@** command to assign a numeric value to an individual element in an array. You can use the **@** command to perform either arithmetic or logical tests on individual elements of an array.

## Variables reserved by the C shell

Nineteen variables are reserved for use by the C shell (see Appendix I, "Summary of the C Shell"). You can set some of these reserved variables in your **.cshrc** file, which is executed each time you log on and each time a new process is created.

# 27

# C Shell Procedures

## 27.1 Executing a file as a shell procedure

When used as a programming language, the C shell provides mechanisms similar to those used in the Bourne shell, but these mechanisms require slightly different program syntax. A file can be invoked as a C shell procedure in one of two ways:

- Make the file executable.
- Execute the **csh** command with the file's name as an argument.

### Making a file executable

Whenever you make a file executable with **chmod**, it becomes a command file. This means that you can execute it as a normal command within the directory where this file resides. Before executing a shell file, the C shell checks for a pound sign (#) as the first non-blank character. If this symbol is missing, the file will automatically be executed in the Bourne shell by default (see the example in the next section).

### Executing the **csh** command

Invoking the **csh** command with the name of the desired file as an argument is the surest way to have this file executed by the C shell, as shown here:

```
<1> csh filename
```

Like the Bourne shell, the C shell passes the filename as the value of **argv[0]** and any following words as **argv[1]**, **argv[2]**, and so on.

### File checking

The C shell has a method for checking files similar to the **test** statement of the Bourne shell. However, the format of the expressions, which may be used in **if**, **for each**, and **while** statements, is different. Each expression returns a status of either true (1) or false (0), depending on what is being tested:

| | | |
|---|---|---|
| **-e** | *filename* | true if the file exists |
| **-z** | | true if the size of the file is zero |
| **-f** | | true if the file is an ordinary file |
| **-d** | | true if the file is a directory |
| **-o** | | true if the user owns the file |
| **-r** | | true if the file is readable |
| **-w** | | true if the file is writable |
| **-x** | | true if the file is executable |

## 27.2 Forming conditional statements

### THE SIMPLE CONDITIONAL STATEMENT **if**

Like the **if** statement of the Bourne shell, the **if** statement of the C shell evaluates the status of the expression. If it is true, the shell executes the statement that follows; if it is false, the shell skips the statement that follows. Here is an example:

```
<2> cat checkfile.1
# This program checks for existence of the argument file
if -e $argv[1]  echo "File $argv[1] exists"
```

The pound sign (#) is mandatory; without it, the C shell cannot execute this file. As shown in the example, the line that begins with the pound sign may also contain comments, which will be ignored by the C shell.

The **if** statement evaluates the expression **-e $argv[1]**, which tests for the existence of an argument file. If an argument file exists, the shell will execute the statement

```
echo "File $argv[1] exists"
```

The message is enclosed in double quotes to preserve the special meaning of the dollar sign ($). A syntax error will result if the statement that follows the logical test is complex (that is, contains a pipeline or a group of statements).

### THE THREE-WAY CONDITIONAL STATEMENT **if**...**then**...**else**

This statement works just like the **if**...**then**...**else** statement used in the Bourne shell. This allows an alternative logical path for the program, based on the status returned from the expression that follows **if**. Here is an example:

```
<3> cat checkfile.2
# This is a modified version of checkfile.1
if -e $argv[1] then
   echo "File $argv[1] exists"
else
   echo "File $argv[1] not found"
endif
```

If the expression that follows the **if** statement is true (1), the command list between the **if** and **else** statements is executed; if it is false (0), the command list between the **else** and **endif** statements is executed. Notice that **endif** is used to close the **if**. . . **then**. . . **else** statement. If **endif** is missing, a syntax error will be displayed. In the example below, file checkfile.2 invokes itself and passes itself as its own argument:

```
<4> checkfile.2 checkfile.2
File checkfile.2 exists
```

## An additional alternative **else if**

The **else if** statement provides additional alternative logical paths for a C shell program. This statement works the same way as the **elif** statement of the Bourne shell. Here is an example:

```
<5> cat checktype
# This program checks the file-type of the argument file
#
if -f $arg[1] then
  echo "$argv[1] is an ordinary file"
  else if -d $argv[1] then
    echo "$argv[1] is a directory"
  else
    echo "$argv[1] is not found"
endif
```

The expression **-f $argv[1]** tests whether or not the first argument names an ordinary file. If it does not, the shell evaluates the expression that follows **else if** (**-d $argv[1]**) to determine whether or not the first argument names a directory. If it does not, the shell executes the command list between the **else** and **endif** statements. If any of the expressions is true, then the shell will execute the command list that follows immediately.

You can have more than one **else if** statement in a C shell procedure. The **else** statement is optional, and only one **endif** is needed to close the **if** control structure, no matter how many logical paths are given. Let's invoke **checktype** with checkfile.2 as its argument:

```
<6> checktype checkfile.2
```

```
checkfile.2 is a normal file
```

## 27.3 Forming loops

### The branching statement **goto**

The format of the branching statement **goto** is

```
goto label
```

The **goto** statement causes the C shell to search for a line within the file that contains *label:* (with a colon following), then transfers program control to this line. Here is an example:

```
<7> cat list.files
# This program lists the files in a directory
#
if -d $argv[1] then
  echo "$argv[1] is a directory"
  goto list
  else if -f $argv[1] then
    echo "$argv[1] is an ordinary file"
else
  echo "$argv[1] not found"
endif
list:
  echo "Contents of $argv[1]:"
  ls $argv[1]
  exit
```

In this program, if the expression that follows the **if** statement is true (the argument is a directory), the shell executes the **goto** *list* statement: the C shell will search for the line that starts with the label **list:**, then executes the commands that follow. In this case, the shell will display a list of all filenames in the directory named by **$argv[1]**. For example, directory **backup** is a subdirectory that contains four files: **file1.bk**, **file2.bk**, **file3.bk**, and **file4.bk**. Let's invoke **list.files** with **backup** as its argument:

```
<8> list.files backup
backup is a directory
Contents of backup:
file1.bk file2.bk file3.bk file4.bk
```

## THE LOOP CONTROL STATEMENT **foreach**

The **foreach** statement controls the loop structure, performing a function similar to the **for** statement of the Bourne shell. The format of the **foreach** is as follows:

```
foreach  variable (wordlist)
  command(s)
end
```

The value of *variable* is initialized to the first member of *wordlist* on the first iteration, then to the next member on each subsequent iteration. If the value of *variable* is not null, the shell executes the command list between the **foreach** statement and the **end** statement. Then the process is repeated until *variable* has a null value, indicating that there are no more words in *wordlist* to be assigned. In the example that follows, the shell copies each item in **buffer2** to **buffer1** one at a time without changing its order:

```
<9> cat cp_buff
# Copy data from buffer2 to buffer1
#
set buffer1 = ("  "  "  "  "  ")      [Initialize buffer1]
set buffer2 = (A B C D E F)            [Initialize buffer2]
set pointer = 1                        [Initialize the pointer]
#
foreach data ($buffer2)                [Read a word of buffer2]
  set buffer1[$pointer] = $data        [Write a word to buffer1]
  @ pointer++                          [Increment the pointer]
end
#
echo "buffer1 $buffer1"                [Display the name then the]
echo "buffer2 $buffer2"                [contents of each buffer  ]
exit
```

As explained in the previous chapter, before an array can be used, it must be declared with the desired number of elements. In this case, **buffer1** and **buffer2** are declared with six elements each. In the first iteration of the **foreach** loop, the value of variable **data** is initialized to the first element of array **buffer2**, which is A.

Since variable **pointer** has been initialized to be 1, the statement **buffer1[$pointer]** means **buffer1[1]**. Therefore, $data is written to **buffer[1]** on the first iteration. At the end of the loop, the statement **@ pointer++** increases the value of **pointer** by one. So in the second iteration, **buffer1[$pointer]** becomes **buffer1[2]**; in the third iteration, it becomes **buffer1[3]**; and so on.

After processing character F (i.e., transferring it from **buffer2[6]** to **buffer1[6]**, **foreach** finds no more characters left in buffer 2. So when

**foreach** attempts a seventh loop, assigment of the next character to **data** fails, and the loop terminates. This is what you will see on the screen when you invoke **cp_buff**, as shown here:

```
<10> cp_buff
buffer1 A B C D E F
buffer2 A B C D E F
```

## Looping while true **while**

The **while** statement evaluates the the expression that follows it in parentheses. If the status of the expression is true, the shell executes the command-list between the expression and the **end** statement. If its status is false, the shell halts execution of the **while** loop. The syntax of the **while** loop is

```
while  (expression)
  command(s)
end
```

In this example, the **while** loop is used to fill the buffer (which has been declared as an array) with numbers:

```
<11> cat fillup
# Fill an array with numbers in ascending order
#
set buffer = (0 0 0 0 0)       [Initialize buffer with zeroes]
set pointer = 1                [Initialize pointer at 1]
set count = $#buffer           [Set count to the buffer length (5)]
#
while ($count)                 [Test whether to begin an iteration]
  set buffer[$pointer] = $pointer
                               [Write a number into buffer]
  @ pointer++                  [Increment pointer by one]
  @ count---                   [Decrement count by one]
end
#
echo $buffer                   [Display the contents of the array]
```

The key to this loop is the value of **$count**, which is initialized to the number of elements in **buffer** (5), then decremented by one at the end of each loop. The **while** statement ends the loop when the value of **count** becomes zero, meaning that there are no more numbers left.

During the first iteration, since **$count** is not a zero, the shell executes the statements in the loop: the shell assigns to the first element of **buffer** the value of the pointer (1) (**set buffer[$pointer] = $pointer** becomes **set buffer[1] = 1**), the pointer is increased to 2 by the statement **@ pointer++**, and the count is decreased to 4 by the statement **@ count—**.

During the second iteration, the shell assigns 2 to **buffer[2]**, the pointer goes up to 3, and the count drops to 3; during the third iteration, the shell assigns 3 to **buffer[3]**, the pointer goes up to 4, and the count drops to 2; and so on until the fifth iteration. Here is a diagram that summarizes what happens during each iteration (with "before" and "after" values for the pointer and the count):

| Iteration | Value at | the Start | Value Assigned | Value at | the End |
|---|---|---|---|---|---|
| **Number** | **Pointer** | **Count** | **to the Array** | **Pointer** | **Count** |
| 1 | 1 | 5 | buffer[1] = 1 | 2 | 4 |
| 2 | 2 | 4 | buffer[2] = 2 | 3 | 3 |
| 3 | 3 | 3 | buffer[3] = 3 | 4 | 2 |
| 4 | 4 | 2 | buffer[4] = 4 | 5 | 1 |
| 5 | 5 | 1 | buffer[5] = 5 | 6 | 0 |

At the end of the fifth iteration, the count becomes zero, indicating that there are no more numbers left. When the **while** statement attempts to begin a sixth iteration, it discovers a zero in the statement **while ($count)**, and the loop ends. This is what you will see on the screen when you invoke **fillup**:

```
<12> fillup
1 2 3 4 5
```

## Breaking out of a loop **break**

The **break** statement of the C shell is similar to the **break** statement of the Bourne shell. In the C shell, the statement breaks to the nearest **while** or **foreach** loop. This example will help you with file management:

```
<13> cat backup
# Copy files from current directory to "$home/backup"
if -d #home/backup then    [Does directory backup exist?]
  goto COPY:               [If so, go to COPY]
else                       [If not,...
  mkdir $home/backup       create such a directory]
endif
COPY:
set total = 'ls'           [List all the files in total]
foreach file ($total)      [Test whether to begin another]
  if ($#argv == 1) then    [Is there an argument after backup?]
    if ($#argv[1] == $file) [If so and if it matches the current file,]
       break                [then stop here                           ]
  else
    cp $file $home/backup  [If not, then copy this file...
    echo '   $file copied' and display a message]
 endif
```

```
end
echo '* Backup completed *'
```

This procedure lets you back up all the files in the current directory to a directory in your home directory called **backup**—with one added feature. If you include a file name as an argument, **backup** will copy up to, but not including, the file named. For example, suppose your current directory contains these files:

```
apple          date           kiwi           orange
banana         fruit          lemon          peach
coconut        grape          melon          pear
```

If you invoke **backup** without an argument, all 12 files will be copied to **backup**. If you type one of these filenames after **backup**, then only those files in the directory that precede the file named will be copied, as shown here:

```
<14> backup fruit
   apple copied
   banana copied
   coconut copied
   date copied
 * Backup completed *
<15> _
```

## Continuing a loop **continue**

This statement directs program control to the nearest **while** or **foreach** loop. Here is a program that takes an argument and selects even numbers:

```
<15> cat select
# Select only even numbers from arguments entered
#
if ($#argv == 0) then                   [Are there no arguments?]
echo "Numeric arguments required"       [If so, display a message
exit                                    and terminate the program]
if ($#argv > 10) then                   [Are there more than ten arguments?]
echo "Only ten numbers allowed"         [If so, display a message
exit                                    and terminate the program]
endif
set output = (0 0 0 0 0 0 0 0 0 0)      [Initialize output with ten zeroes]
set count = 1                           [Begin the counter at one]
foreach number ($argv)                  [Set number to the next argument]
```

```
    if ($number % 2) continue        [Test the argument for odd or
                                      even]
    set output[$count] = $number     [If it is even, store it in output
    @ count++                         and increase the counter
  end                                 by one]
  echo "Even numbers: $output"       [Display the even numbers
                                      stored]
```

You activate this program by typing **select**, followed by a sequence of no more than ten whole numbers. The first seven lines of statements take care of incorrect invocations of the command (no numbers typed after **select** or too many numbers). Then two variables are initialized (the array that will hold the numbers selected and the counter).

Finally, the main procedure begins the **foreach** statement, which begins by assigning the first number typed to the variable **number**. The **if** statement on the next line is the key to this program. It uses the remainder operator (**%**) to determine whether or not the number is even:

1. If the number is even, the remainder that results from dividing by 2 will be zero (indicating false), the **continue** statement will be ignored, the number will be assigned to **output** on the next line, and the counter will be increased by one for the next iteration. The counter serves as a pointer to the next location in **output**—in which the number is to be stored.

2. If the number is odd, the remainder will be 1 (indicating true), and the shell will execute the **continue** statement, which acts like a **goto** statement, returning execution to the foreach statement. This means that the number will *not* be assigned to **output** and the counter will not be increased.

3. If the argument isn't a number, the shell will display an error message.

Finally, after all arguments have been processed, the resulting value of **output** is displayed. Here is what you will see displayed on the screen:

```
<16> select
Numeric arguments required

<17> select 3 4 5 11 16 17 22 31 46 63
Even numbers: 4 16 22 46 0 0 0 0 0 0
```

## Exiting from a loop **exit**

This statement works exactly like the **exit** statement of the Bourne shell: it forces a program to abort and return to the parent process. You can pass an exit status as an argument of the **exit** statement to the parent process. See Chapter 24, "Bourne Shell Program Control," for a detailed description of this statement.

## 27.4 Other programming techniques

### Multi-way branching **switch**

The **switch** statement, which is similar to the **case** statement of the Bourne shell, provides multi-way branching. End each **switch** statement with **breaksw**, which is analogous to the double semicolon (`;;`) of the Bourne shell. Program control will then break from the **switch** statement and resume at the **endsw** statement. The format of the **switch** statement is as follows:

```
switch ( string)
case pattern1:
      command(s)
breaksw
case pattern2:
      command(s)
breaksw
default:          [optional]
      command(s)
breaksw
endsw
```

The **switch** statement passes **string** to each **case** to match the pattern. After executing the command list of the **case** selected, the **breaksw** statement directs program control to the statement that follows the **endsw** statement. Without the **breaksw** statement for each **case**, the program will continue to execute the next **case**, even though there is no match. If no match is found, the command list following the **default** statement is executed. Here is an example:

```
<18> cat matchfile.1
# This program matches one of two files
#
if ($#argv == 0) then
  echo "No argument is declared"
  exit
endif
switch ($argv[1])
  case FILE1:
     echo " You have selected FILE1"
  breaksw
  case FILE2:
     echo "You have selected FILE2"
  breaksw
  default:
     echo "You must select either FILE1 or FILE2"
  breaksw      [optional]
endsw
```

The **if** statement is to test for the existence of an argument to be passed by the C shell before going to the **switch** structure (optional). If you leave out the **if** statement to test the existence of **argv[1]** as required, the **switch** statement will refuse to go on, and the program will terminate with a vague message from the C shell.

If the above program is invoked with an argument, The **switch** structure will try to match this argument with string FILE1 or FILE2. If there is no match, the **default** is invoked; this procedure includes error-checking to guarantee correct usage. The last **breaksw** is optional. Although it is not required, it is good practice to have it there. If you omit it, program control will go to the **endsw** statement anyway. (The **breaksw** statement will also direct program control to the **endsw** statement.) Here you see **matchfile.1** invoked three different ways, the first two incorrect:

Invoke **matchfile.1** without an argument:

```
<19> matchfile.1
No argument is declared
```

Invoke **matchfile.1** with argument **unknown**

```
<20> matchfile.1 unkown
You must select either FILE1 or FILE2
```

Invoke **matchfile.1** with argument **FILE2**:

```
<21> matchfile.1 FILE2
You have selected FILE2
```

Since the **switch** structure also accepts metacharacters such as *, ?, and so on, as expansion characters, we can change **default** to become another **case** with the statement **case** *—without changing program performance:

```
<22> cat matchfile.2
# This program matches one of two files
#
if ($#argv == 0) then
  echo "No argument is declared"
  exit
endif

switch ($argv[1])
  case FILE1:
     echo " User seclects FILE1"
  breaksw
  case FILE2:
     echo "User selects FILE2"
  breaksw
  case *:        [Change]
```

```
     echo "You must select either FILE1 or FILE2"
endsw
```

## SHIFTING ARGUMENTS **shift**

This statement is the same as the **shift** statement of the Bourne shell. It shifts each element of **argv** to the left one position, and discards **argv1]**. If the value of **argv** does not have at least one word, then the C shell regards **argv** as an unset variable, and displays an error message. Here is an example:

```
<23> cat moveleft
# Shift arguments to the left
foreach words ($argv[*])
  shift
  echo $argv
end
```

Let's invoke **moveleft** with three arguments (1, 2, 3):

```
<24> moveleft 1 2 3
2 3              [Argument 1 is discarded]
3                [Argument 2 is discarded]
<25> _           [Argument 3 is discarded]
```

## INTERRUPT-HANDLING **onintr**

At times you may want to terminate a C shell program while it is running by pressing the (DEL) key. However, you also want to remove all of the temporary files that were created by this program at the time it terminates. You can do so with the **onintr** statement. This is the format of the statement:

```
onintr label
```

When an interrupt occurs, the C shell clears the interrupt and passes control to the **onintr** statement, which executes the line that starts with *label:*—as if you had given a **goto** statement. Here is an example:

```
<25> cat print.format
#
if ($#arv == 0) then            [Does the file exist?]
  echo " Missing input file"
  exit 0
  else if -d $argv[1] then      [Is the file a directory?]
    echo " $argv[1] is a directory"
  exit
```

```
endif
onintr cleanup                  [Interrupt control]
  pr $argv[1] > $home/bin/tempfile
  cp $home/bin/tempfile $arg[1]
cleanup:                        [Interrupt-handling routine]
  if -f $home/bin/tempfile
    rm $home/bin/tempfile
  exit
```

If you press the (DEL) key while this program is running, **onintr** will transfer program flow to the line labeled **cleanup**. This interrupt handling routine checks for the existence of **tempfile** in your **bin** directory. and removes it before exiting.

## 27.5 Built-in commands

### General description

The C shell includes built-in commands, to which the C shell can pass control without using a search path or creating a new process. One of these commands is described below. For a complete list of built-in commands, see Appendix I, "Summary of the C Shell."

### Executing a command

This built-in command executes the command argument in the current shell without returning to the calling program. Here is an example:

```
% cat example
# Check the directory
if -d $argv[1] then
  cd $argv[1]
  exec pwd
  echo 'This line is never displayed'
% _
```

In this program, the last line will never be executed because the **exec** command after executing the **pwd** command will exit to the C shell and never return to program **example**.

## 27.6 Summary

In this chapter you learned about executing a file as a shell procedure; the statements for forming conditional structures (**if, then, else**, and **else if**); the statements for forming loops (**goto, foreach, while, break, continue**, and **exit**); and a few other tools for writing C shell procedures (**switch, shift**, and **onintr**).

## Executing a file as a shell procedure

To make a file executable by the C shell, you must place a pound sign (#) at the beginning of the first line, then use the change mode command with **u+x**. To execute any file as a C shell procedure, use the **csh** command, followed by the filename. The C shell uses expressions to test files and directories in **if, foreach**, and **while** statements the way the Bourne shell uses **test** to performs similar tests.

## Forming conditional statements

You can use the **if** statement by itself to carry out a simple test. For more involved tests, you can use the **if**...**then**...**else** construction. You can use the **else if** statement to add an additional set of alternatives.

## Forming loops

Use the **goto** statement to force the C shell to branch to a labeled line. Use the **foreach** statement in the C shell as you would use the **for** statement in the Bourne shell. Use the **while** statement to form a loop, and sustain execution of the loop as long as the expression being tested indicates true.

Use the **break** statement in the C shell much the way you would use the **break** statement in the Bourne shell. Use the **continue** statement to direct program control to the nearest **while** or **foreach** loop. Use the **exit** statement in the C shell as you would use the **exit** statement in the Bourne shell.

## Other programming techniques

Use the **switch** statement as you would use the **case** statement in the Bourne shell to perform multiple branching, based on pattern-matching, terminating each alternative with **breaksw** (instead of ;;). Use the **shift** statement in the C shell as you would use it in the Bourne shell. Use the **onintr** statement in the C shell to perform clean-up in the event that the procedure is interrupted before completing execution. It will then act like a **goto** statement.

## For further reading

If you'd like additional information about shell programming, refer to the following:

Kochan, Stephen G. and Patrick H. Wood, *UNIX Shell Programming*, Hasbrouck Heights, NJ: Hayden Book Company, 1985

Anderson, Gail and Paul, *The UNIX C Shell*, Englewood Cliffs, NJ: Prentice-Hall, 1986

# Part VI

# System Administration

In Part VI, you will learn a little about what goes on internally in UNIX. You will also learn the commands and procedures for providing disk space for users, checking for errors, formatting disks, performing backup and recovery, starting the system up, shutting the system down, adjusting terminals and printers, taking care of security measures, and monitoring system activity and performance. The exact procedures may vary from one system to another.

# 28

# Basic Information

## 28.1 The system administrator

### Functions

When many people are using a single computer system at the same time, there must be orderly procedures for dealing with administrative details. On larger systems, there is usually a *system administrator* who is responsible for these tasks. However, if there is no administrator for your system (or if you have a smaller system), you may have to take over. The person in charge of system administration always has to take care of these things:

1. *Maintaining file systems*—This includes keeping file systems free of errors.

2. *Taking care of devices*—This includes mounting and unmounting tapes, installing terminals and printers, backing up user files, and (if necessary) reconstructing the system after a failure.

3. *Maintaining disk space*—This includes making sure that users have enough disk space to work with.

4. *Operating the system*—This includes starting the system up, formatting disks, shutting the system down, handling terminals, and so on.

5. *Maintaining security*—This involves protecting the system and individual user files from unauthorized use and possible damage.

Some administrators may also have to assume these additional, more complex tasks:

6. *Maintaining accounting*—This includes monitoring usage by user to aid security, to evaluate performance, or to implement a billing system.

7. *Setting up communication*—This involves connecting communication lines and setting up the software to allow communication between different UNIX systems. (See Part VII.)

## A SPECIAL STATUS

To make it easier for a system administrator to get the job done without having to worry about access permissions, the system administrator is allowed to perform certain critical tasks as a *super-user.* The super-user has unlimited access to everything in the UNIX system, no matter what permissions have been granted or denied to other users.

However, because of this unlimited access, the super-user can also cause enormous damage to a UNIX system. One mistyped command can demolish an entire file system with one press of the RETURN key. For this reason, only one person should be allowed to have super-user status. And even then, this person should log in as an ordinary user most of the time, and assume super-user status only when necessary for system maintenance.

## A SPECIAL PROMPT

There are two ways to log in as a super-user, depending on whether or not you are already logged in as an ordinary user. If you are *not* currently logged into the system, type the following, then the required password:

```
login: root
Password:
# _
```

If you are currently logged into the system, type the following, then the required password:

```
$ su
Password:
# _
```

In either case, you will be presented a special prompt to remind you of your super-user status (#). Many, but not all of the tasks performed by the system administrator require super-user status.

## A SPECIAL DIRECTORY /etc

The directory in which most administrative files, directories, and commands reside is called /etc. Having the commands here keeps them separate from ordinary commands, which reside in /bin and /usr/bin. This reduces the possibility that an ordinary user may damage the system by accident.

In this book, we'll use full pathnames for commands in most instances to avoid confusion about which commands are administrative and which are not. However, if you you'd like to use the shorter names (**fsck** instead of /etc/fsck, **cron** instead of /etc/cron, and so on), just add /etc to the front of your command pathname list.

The procedure for the Bourne shell is to include the following lines in root's shell start-up file **.profile**, which will be executed if you log in as `root`:

```
PATH=/etc:/bin:/usr/bin:$HOME/bin:
export PATH
```

The procedure for the C shell is to include the following line in root's shell start-up file **.login**:

```
set path =(/etc /bin /usr/bin $HOME/bin .)
```

Another way for you to distinguish administrative commands from ordinary commands in this book is to look at the prompt that precedes the command (# means administrative, $ or % means ordinary).

## Communicating with users

Besides using the **mail** command, there are three ways a system administrator can send messages to users:

- Use the message of the day.
- Use a news item.
- Use the **wall** (write-all) command.

The file **/etc/motd** contains the message of the day, which the system administrator can arrange to have displayed on the screen as each user logs in. This file can be used for any announcement the system administrator may want to make to all users. For example, suppose you want each user to see this message at login: "System shutdown at 6:00 pm today (Tuesday)." You can do this by changing to the **/etc** directory, editing the file **motd**, changing the current message of the day to the new one, and storing the file.

To get a brief announcement to users, use the **news** command. To reach users who are currently logged in, use the write-all command (**wall**).

Users can send messages back to the system administrator by directing mail to root, like this:

```
$ mail root < problem
```

## Plan for part VI

It might appear that now is the time to begin describing how to start up the UNIX system. However, that procedure involves using many different administrative commands, which in turn require that you know something about what goes on inside UNIX. Therefore, in keeping with the rest of this book, we've chosen to lay the groundwork one step at a time, which means saving start-up procedures for a later chapter. Starting up the system will be much clearer to you if you first become familiar with the individual subprocedures.

## 28.2 Time-sharing concepts

### MEMORY AND DISKS

A program can only be executed from within a computer's memory. The most efficient way for a computer to operate is to have all the programs that it will ever execute available in memory at all times. However, with UNIX, this is out of the question; there are just too many programs, and memory is too expensive. Of the hundreds of UNIX programs that exist, only about two or three dozen of them will fit into the memory of a computer with today's technology.

The solution is to store the programs on disk, which is much less expensive than memory, then read them into memory when they are needed. This is slower than keeping them in memory, but it allows UNIX to function.

### TIME-SHARING

UNIX is a multi-user, multi-tasking time-sharing system:

- *multi-user*—many users can operate it at the same time.
- *multi-tasking*—each user can execute several commands at the same time.
- *time-sharing*—it accommodates multiple users and tasks by sharing its processing time with all of them, dividing it into small intervals.

By allocating to each process by turns a tiny interval of time, rather than completing one process before starting the next, UNIX can create the illusion that each user has exclusive access to the system. This rotation of processes is an example of *multiplexing* memory space.

As each process begins execution, the system attempts to find space for it in memory. Then to each process, the system allocates a pre-determined time slice. If there is insufficient space left in memory to accommodate a new process, the system can make room for it by copying one of the existing processes to a special disk area called the *swap area.*

For example, suppose various users enter the following commands from their terminals: **date**, **who**, **bc**, **vi**, **nroff**, **sort ps**. Then, at consecutive time intervals, memory allocation may change as shown in Figure 28.1 (shading indicates unused memory).

FIGURE 28.1. Time-sharing with various processes.

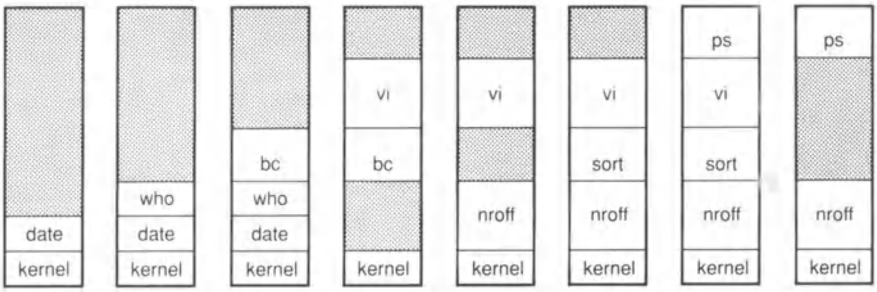


Figure 28.1 shows how various processes occupy memory together, coming and going at various time intervals. It also shows how these processes may overlap one another in time. As each new command is executed, the system finds space for another process in memory, while continuing to rotate processes through its time-sharing scheme.

## Swapping

In most of the eight snapshots of memory shown in Figure 28.1 there is some unused memory. What happens if all memory is occupied (as in the seventh snapshot) and the system receives more processes? It copies processes to and from the *swap area*, an action that is known as *swapping*. Under a swapping scheme, entire processes are swapped back and forth between disk and memory.

Swapping is more efficient than executing one process at a time or executing processes in partitions of fixed sizes. However, it requires a great deal of reading to and writing from disk. This is one reason why UNIX requires high-capacity, high-speed disks to run effectively.

## Paging

Just as swapping is more efficient than earlier methods of disk management, an additional refinement results in an even more efficient method. Instead of swapping entire processes, why not swap only fixed segments of code called *pages*? This refined method of swapping is called *paging* (or *demand paging*). Statistical studies show that the optimal size of a page is one or two kilobytes. A paging diagram would look similar to Figure 28.1, with one difference: only one page of each process would be in memory at any given moment. The remaining pages of the process would reside in the swap area, available upon demand.

### Synchronization

While swapping or paging is taking place, information in various user files becomes altered in memory. However, there is nearly always a certain interval between the time the information is altered and the time the disk file is updated to reflect the alterations. During this interval, there is a discrepancy between the original file and the copy of the file being processed in memory. In other words, the two are not synchronized. If a power failure should halt the system while it is in this state, there will be flaws in the system the next time you start it. You can issue a command, called **sync**, that forces all information in memory to be written to disk. Specific uses for this command are discussed in later chapters.

## 28.3 Disks and file systems

In the chapters that follow, you will learn about disks and file systems in detail. Before getting into these detailed discussions, you may want some background information.

### Formatting disks

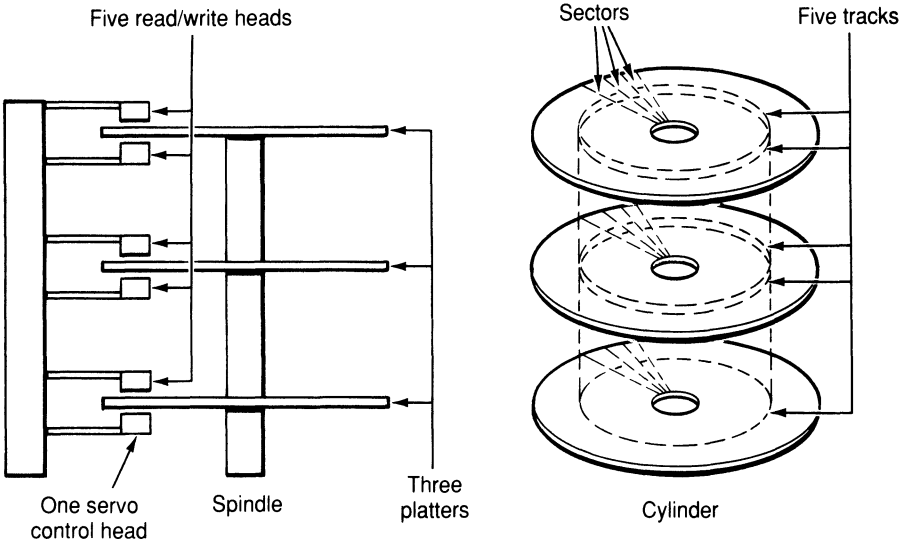
A new disk that comes from the manufacturer is simply a blank magnetic medium, which is unusable in a computer system. To make the disk usable, you have to install the disk and run a program that organizes the blank surface into locations that can be accessed by a program. This process is called *formatting*, and the result is a *formatted* disk.

A formatted diskette contains narrow concentric rings of read/write space called *tracks.* Each track is divided into a fixed number of addressable areas called *sectors.* A program can then locate information on the disk by identifying tracks and sectors by number.

On larger drives with multiple heads and multiple disk platters, each set of tracks that are vertically aligned is called a *cylinder* (named for the geometric shape formed by the tracks). For example, if a drive has six heads facing the six surfaces of three platters, with 100 tracks on each platter, there will be exactly 100 cylinders, each formed from six vertically aligned tracks (see Figure 28.2).

On a three-platter disk drive with six surfaces, one surface would be used for servo control information, requiring only a read head. Since each of the other surfaces would have read/write heads, the drive would have five tracks per cylinder. Since the tracks are vertically aligned, it is possible to change tracks without changing the position of the read/write heads.

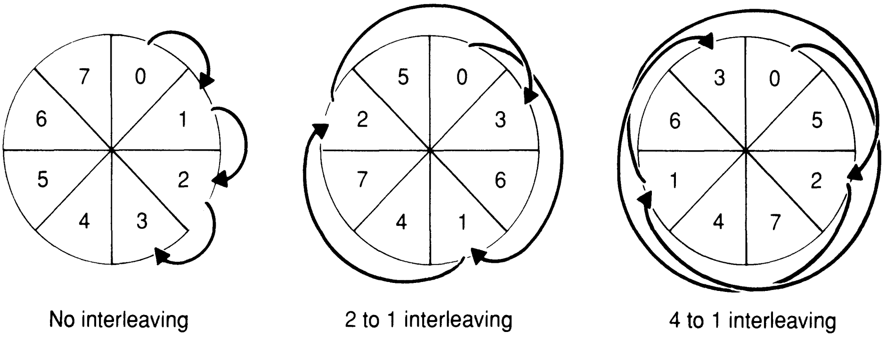
FIGURE 28.2. Cylinders, tracks, and sectors on a disk.



## INTERLEAVING

If the sectors of a cylinder are numbered consecutively, they may be too close together for the read/write head to read or write in a single pass. Extra passes will be required, and this will slow down the entire system. To remedy this, you can offset the numbering, allowing the read/write head to read or write every other sector (or every third sector, or every fourth). This kind of numbering scheme is called *interleaving* (see Figure 28.3).

FIGURE 28.3. Interleaving sectors.



For each disk controller and drive, there is an optimal interleave factor. If you set interleaving too low, disk operations will be seriously degraded; if you set interleaving too high, disk operations will be slowed, but not nearly as drastically. The statistics in the following table, gathered from an actual 30-megabyte disk, illustrate the differences in disk performance for different interleave factors.

| Interleave Factor | Disk Revolutions Required to Transfer a Single Track | Data Transfer Rate (Bytes per Second) | |
|---|---|---|---|
| 1 to 1 | 26 | 30,706 | |
| 2 to 1 | 26 | 30,706 | |
| 3 to 1 | 26 | 30,706 | |
| 4 to 1 | 26 | 30,706 | |
| **5 to 1** | **5** | **159,744** | Optimum |
| 6 to 1 | 6 | 133,120 | |
| 7 to 1 | 7 | 114,088 | |
| 8 to 1 | 8 | 99,840 | |

## Placement of files

Files that are written to a newly formatted disk can usually be placed on the disk fairly neatly in contiguous sectors. However, as files are edited, changed, and rewritten to the disk over time, the arrangement can become very disorderly. It's like filling part of a page with writing, and then going back to add more words to a sentence. There's no room left around the sentence itself, so you write the addition in an open space and draw an arrow pointing from the sentence to the added text.

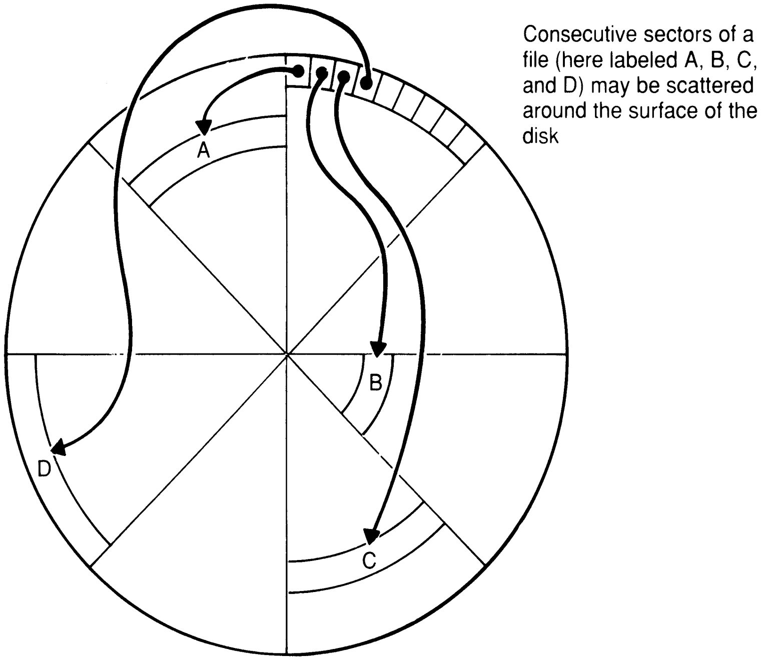
This is what happens on the surface of a disk. As the space begins to fill up and files are enlarged, files become more and more fragmented, with fragments scattered around the disk. What keeps each file intact from the point of view of the user is a set of pointers (like your handwritten arrows) that show where all the fragments are located (see Figure 28.4).

## File systems

The logical structure that keeps track of the location of files and directories (which are also files) is a *file system.* In Chapter 29, "File Systems," you will learn in detail how a file system is organized. For now, let's see how file systems and disks are related in general terms. In the simplest case, one file system may occupy an entire disk. However, it is more likely that each disk will contain two or more file systems.

Each UNIX system is a collection of file systems. However, because these file systems are woven into a seamless whole, there is no need to refer to individual disk drives, as there is in MS-DOS. Each physical device (disk) contains at least one logical device (file system). But once the file system is incorporated into the over-all system (*mounted*), the disk becomes transparent.

FIGURE 28.4. Files and pointers.


Consecutive sectors of a file (here labeled A, B, C, and D) may be scattered around the surface of the disk
A
B
C
D

A disk that contains more than one file system is said to be *partitioned.* For example, suppose a disk with 100 cylinders is to be divided into two equal parts to contain two file systems. Then the first file system will occupy cylinders 0-49 and the second will occupy cylinders 50-99.

## The swap area

Although the swap area is not a file system, it can also occupy either an entire disk or one parition of a disk. This explains why it is sometimes called the *swap disk* (or partition). As of System V, Release 3, the swap area contains pages of programs, rather than entire programs.

## XENIX aids **custom**

XENIX provides a series of menus for assisting the system administrator in preparing disk drives and installing the system. In particular, the XENIX **custom** command allows you to install or remove portions of the system with the aid of prompts. Screen displays tell you how much free space is left on the disk, and how much space is occupied by each feature currently installed.

## 28.4 Summary

In this chapter, you learned about the system administrator's responsibilities and tools, the concept of time-sharing, and some basic facts relating to disks and file systems.

### The system administrator

The system administrator (or the person who is handling system administration) takes care of disk maintenance, system operation, backup and recovery, and security, and possibly system accounting and communication. This person has super-user status, a special prompt (#), and a special directory called /etc.

A system administrator can communicate with users with the message of the day (stored in /etc/motd), a news item (either the **news** command or the /usr/news directory), or the **wall** (write all) command. Users can communicate with the system administrator by sending mail to root.

### Time-sharing

The commands for a computer system are stored on disk, from which they are copied into memory to be executed. In a time-sharing system, no one command is processed to completion in one continuous period of time; instead, many commands are rotated through execution in small time intervals. To make this possible, UNIX reserves a special area of disk space, called the *swap area*, for shuffling command files in and out of memory. The swap area stands apart from the rest of the UNIX system, in that it does not belong to any file system. (In System V, Release 3, *paging* replaces swapping.)

### Disks and file systems

Disks, which are the storage areas for the UNIX system, must be formatted before they can be used. Formatting organizes a disk into cylinders and sectors (with vertically aligned tracks on a multi-disk system forming a cylinder). The operating system can then place files on the disk in addressable locations. Offsetting the numbering of sectors to prevent delays is called interleaving.

A collection of files and directories, along with the information for keeping track of them, is called a *file system.* Each file system can occupy either an entire disk or one segment of a disk called a partition. The swap area likewise can occupy either an entire disk or one partition. (Beginning with System V, Release 3, the swap area contains pages of programs, rather than entire programs.)

## FURTHER READING

To learn about memory management in greater detail, refer to

Tanenbaum, Andrew S., *Operating Systems: Design and Implementation*, Englewood Cliffs, NJ: Prentice-Hall, 1987.

Bach, Maurice J., *The Design of the UNIX Operating System*, Englewood Cliffs, NJ: Prentice-Hall, 1986.

Comer, Douglas, *Operating System Design: the XINU Approach*, Englewood Cliffs, NJ: Prentice-Hall, 1984.

# 29

# File Systems

In Chapter 28, "Basic Information," you learned a few things about the system administrator, time-sharing, file systems, and disks. Now you will learn more about file systems: how they are structured and how to keep them free of errors. You may wish to skip the technical details and proceed directly to "Checking File Systems," page 451.

## 29.1 The structure of a file system

### File systems in general

UNIX includes within itself information about the location of each existing file, as well as information about unused space for new files. The system handles disk allocation for all users, assigning locations automatically, retrieving copies of any files required by users, and returning unused space to a free pool. The information used to manage files, along with the files themselves, constitutes a *file system.* Each UNIX system has at least one file system; most have many files systems. The basic unit of a file system is the *block.* (In earlier versions, each block was 512 *bytes* long; in System V, each block is 1,024 *bytes* long.)

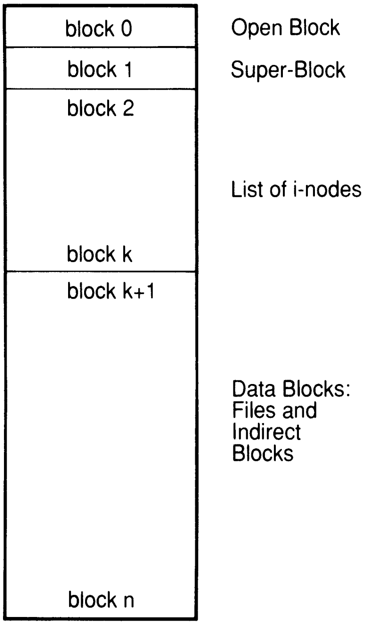
### System V file systems

Each System V file system consists of a *super-block*, which stores information about the file system as a whole and keeps track of records called *i-nodes*; i-nodes, which contain information about files (including where they are located); and actual files themselves. In Figure 29.1 you see a file system with $n$ total blocks, $k - 1$ i-nodes, and $n - k$ data blocks.

The *open block* (block 0) is usually used to hold a bootstrap program, but has no meaning to UNIX, which begins at block 1.

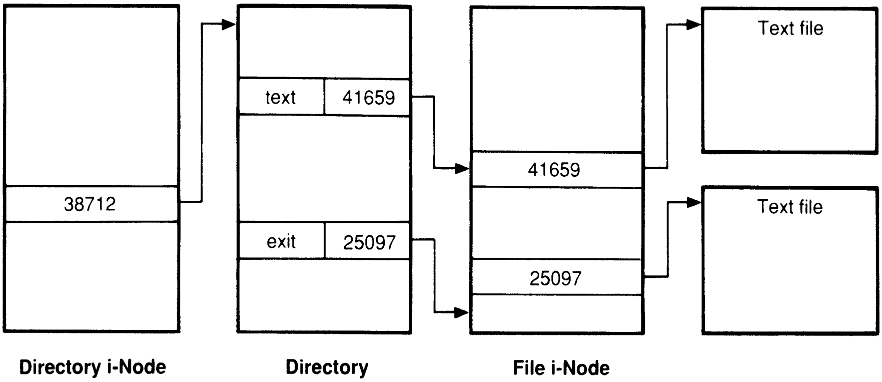
The *super block* (block 1) contains information about the file system as a whole (see Appendix J, "Summary of System Administration," for details). *i-nodes* (blocks 2 through $k$) are records that contain information about directories and files. There is one i-node for each directory and one i-node for each ordinary file. Each i-node contains information about a single file (see Appendix J for details). The relationship between i-nodes, directories, and files is shown in Figure 28.2.

FIGURE 29.1. Structure of a file system.



*Data blocks* (blocks $k + 1$ through $n$) fill the rest of the file system, and can form one of three things: a directory, a file, or an *indirect list* (which is explained below).

FIGURE 29.2. Directories, files, and i-nodes.



## System V i-nodes

As shown in Figure 29.2, one directory i-node provides the location of a directory, which contains the locations of many file i-nodes, each of which points to the location of each block of a particular file. For a small file (up to 10 blocks, or 10,240 bytes), the i-node points directly to every block in the file, using the first 10 of its own 13 addresses (1,024 × 10 = 10,240).

But as soon as a file grows to 10,241 bytes and beyond, a system of *indirect* pointers goes into effect. The eleventh address of an i-node, if needed, points, not directly to a block in the file, but to another data block, which in turn points to the next 256 bytes of the file. This will accommodate up to 266 blocks (or 272,384 bytes).

If a file should outgrow the capacity of the i-node's own pointers and of the indirect block, then the next level of indirection goes into effect. The twelfth address of an i-node points to one block, which then points to 256 blocks, each of which points to one block in the file. This will accommodate up to 65,802 blocks (67,381,248 bytes).

Finally, if triple-indirection is required and the i-node goes to its thirteenth address, an individual file can grow (theoretically) to a maximum size of 16,843,018 blocks (17,247,250,432 bytes). See Figures 29.3 to 29.6. To summarize:

1. The first 10 addresses point directly to the first 10 blocks of the file.
2. The eleventh address (if required) points to a block that contains the addresses of the next 256 blocks of the file (single indirection).
3. The twelfth address (if required) points to as many as 256 blocks, each block pointing to another 256 blocks of the file (double indirection).
4. The thirteenth address (if required) points to as many as 256 blocks, each block pointing to another 256 blocks, with each of them pointing to 256 blocks of the file (triple indirection).

FIGURE 29.3. Direct addressing.

| *Address* | | *Blocks* | |
|---|---|---|---|
| 1 | ⟹ | 1 | |
| 2 | ⟹ | 2 | |
| 3 | ⟹ | 3 | |
| 4 | ⟹ | 4 | 10 blocks addressed (1,024 x 10 bytes) |
| 5 | ⟹ | 5 | |
| 6 | ⟹ | 6 | |
| 7 | ⟹ | 7 | |
| 8 | ⟹ | 8 | |
| 9 | ⟹ | 9 | |
| 10 | ⟹ | 10 | |

## Data blocks

Referring back to Figure 29.1, all data blocks in a file system that follow the i-nodes (blocks $k + 1$ through $n$) can be described as containing one of the following:

FIGURE 29.4. Single indirect addressing.

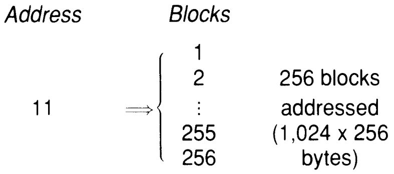


FIGURE 29.5. Double indirect addressing.

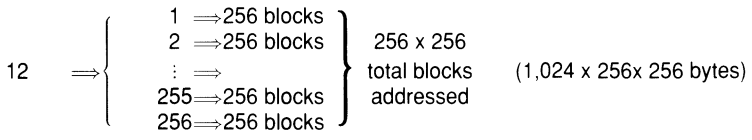
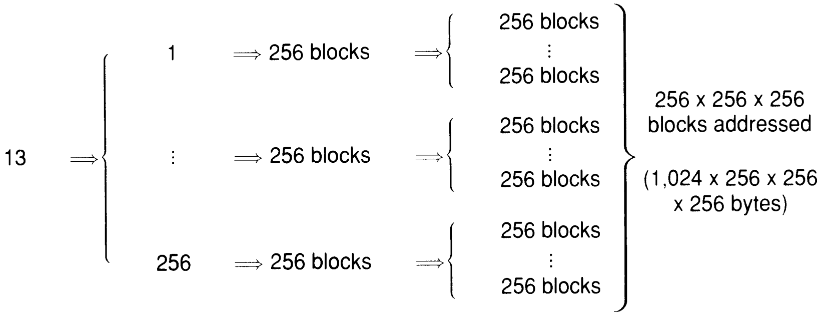


FIGURE 29.6. Triple indirect addressing.



- Files
- Pointers to files (that is, they are indirect blocks)
- Neither files nor pointers to files (that is, they are free)
- Pointers to unused blocks (members of the free-block list)

To summarize, then, most data blocks are either used for files or not used for files. The file system identifies used blocks and maintains a list to keep track of unused data blocks. Blocks used for files, already described, are allocated to particular i-nodes (and possibly some indirect blocks for long files); the list of unused blocks is called the *free list*.

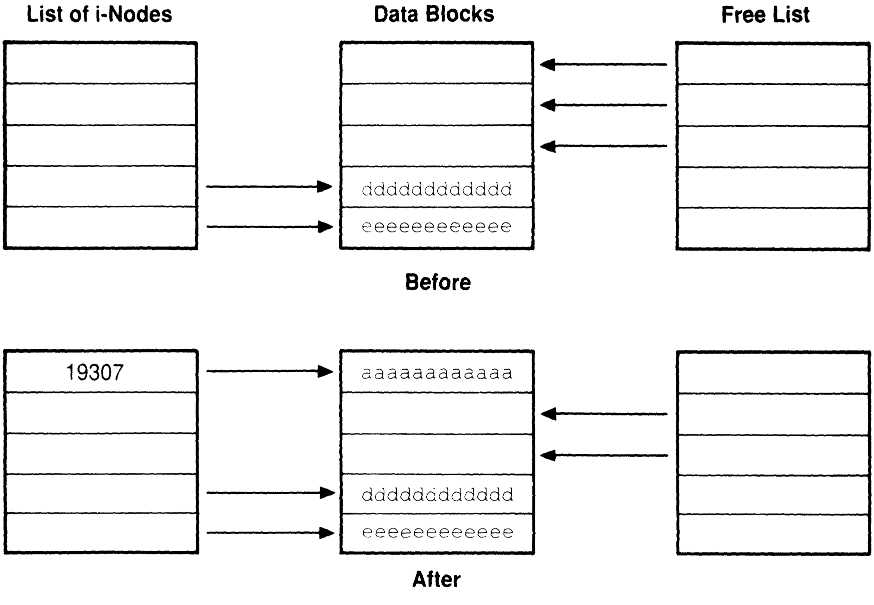
Any time you create a new file, the file system does the following:

1. Sets aside data blocks to hold the file.
2. Adds pointers to those blocks to the "in-use" list (the i-nodes).
3. Removes pointers to those blocks from the free list.

For example, suppose you have just completed a short editing session with **vi** and you type *:w* to write the text to a new file called **letter** in directory **/usr/robin/text**. Assuming that only one data block will be required for the text, here is what happens (see Figure 29.7):

1. The system finds the next available data block in the free list.
2. The system assigns the block to an i-node by creating a directory that contains the filename (**letter**) and i-node number.
3. The system removes the pointer to the block from the free list.

FIGURE 29.7. Writing to a block.

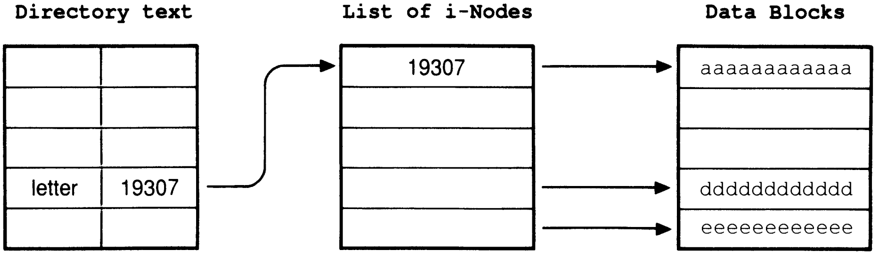


## An example

The next time you request to read file **letter** with **vi**, you set in motion another sequence of events (see Figure 29.8):

1. The system finds the name **letter** in directory **text**, along with the associated i-node number.
2. The system finds the i-node, and reads information about permissions, the starting location, and the length of the file **letter**.
3. With this information, the system can locate the file and read it to the standard output.

FIGURE 29.8. Reading from a block.



To simplify this example, we assumed that you made this request from within directory text. If you had made the request from another directory, then the system would have had to begin by finding the i-node of directory text, which is also a file.

Let's continue this discussion, starting in directory text. Here is a brief sequence of commands, with comments about what happens when each one is executed.

```
$ cp letter ltr.bak
```

The system creates a copy of the text in another block, removes the pointer to the block from the free list, places a new name (ltr.bak) in directory text, and activates a new i-node to point to the block.

```
$ mv letter interview
```

This time the system merely places a new name (interview) in directory text, and links it to the same i-node, without changing the i-node or the text itself.

```
$ cd ..                [Move to directory /usr/robin]
```

The system finds the parent directory and makes it your current directory.

```
$ ln text/ltr.bak LETTER
```

The system places a new name (LETTER) in the parent directory robin, and links it to the same i-node, without changing the i-node or the text itself. There is now one link to this i-node in directory text and one in directory robin.

```
$ rm LETTER
```

The system removes a name (**LETTER**) from the parent directory **robin**, without changing the i-node or the text itself. There is now only one link to this i-node (in directory **text**).

```
$ cd text
```

The system finds directory **text** and makes it your current directory.

```
$ rm ltr.bak
```

The system removes a name (**ltr.bak**) from the directory **text**, thereby removing the last link to this file. The block that contained the file is detached from the i-node and associated with the free list again.

Of course, the file **interview** still exists with the same text, but it has a different i-node.

## 29.2 Checking file systems

Earlier in this chapter, you learned about the structure of a UNIX file system. In this section you will use learn how to keep file systems in order with the aid of a program called **fsck** (file system check). But before discussing **fsck** itself, let's consider why you need this program.

### AN ANALOGY

Maintaining a file system is something like keeping a checking account in balance. In both cases, the key is to make sure that the numbers come out evenly, without any discrepancies. With a checking account, you have a total balance, individual checks written against the balance, and records of the checks written. As long as your records, the bank's records, and the checks themselves are in agreement, everything is fine.

However, any time there is a disagreement, you have to find the error that produced the disagreement and correct it. (This describes very briefly the function of the **fsck** command.) In the case of your checking account, an error can come about if you accidently write one amount on a check and another amount in your record book. An error can arise if the bank clerk transcribes the amount (or your account number) incorrectly.

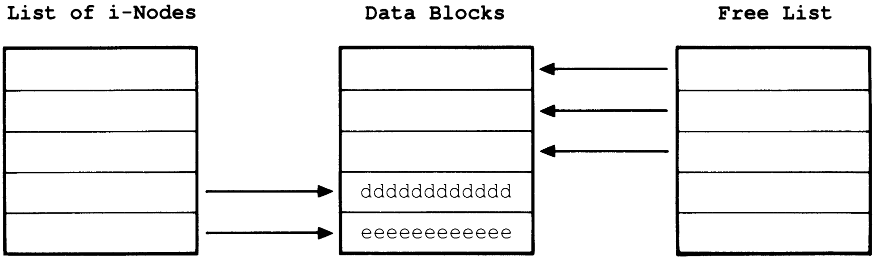
But no matter what causes the error, the result will be a discrepancy between your actual balance and the balance shown in your records. If the error goes undetected, you run the risk of overdrawing your account.

## ORDER IN A FILE SYSTEM

In a file system, you have a "balance" (the total number of blocks), from which you "draw" any time you allocate some of those blocks to a file. In our diagrams here we'll show the blocks next to each other in a row. This may happen on a disk also, but it is more likely that the blocks in which a file is stored will be scattered all over the disk—like the papers on your desk.

As long as the file system's records remain accurate, this is no problem. When you call up the file for editing, the file system locates all the blocks, wherever they may be, and presents them to you very neatly on your screen. You are never aware how chaotic the arrangement may be on your disk. Once again, to review from the previous section, the file system's records consist of one list of pointers to blocks allocated to your files (i-nodes and indirect blocks for large files) and another list of pointers to blocks *not* allocated to your files (free-block list, or free list). Figure 29.9 shows part of a file system where everything is in order.

FIGURE 29.9. An errorless file system.



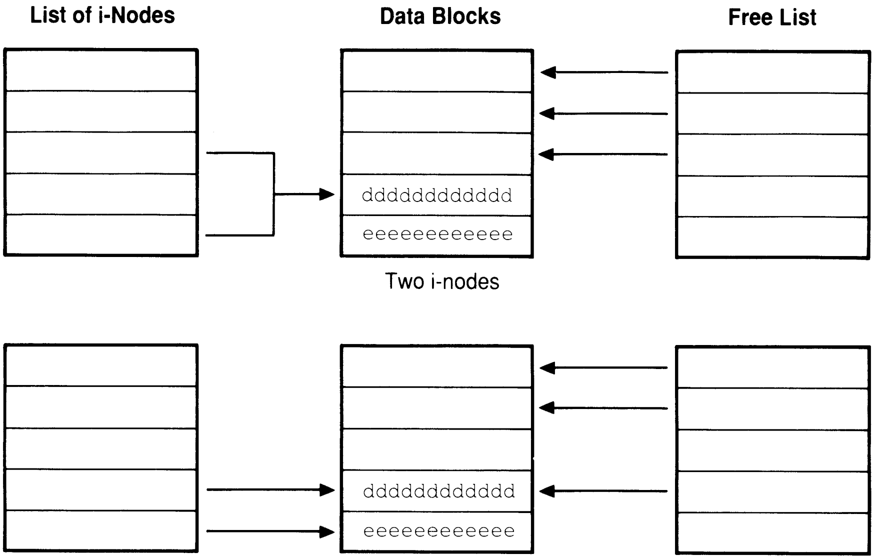
Everything is in order in Figure 29.9 because each i-node points (with a number) to a different block and each member of the free list points (with a number) to a different block. Each block is accounted for as either allocated to a file or unallocated (free), with no conflicts or discrepancies. (On disk, unallocated blocks are not blank; this is just a convenient way of representing free blocks symbolically.)

## ERRORS IN A FILE SYSTEM

There are various ways errors can arise in a file system. An electrical power surge or momentary loss of power can produce errors. You can introduce errors into a file system by shutting down or starting up UNIX incorrectly. A file system that contains errors is said to be *corrupted*, or *damaged.* Correcting errors is referred to as *cleaning*, or *repairing*, the file system.

Because of the way files are interrelated in a file system's structure, errors that are undetected tend to spread and multiply, resulting in the loss of more and more files until the entire file system becomes completely unusable. Yet some of the original errors appear quite harmless, as you can see in Figure 29.10.

FIGURE 29.10. File systems with duplicate blocks.



One i-node and one member of the free list

In the upper half of Figure 29.10, two different i-nodes are both pointing to the same block (that is, both contain the same i-number). This causes two problems: 1) a block that appears to be in two different files at the same time and 2) another block that is now unaccounted for. A possible remedy could be to discard the block.

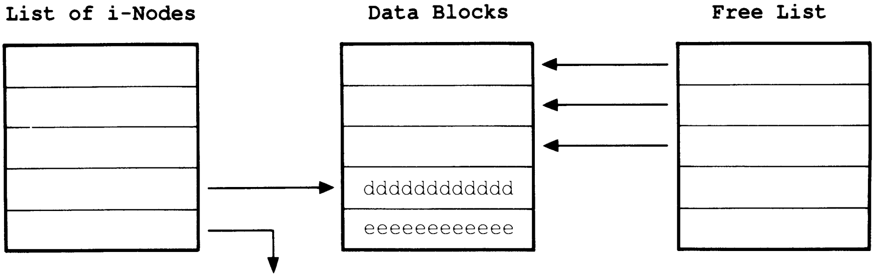
In the lower half of Figure 29.10, a block is the subject of the attention of both an i-node and a member of the free list at the same time. This makes the block appear to the file system to be both allocated and unallocated at the same time.

It's easy to see how this can cause problems. On the next write to disk, the system may view this block as unallocated and allocate it to another file; then on a subsequent read, the system may view the same block as part of the original file. The result will be foreign data in the file being read. Again, a possible remedy could be to discard the block.

Another kind of error arises when the block-addressing scheme gets disrupted, as shown in Figure 29.11. Here, the lower i-node is supposed to be pointing to the lower block, but the block address is incorrect. Any time an i-node's block address has become altered, engineers have a technical term, chosen after careful consideration, to describe the block: *bad.*

Other errors include discrepancies between the number of blocks allocated to a file and the number found in the i-node, discrepancies between

FIGURE 29.11. A file system with a bad block.



the number of actual links to a file and the number shown in the i-node, discrepancies between the actual number of i-nodes and the number shown in the super-block, and dead-end i-nodes that do not point to blocks even though they are referred to in directories (known as *unallocated i-nodes*).

Naturally, the closer an error is to the top of the file system, the greater its potential for doing harm. An error in the root directory is much more dangerous than an error far down in a user's subdirectories.

## FINDING NO ERRORS **fsck**

In Chapter 32, "Startup and Shutdown," you will learn how to start up a UNIX system, which involves making a transition from single-user mode to multi-user mode and mounting file systems. The **fsck** program should be executed in *single-user mode* on *unmounted* file systems. If the program encounters no errors, it runs in five phases, as shown here. If you don't name a file system, **fsck** checks the file systems named in the default list /etc/checklist.

```
# /etc/fsck
/dev/dsk
File System: usr  Volume: d01
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Free List
596 files  3987 blocks  12,43 free
# _
```

## FINDING ERRORS **fsck**

If the program encounters errors during phase 1, it proceeds to a phase 1B; if it finds errors during phase 5, it proceeds to a phase 6, not phase 5B. (No one ever said that UNIX was noted for consistency.) Any time there is an error, **fsck** asks you if you want it corrected (**y** or **n**):

**y** You want **fsck** to attempt to correct the error (and run the risk of losing data).

**n** You want **fsck** to continue checking for additional errors (without correcting this one).

In addition, the command may have two options (**-y** and **-n**) to provide for automatic yes or no to all questions about correcting errors. A safe way to proceed with **fsck** is to use the **-n** on the first run to find out the extent of any damage to the file system, then respond with **y** on subsequent runs.

`** Phase 1 - Check Blocks and Sizes`

Check each i-node to determine whether duplicate or bad blocks exist and to locate possible discrepancies in the sizes of files and directories. If there is at least one duplicate, then proceed to Phase 1B.

`** Phase 1B - Rescan for More Dups`

Go back and find out if there are more duplicates.

`** Phase 2 - Check Pathnames`

Check each directory that points to one of the faulty i-nodes, if any, encountered in Phase 1.

`** Phase 3 - Check Connectivity`

Check for unreferenced directories (parent's i-node doesn't exist). If you reply **y** to correction here, **fsck** will place any directory (or directories) named in the **/lost+found** directory for later retrieval.

`** Phase 4 - Check Reference Counts`

Check the link counts from Phases 2 and 3, and report on any unreferenced files or directories, incorrect link counts, duplicate and bad blocks, and incorrect i-node counts.

`** Phase 5 - Check Free List`

Check the free-block list, and report any duplicate or bad blocks in the list, incorrect totals, and unused blocks not shown in the list. If **fsck** finds any errors, proceed to Phase 6.

`** Phase 6 - Salvage Free List`

Correct any errors encountered during Phase 5.

`Conclusion` At the end of a run, **fsck** will display the first of the following messages, and possibly one of the other two in addition:

```
nnnn files   bbbbbb blocks   ff free
***** FILE SYSTEM WAS MODIFIED *****
***** BOOT UNIX (NO SYNC!) *****
```

If you see the third message, boot UNIX as explained in Chapter 32, "Startup and Shutdown," but *without* running the **sync** program (which is ordinarily mandatory). If the error occurs in root, **fsck** may reboot the system automatically (check the manual for your system).

## Saving files

Suppose you've just run **fsck** with the **-n** (automatic no) option, with the prompt shown here appearing in one of the messages:

```
# /etc/fsck -n
   . . .
** Phase 4 - Check Reference Counts
UNREF FILE  I=2157   OWNER=robin  MODE=000755
SIZE=1286  MTIME=Jan 23 09:47 1987
CLEAR? n
```

After **fsck** has completed its run, it may be possible to save the file, provided that the damage is not too great. Here is the procedure:

1. Find out the name of the file (**fsck** doesn't use names, only numbers):

   □ Identify the file by naming the i-node and file system in an **ncheck** command:

   ```
   # /etc/ncheck -i 2157 /dev/dsk/c0d0s3
   exam
   # _
   ```

   □ The file is called exam. (If you execute **ncheck** without options, it will list all files in all file systems, along with their associated i-numbers.)

2. Find out if the text is still intelligible:

   □ Move to the directory and examine the contents of the file:

   ```
   # cd /usr/robin/admin
   # cat exam | more
   Sometimes we need to examine what
   we are accomplishing to determine
                  .
                  .
                  .
   # _
   ```

   □ The text appears to be intact.

3. Copy the file to a safe place:

□ Copying to another directory on a different file system would be the easiest alternative:

```
# cp exam /tmp/haven
# _
```

□ Archiving to tape might be the surest method (see Chapter 30, "Disks and Tapes"):

```
# find . -name exam -exec cpio /dev/rmt/mt1
# _
```

4. With the file safely copied, clear the damaged i-node:

□ Start **fsck** again (without the **-n** option):

```
# /etc/fsck
```

□ Answer yes to the "CLEAR" prompt this time:

```
UNREF FILE  I=2157   OWNER=robin  MODE=000755
SIZE=1286  MTIME=Jan 23 09:47 1987
CLEAR? y
   . . .
***** FILE SYSTEM WAS MODIFIED *****
```

## Saving a directory

Saving a directory is a little easier, since **fsck** makes the backup copy for you. Suppose once again that you've started **fsck** and that the prompt shown below appears during checking:

```
# /etc/fsck
   . . .
** Phase 4 - Check Reference Counts
UNREF DIR  I=4323  OWNER=robin  MODE=000751
SIZE=3888  MTIME=Mar 30 15:08 1987
RECONNECT: y
```

1. After **fsck** has run, locate the unreferenced directory:

□ Display the contents of **/lost+found**:

```
# cd /lost+found
# ls -li
   . . .
4323  2 drwxr-x--x  11 robin enter 3888 Dec 16 09:03 004323
   . . .
# _
```

- □ Your directory **memos** shows up there.

2. Recover your directory:

```
# mv 004323 /usr/robin/admin/memos
# cd
# _
```

## 29.3 Summary

In this chapter you learned about file systems: how they are structured and how to keep them free of errors.

### The structure of a file system

A System V file system consists of a *super-block* (block 1), which contains information about the file system as a whole; *i-nodes* (blocks 2 through $k$), which contain information about files (including directories); and *data blocks* (blocks $k - 1$ through $n$), which are blocks for directories, files, indirect lists, and the free list.

### Maintaining the integrity of the file system

To keep the file system free of errors, you have the **fsck** program to check a file system for errors, with the option of making corrections on the spot. By merely perusing through errors on the first pass, backing up the files that are affected, then asking for corrections on the following pass, you can salvage files and directories that have been damaged.

### Further reading

To learn more about the internal workings of UNIX, refer to

Tanenbaum, Andrew S., *Operating Systems: Design and Implementation*, Englewood Cliffs, NJ: Prentice-Hall, 1987.

Bach, Maurice J., *The Design of the UNIX Operating System*, Englewood Cliffs, NJ: Prentice-Hall, 1986.

Comer, Douglas, *Operating System Design: the XINU Approach*, Englewood Cliffs, NJ: Prentice-Hall, 1984.

# 30

# Disks and Tapes

In the previous chapter, you learned about the structure and care of file systems. In this chapter you learn about disks and tapes, which may contain file systems.

## 30.1 Devices and file types

Until now we have discussed only ordinary files and directory files (or directories). But UNIX also allows three other file types, known as *special files.* Before describing these, let's discuss devices, which will provide a little background.

### UNIX DEVICES

UNIX, like other operating systems, is designed to handle processing for various peripheral devices, such as disk drives, magnetic tape drives, terminals, printers, and modems. As a piece of hardware, each of these is represented within UNIX as a *file*, but disks and tapes may also contain *file systems*, which will be explained later in this chapter.

Each device is described by its type (*major device number*), as well as by a sequence number (*minor device number*). For example terminal number 7 (*tty07*) might have a major device numer of 4 (local terminal) and a minor device number of 7. Line printer number 3 might have a major device number of 6 (line printer) and a minor device number of 3. The major device numbers for an installation might be assigned like this (this varies widely from one installation to another):

| | | | |
|---|---|---|---|
| 0 | Hard disk (block device) | 5 | Outside terminal |
| 1 | Magnetic tape (block device) | 6 | Line printer |
| 2 | [Not used] | 7 | Character printer |
| 3 | [Not used] | 8 | Raw disk (character device) |
| 4 | Local terminal | 9 | Raw tape (character device) |

The mass storage devices (disk and tape drives) process data in blocks, and so are called *block devices.* The devices intended for human interaction (terminals, printers, and modems) process one character at a time, and so are called *character devices.* The file that represents a device must correspond in type to the device itself, as described in the next section.

## Device special files

UNIX has a directory called /dev that is reserved for the files, known as *special files*, that represent hardware devices. In System V, Release 1 and earlier versions of UNIX, all device special files reside in /dev proper; however, in System V, Release 2 /dev has four subdirectories set aside for mass storage devices:

- /dev/dsk—for hard disk drives to be treated as block devices.
- /dev/rdsk—for hard disk drives to be treated as character devices.
- /dev/mt—for magnetic tape drives to be treated as block devices.
- /dev/rmt—for magnetic tape drives to be treated as character devices.

The r in the two directory names stands for *raw* device, a device that performs direct input and output without grouping data in blocks. The absence of r in the other two names indicates block device, a device that sends and receives data only in blocks. Therefore, it is necessary to have two entries in /dev for each disk and tape: one as a block device and one as a character device (or raw device).

## File types

As you have seen, UNIX views devices connected to the system as files (actually, *special files*). Two of the special file types are associated with devices: block special files and character special files.

There is also a third special file type that is associated with *fifo* (first-in, first-out) files (also known as *named pipes*). Briefly, these are unrestricted pipes that are used in system programming.

To summarize, then, there are five file types allowed in UNIX:

- Ordinary files (text, data, and programs)
- Directory files (names of other files)
- Block special files (mass storage devices)
- Character special files (asynchronous devices)
- fifo special files (named pipes)

File types b (block special) and c (character special) are found only in directory /dev.

## UPDATE ON LISTING A DIRECTORY **ls**

Now that you've learned a little about the structure of file systems and all the file types allowed in UNIX, this may be a good time to discuss some features of the **ls** command that were not mentioned in Chapter 3, "The UNIX File System." First of all, a long listing (**ls -l**) always includes a one-character file type (as well as permissions, the number of links, size in bytes, and time of last modification). Here are the characters used to identify file types:

| | |
|---|---|
| - | ordinary files |
| d | directory files |
| b | block special files |
| c | character special files |
| p | fifo special files |

There are additional options that you can bundle with the **-l** option to obtain information useful in system administration:

**-a** Show all entries (including those that begin with a period)
**-i** Display i-numbers
**-s** Display sizes in blocks (including indirect blocks)

Here are some examples of **ls** command lines that include these options, either by themselves or in combinations:

1. Display all files in the current directory:

□ Bundle **-l** (long) with **-a** (all):

```
$ ls -la
total 7
drwxr-xr-x   6 robin   users    112 Jan  7 23:03 .
drwxr-xr-x  18 bin     bin      288 Dec 16 15:54 ..
-rw-r--r--   1 robin   users    336 Dec 15 18:52 .profile
drsxr-xr-x   2 robin   users     48 Dec 15 19:27 admin
drwxr-xr-x   2 robin   users    128 Jan  8 23:01 c.progs
drwxr-xr-x   2 robin   users     64 Jan  8 22:45 test
drwxr-xr-x   2 robin   users     48 Jan  7 23:06 text
$ _
```

□ Included here are the directory's link to itself (.), its link to its parent directory (..), and initialization file for the shell (**.profile**). These appear only when you use **-a**.

2. Include i-numbers in the directory display:

□ Bundle **-l** (long) with **-i** (i-numbers):

```
$ ls -li
total 4
 2398 drwxr-xr-x   2 robin   users    48 Dec 15 19:27 admin
 2399 drwxr-xr-x   2 robin   users   128 Jan  8 23:01 c.progs
 2400 drwxr-xr-x   2 robin   users    64 Jan  8 22:45 test
 2401 drwxr-xr-x   2 robin   users    48 Jan  7 23:06 text
$ _
```

□ Each line now begins with an i-number, which is associated with the file name on the right.

3. Include block sizes in the directory display:

□ Bundle **-l** (long) with **-s** (sizes in blocks):

```
$ ls -als
total 4
   1 drwxr-xr-x   2 robin   users    48 Dec 15 19:27 admin
   1 drwxr-xr-x   2 robin   users   128 Jan  8 23:01 c.progs
   1 drwxr-xr-x   2 robin   users    64 Jan  8 22:45 test
   1 drwxr-xr-x   2 robin   users    48 Jan  7 23:06 text
$ _
```

4. Display all files with block sizes for root:

□ Bundle **-l** (long) with **-a** (all) and **-s** (block size):

```
$ ls -als /
total 779
   1 drwxr-xr-x  12 root   root    400 Jan  2 09:42 .
   1 drwxr-xr-x  12 root   root    400 Jan  2 10:36 ..
   6 drwxr-xr-x   2 bin    bin    2592 Oct 18 15:18 bin
   4 drwxr-xr-x   5 root   root   1824 Nov 27 19:53 dev
   4 drwxr-xr-x   3 root   root   1840 Jan  7 05:29 etc
   1 drwxr-xr-x   4 bin    bin     304 Jan  3 17:00 lib
   1 drwxr-xr-x   2 bin    bin      32 May 15  1985 lost+found
   1 drwxr-xr-x   2 root   root     32 May 15  1985 mnt
   1 drwxrwxrwx   4 root   root     64 Jan  7 23:05 tmp
   6 drwxr-xr-x   2 bin    bin    2656 Oct 16 10:56 usr
$ _
```

□ This time all files are shown, and block sizes are included.

5. Display the contents of the **/dev** directory:

□ This will require only the **-l** option:

```
$ ls -l /dev
total 10
crw--w--w-   2 root   design   0,  0 Feb  8 12:00 console
crw--w--w-   2 root   design   0,  0 Feb  8 12:00 syscon
brw-------   1 root   root     0,  0 Jan 12 19:34 dsk
brw-rw-rw-   1 root   root     1,  0 Jun 23  1986 mt
cr--r-----   1 sys    sys      3,  0 Aug 23  1986 mem
cr--r-----   1 sys    sys      3,  1 Apr  2  1986 kmem
crw--w----   2 lp     daemon   6,  0 Jan 11 15:21 lp
crw--w--w-   1 root   design   5,  0 Mar 22 10:42 tty00
crw--w--w-   1 root   design   5,  1 Apr 11 11:21 tty01
crw--w--w-   1 root   root     5,  2 Oct 19 12:37 tty02
crw--w--w-   1 root   design   5,  3 Nov 20 13:24 tty03
crw-------   1 root   root     8,  0 Sep  5  1986 rdsk
crw-rw-rw-   1 root   root     9,  0 Apr 24 15:43 rmt
crw-rw-r--   1 root   root    10,  0 Dec 23  1986 error
crw-------   1 root   root    16,  0 Mar 17  1986 init
```

□ Major and minor device numbers replace the number of bytes.

Here are some other **ls** options of interest to system administrators:

| | |
|---|---|
| **-n** | Same as **-l**, but replace file name with owner and group IDs |
| **-t** | Sort by time of last modification of the file |
| **-tc** | Sort by time of last modification of the i-node |
| **-lc** | List by time of last modification of the i-node |
| **-tu** | Sort by time of last access |
| **-lu** | List by time of last access |

## 30.2 Adding and removing devices

Adding a device to the UNIX system begins with a command that simply identifies the device. For disks (and occasionally for tapes), the next step is to execute a series of commands that will add a UNIX file system.

### Creating a device file **mknod**

The first step in adding any device to UNIX is to create a device file in /dev that will identify the device. The command for doing this, called **mknod** (make node), requires a filename, a file type, and major and minor device numbers. (Remember, major and minor device numbers vary from system to system.) Here are a few examples:

```
# /etc/mknod /dev/tty08 c 4 8
# _
```

Create a file for terminal number 8, defining it as a character device.

```
# /etc/mknod /dev/mt/0 b 1 0
# /etc/mknod /dev/rmt/0 c 9 0
# _
```

Create two files for magnetic tape drive 0, one that defines it as a block device and one that defines it as a character device.

```
# /etc/mknod /dev/dsk/c0d0s6 b 0 6
# /etc/mknod /dev/rdsk/c0d0s6 c 8 6
# _
```

Create two files for a hard disk drive designated as controller 0, disk 0, slice (partition) 6: one that defines it as a block device and one that defines it as a character device.

For terminals, printers, and modems, this is all that is required at this point (see Chapters 33, "Terminals" and 34, "Printers," for information about setting features). But for disk and tape drives, there are further optional steps you can also take.

## Formatting the disk

The next step in adding a disk drive to the UNIX system is to format the disk, using a special utility program. This will mark the disk's sectors in a prearranged format for reading and writing. The name of the program and the procedure for running it will vary from one installation to another. Formatting isn't usually necessary for tape drives, since they store information sequentially.

## Creating a file system **mkfs**

After formatting, the next step is to create a file system, using the **mkfs** command. In its ordinary form, the **mkfs** command requires the name of the device and the number of blocks desired. The name of the device is the name used for the **mknod** command. By default, the system will assign a number of i-nodes equal to 25% of the number of blocks. Here are two examples:

```
# /etc/mkfs /dev/rmt/0 16000
# _
```

Create a file system for magnetic tape drive 0 with 16,000 blocks (4,000 i-nodes implied).

```
# /etc/mkfs /dev/dsk/c0d0s6 40000
# _
```

Create a file system for the disk drive designated controller 0, disk 0, slice (partition) 6 with 40,000 blocks (10,000 i-nodes implied).

With four blocks per i-node, this means an average of 4 kbytes (4 blocks) per file. If you expect to use either a larger number of small files or a smaller number of large files, you can provide for this by appending another number to represent the desired number of i-nodes, using a colon as a separator. Here are two examples:

```
# /etc/mkfs /dev/rmt0 16000:12000
# _
```

Create a file system for magnetic tape drive 0 with 16,000 blocks and 12,000 i-nodes (three times as many smaller files allowed).

```
# /etc/mkfs /dev/dsk/c0d0s6 40000:2000
# _
```

Create a file system for the disk drive with 40,000 blocks and 2,000 i-nodes (one-fifth as many larger files allowed).

The maximum number of blocks allowed for a file system will vary from one installation to another.

## Creating a label **labelit**

To create a label for ease of identification, use the **labelit** command to write one to a newly created file system. If you choose to execute this command, you will have to provide the name of the file system, a six-character file system identifier, and a six-character volume identifier. Here are two examples:

```
# /etc/labelit /dev/rmt/0 tape00 vol003
# _
```

Create a label for magnetic tape drive 0 with **tape00** and **vol003** as the file system and volume identifiers.

```
# /etc/labelit /dev/dsk/c0d0s6 disk01 vol002
# _
```

Create a label for the hard disk drive designated controller 0, disk 0, slice 6 with **disk01** and **vol002** as the file system and volume identifiers.

To display identifiers already created, execute the **labelit** command, followed by the name of the file system, without identifiers.

## Mounting a file system **mount**

The final step in adding a disk or tape drive to the UNIX system is to mount it with the **mount** command, which allows you to select the location of this file system. (The name of the file system will be added to a list in a file called /etc/mounttab.) If a device is write-protected, you must use the **-r** (read only) option at the end of the command line; otherwise, errors will result. Here are two examples:

```
# mkdir /usr/tape00
# /etc/mount /dev/rmt/0 /usr/tape00 -r
# _
```

Mount magnetic tape drive 0 as /usr/tape00 (write-protected).

```
# mkdir /usr/disk12
# /etc/mount /dev/dsk/c0d0s6 /usr/disk12
# _
```

Mount hard the disk drive as /usr/disk12.

To display file systems already mounted, execute **mount** alone. To mount a file system on another UNIX system using the Remote File Sharing feature of Release 3, use the **-d** option, as described in Chapter 40, "Introduction to Resource Sharing."

### UNMOUNTING A FILE SYSTEM **umount**

To undo what you have done with the **mount** command, you can use the **umount** (unmount) command (spelled u-m-o-u-n-t without an n). As long as the file system is inactive, you can execute **umount** to make the file system inaccessible to users. Here are two examples, one to unmount a magnetic tape drive and one to unmount a hard disk drive:

```
# /etc/umount /dev/rmt/0
# _
```

```
# /etc/umount /dev/dsk/c0d1s2
# _
```

You can also unmount a file system on another UNIX system using the Remote File Sharing feature of Release 3. See Chapter 40 for details.

**Caution:** To avoid damage to a file system, always connect the device *before* mounting it and disconnect the device *after* unmounting it. Here is the correct sequence:

connect

mount

use

unmount

disconnect

### THE SWAP AREA */dev/swap*

The swap area, described in Chapter 28, "Basic Information," is not a file system, and therefore cannot be mounted or unmounted. However, it is often identified in the UNIX system as **/dev/swap**.

## 30.3 Backup and recovery

Backing up files is even more important with UNIX than it is with a single-user operating system. If something goes wrong, then a backup copy of the system can mean returning to work in a few hours, instead of spending a week trying to reconstruct a damaged system. We'll be discussing two programs in this chapter: **tar** (tape archive) and **cpio** (copy I/O). The XENIX command **sysadmin**, not discussed here, permits system backup via screen menus.

## THE TAPE ARCHIVE PROGRAM **tar**

Use the **tar** program to copy files or directories to tape, diskette, or some other storage device. To invoke **tar**, insert the tape cartridge (or diskette), and type a command like this:

```
$ tar key[options] [file(s)]
```

The key allows you to select one of five functions; the options allows you to modify the basic function selected. The five functions that you can select by key are shown in Table 30.1. Note that **tar** is one of the few UNIX commands that uses arguments without leading hyphens. In each description, *tape* means *tape (or diskette)*.

TABLE 30.1. The Functions of **tar**

| Key | Function |
|---|---|
| **c** | *Create.*—Create a new tape and write the files named to this new tape, overwriting any existing files. |
| **r** | *Write.*—Write the files named to the end of an existing tape, leaving any existing files in place. |
| **u** | *Update.*—Add to an existing tape any files that either do not already exist or have been since modified. |
| **t** | *Table of contents.*—Display the name of each file named as it occurs on tape; if no files are named, display the names of all files on the tape. |
| **x** | *Extract.*—Read from an existing tape each file named; if no files are named, read all files on the tape. |

## OPTIONS

The **tar** command has the following options:

**f** *File*—Use the name that follows as the name of the archive in place of the system's default; use a hyphen instead of a name to indicate the standard input.

**b** *Block*—Use the number that follows as a blocking factor (1-20) for writing to tape on a raw device (default: 1); not required for reading from tape; inappropriate for updating a tape or for creating a disk file.

**v** *Verbose*—Display the name of each file being copied.

**l** *Links*—Display messages if **tar** is unable to figure out all the links to the files being copied.

**m** *Modification time*—Replace the modification time in the i-node with the current time.

**w** *Confirm*—Ask for confirmation (**y**) before copying.

**0-7** *Drive*—Override the default drive number with the number given here (0–7).

## Using tape

Tape is used to back up very large amounts of data. Although it may be used in multi-user mode, it is best to access tape in single-user mode. This is because reading or writing tape will tend to have priority over almost all other activity on the system.

If you are using a streaming tape cartridge, there are a few other points to note. When you first insert a cartridge into the tape drive, the tape may begin retensioning (streaming to the end of the reel and then back to the beginning again). If so, wait until this is completed before trying to access the tape with a UNIX command.

## Backing up a file

To back up a file (or group of files) to tape (or other medium), overwriting anything already on the medium, move to the directory that contains the files and use **tar** with the **c** key:

```
# cd directory                  [Move to the desired directory]
# tar cvf /dev/mt/3 file(s)     [Copy the file(s) named to /dev/mt/3
                                 in verbose mode                    ]
```

## Backing up a directory

To back up an entire directory to tape (or other medium), overwriting anything already on the medium, move to the desired directory and use **tar** with **c** and the name of the directory:

```
# cd /usr/robin/admin           [Move to the desired directory]
# tar cvf /dev/mt/3 .           [Copy all files in the directory to
                                 /dev/mt/3                          ]
```

Note that we use **.** to name the current directory; “***** would name all the visible files in the current directory, but would miss dot files such as **.login**.

## Restoring a file

To restore a file (or group of files) from tape (or other medium), move to the target directory and use **tar** with the **x** key:

```
# cd /usr/robin/admin           [Move to the desired directory]
# tar xvf /dev/mt/3  file(s)    [Copy the file(s) named from
                                 /dev/mt/3 in verbose mode
```

If the file(s) already exist in the directory, it (or they) will be overwritten by the file(s) extracted from tape.

### Restoring a directory

To restore an entire directory from tape (or other medium), move to the desired directory and use **tar** with the **x** option:

```
# cd /usr/robin/admin   [Move to the desired directory]
# tar xvf /dev/mt/3     [Copy all files on the tape to the directory]
```

## The copy I/O program **cpio**

Like **tar**, the **cpio** program also has keys to determine which function to perform, along with options to modify the basic function. The functions that you can select are summarized in Table 30.2.

TABLE 30.2. The Functions of **cpio**

| Key | Function |
|---|---|
| **-o** | *Output*—Concatenate the files named in the standard input, add a header, and copy the single resulting file to the standard output. |
| **-i** | *Input*—Using patterns entered on the command line, extract from the standard input, which must be the output of a previous **cpio -o** command, any files matched. |
| **-p** | *Pass*—Copy the ordinary files named in the standard input to the directory named on the command line. |

### Options

The **cpio** command has the following options, which are bundled with the key on the command line: All Keys

| | |
|---|---|
| **c Compatible** | Write the header in ASCII code so that any ASCII-compatible machine can read it. |
| **r Rename** | Prompt for a new name for each file before copying: if you type a name, use it; if you just press (RETURN), don't copy the file. |

**t Table of contents** Display the names of the files, but don't copy them.

**v Verbose** Display the name of each file while copying it; **vt** provides additional information.

**u Unconditional** Overwrite newer files with older versions with the same names.

**a Access time** Reset the access time of each file copied.

**m Modification time** Retain the original modification time of each ordinary file copied.

Restricted to Certain Keys

**B Block** Write to a raw magnetic tape device in blocks of 5,120 bytes (**-i** and **-o** keys only).

**d Directory** Create directories when needed while copying ordinary files (**-i** and **-p** keys only).

**l Link** Link files instead of copying them (**-p** key only).

**f Reverse** Copy all files that do *not* match the patterns given (**-i** key only).

## COPYING FILES OUT **-o**

The general form of the **cpio** command for copying out (writing to disk or tape) is as follows:

```
$ cpio -o[crtvuamBdf]
```

The most likely candidates to supply input to this command are the commands that produce lists of filenames, such as **ls** and **find** (which is discussed in detail in the next chapter). For example, the following sequence could be used to copy all the files in directory /usr/robin/admin/sched to tape /dev/rmt/3, with blocking:

```
$ pwd
/usr/robin/admin/sched
$ ls | cpio -oB > /dev/rmt/3
5 blocks
$ _
```

The command just used writes only ordinary files, excluding directories. If you want directories included, you can use the **find** command (described in detail in Chapter 31, "Disk Maintenance"). In the following example, we copy all the files in directory /usr/robin/admin to tape /dev/rmt/3, with blocking:

```
$ cd /
$ find /usr/robin/admin -print | cpio -oB > /dev/rmt/3
```

```
12 blocks
$ _
```

The following sequence performs a function similar to that performed by the previous command line, but with pathnames relative to /usr/robin/admin:

```
$ pwd
/usr/robin/admin
$ find . -cpio /dev/rmt/3
$ _
```

The following command line is equivalent to the first in this subsection, except that it also requests a list of files being copied:

```
$ cd /
$ find /usr/robin/admin -print | cpio -ovB > /dev/rmt/3
/usr/robin/admin/sched/part.a
/usr/robin/admin/sched/part.b
/usr/robin/admin/sched
/usr/robin/admin/memos/3-15.87
/usr/robin/admin/memos/4-21.87
/usr/robin/admin/memos
12 blocks
$ _
```

## Copying files in -i

The general form of the **cpio** command for copying in (reading from disk or tape) is as follows:

```
$ cpio -i[crtvuamB] patterns
```

This form of the command is used to read files that have previously been written to tape or disk with **cpio -o**, described above. For example, the following sequence could be used to copy all the files from the tape mounted as /dev/rmt/3 back to /usr/robin/admin/sched, with blocking retained:

```
$ pwd
/usr/robin/admin/sched
$ cpio -iB < /dev/rmt/3
$ _
```

The following sequence is similar, except that it reads files from disk, instead of from tape. Since blocking is used only when writing to tape, the **B** option has been omitted this time.

```
$ pwd
/usr/robin/admin/sched
$ cpio -i < /dev/rdsk/c0d1s2
$ _
```

### Passing files **-p**

The general form of the **cpio** command for passing files (writing files to a directory) is as follows:

```
$ cpio -p[crtvuamdl] directory
```

This form of the command is similar to **cpio -o**. The difference is that this form copies the files into a directory, rather than to a backup medium. For example, the following sequence could be used to copy all the files in directory /usr/robin/admin/sched to directory /usr/paul/reports:

```
$ pwd
/usr/robin/admin/sched
$ ls | cpio -p /usr/paul/reports
5 blocks
$ _
```

In the following example, we copy all the files in directory /usr/robin/admin to directory /usr/paul/reports:

```
$ pwd
/usr/robin/admin
$ find . | cpio -p /usr/paul/reports
$ _
```

## 30.4 Summary

In this chapter you learned about devices and file types, additional features of the **ls** command for listing a directory, making a device file, mounting and unmounting file systems, and backup and recovery.

### Devices and file types

Peripheral equipment that can be attached to the computer system on which UNIX is running includes mass storage devices that process data in blocks (hard disk and magnetic tape drives) and asynchronous devices that process data one character at a time (terminals, printers, and modems).

The files that represent these devices within the UNIX system (called special files) must correspond in type to their respective devices (block or character).

Special files are stored in the device directory /dev, where they are identified by generic type (major device number) and sequence (minor device number). In System V, Release 2, /dev reserves four subdirectories for the special files that represent disk drives (/dev/dsk and /dev/rdsk) and tape drives (/dev/mt and /dev/rmt).

The UNIX command for listing a directory (**ls**) has options that allow you to display hidden files that begin with periods, i-numbers, and block sizes, along with owner and group IDs (UID and GID) and various sorting schemes.

## Making a device file

To identify a hardware device, use the **mknod** command, giving the file name and type, along with the major and minor device numbers.

## Mounting and unmounting file systems

To add devices to, and remove them from, the UNIX system, (that is, to mount and unmount file systems), you have the following programs available to you:

| | |
|---|---|
| **mkfs** | Make a file system. |
| **labelit** | Label a file system. |
| **mount** | Add a file system to UNIX and designate a specific location for it within the UNIX hierarchy. |
| **umount** | Remove a file system from UNIX. |

## Backup and recovery

To copy files and directories to and from tape or disk, you can use one of the following commands:

| | |
|---|---|
| **tar** | Copy to and from tape or disk, with a variety of options. |
| **cpio** | Copy to and from tape or disk, with the added options of file selection and copying into a directory. |

# 31

# Disk Maintenance

In Chapter 29 you learned about file systems: how they are structured and how to keep them free of errors. Then in Chapter 30 you learned how to work with disks and tapes. In this chapter you how to use a number of programs and procedures that relate to disk maintenance.

## 31.1 Providing disk space

As long as there are plenty of free blocks available, users won't run out of space on the disk. The commands for determining disk space on a file system are shown in Table 31.1, with descriptions following.

TABLE 31.1. Commands for Checking Disk Space

| Command | Description |
|---|---|
| **df** | *Disk Free*—Display the number of free blocks available. |
| **du** | *Disk Usage*—Display the number of blocks used by each file. |
| **find** | *Find a File*—Find a file and take some action. |

### Free or allocated blocks **df**

The **df** (disk free) command can be executed by any user. When entered without an option, **df** displays the number of free blocks and the number of free i-nodes available in the file system specified. If no file system is named, **df** displays the free space in all mounted file systems:

```
$ df
/dev/rmt/mt1   (/tmp    ):   11320 blocks   1574 i-nodes
/dev/rdsk/hd2  (/usr/src):   32904 blocks   4127 i-nodes
/dev/rdsk/hd1  (/usr    ):  112654 blocks   1825 i-nodex
$ _
```

Use **df** with the **-f** option to display the number of free blocks only (omitting the number of free i-nodes).

```
$ df -i
/dev/rmt/mt1   (/tmp    ):   11320 blocks
/dev/rdsk/hd2  (/usr/src):   32904 blocks
/dev/rdsk/hd1  (/usr    ):  112654 blocks
$ _
```

Use **df** with the **-t** option to display the total number of free blocks and i-nodes, along with the number of allocated blocks and i-nodes.

```
$ df -t
/dev/rmt/mt1   (/tmp    ):   11320 blocks   1574 i-nodes
                    total:   19158 blocks   2367 i-nodes
/dev/rdsk/hd2  (/usr/src):   32904 blocks   4127 i-nodes
                    total:   48070 blocks   5239 i-nodes
/dev/rdsk/hd1  (/usr    ):  112654 blocks   1825 i-nodex
                    total:  144534 blocks   6848 i-nodes
$ _
```

## Displaying disk usage **du**

The **du** (disk usage) command, which can be executed by any user, provides information about the amount of disk space used in directories and subdirectories. The command has three options, as shown here:

```
       ⎧-s⎫            Summary report—total number of blocks used
$ du   ⎨-a⎬  name(s)   All report—blocks used by ordinary files also
       ⎩-r⎭            Error report—troublesome files
```

When used without an option, **du** gives the number of blocks that are used by each directory beginning with the current directory (or the directory named), including each subdirectory. In the following example, the last line gives a cumulative total for the entire directory:

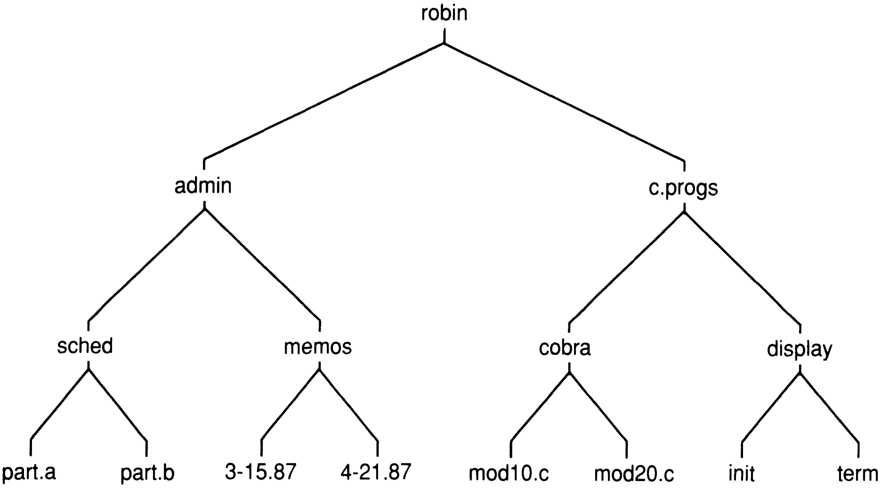
```
$ du /usr/robin
6       /usr/robin/c.progs/cobra
8       /usr/robin/c.progs/display
16      /usr/robin/c.progs
6       /usr/robin/admin/sched
6       /usr/robin/admin/memos
14      /usr/robin/admin
34      /usr/robin          [The -s option displays this line only]
$ _
```

If you would like to see an expanded display, with a line for each individual file, include the **-a** option (see Figure 31.1). If you omit the directory's name, **du** will use the current directory.

```
$ du -a /usr/robin
2       /usr/robin/.cshrc
```

```
2       /usr/robin/c.progs/cobra/mod10.c
2       /usr/robin/c.progs/cobra/mod20.c
6       /usr/robin/c.progs/cobra
2       /usr/robin/c.progs/display/init
2       /usr/robin/c.progs/display/term
8       /usr/robin/c.progs/display
16      /usr/robin/c.progs
2       /usr/robin/admin/sched/part.a
2       /usr/robin/admin/sched/part.b
6       /usr/robin/admin/sched
2       /usr/robin/admin/memos/3-15.87
2       /usr/robin/admin/memos/4-21.87
6       /usr/robin/admin/memos
14      /usr/robin/admin
34      /usr/robin
$ _
```

FIGURE 31.1. Robin's home directory.



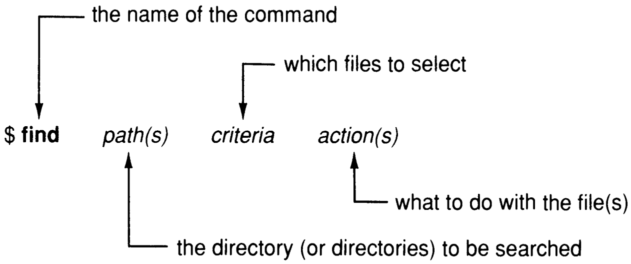
To find out about directories and files that cannot be accessed, include the **-r** (error report) option. There will be no block totals, but the system will report on troublesome files.

```
$ du -r
Unable to open ./text/message
$ _
```

## FINDING FILES **find**

The **find** command, which is available to all users, but is especially valuable to the system administrator, is similar to **awk**. You could say that **find** allows you to do with entire files about what **awk** allows you to do with lines in a file. Even the syntax is similar, in that both commands specify criteria for finding a match, followed by statements for taking action. One difference, however, is that pathnames come second on a **find** command line, whereas file names come last on an **awk** command line. A general command line for **find** is shown in Figure 31.2 (compare with Figure 14.1, page 207).

FIGURE 31.2. A general **find** command line.



Here's a very simple example: Find and display the full pathnames of all files in directory **/usr** that are called *secret*. We'll use the **-name** option to search on the name and the **-print** option to obtain a display, as shown here:

```
$ find  /usr  -name secret  -print
/usr/ann/secret
/usr/bill/secret
/usr/jean/secret
$ _
```

## Criteria options

The *string-valued* options available to form search criteria on a **find** command line are as follows:

| | |
|---|---|
| **-name** *file* | Find file(s) named *file* |
| **-type** { **f**<br>**d**<br>**b**<br>**c**<br>**p** } | Find file(s) of type { ordinary file<br>directory<br>block device<br>character device<br>named pipe } |
| **-user** *name* | Find file(s) owned by user *name* |
| **-group** *name* | Find file(s) that belong to group *name* |
| **-perm** *pppp* | Find file(s) with octal permission code *pppp* (see Chapter 34, "System Security") |
| **-newer** *file* | Find file(s) modified more recently than file *file* |

Here is a ridiculous, contrived example of a command line that uses all of these options:

```
$ find / -name '*.c' -type f -user kate -group pro \
-perm 0644 -newer /usr/robin/admin/tech -print
/usr/robin/c.progs/table.c
/usr/robin/c.progs/metric.c
/usr/robin/c.progs/convert.c
$ _
```

Find in any directory in the system and display any source file(s) for any C program(s) that belong to user **kate**, in group **pro**, with permission 0644 (rw-r-r-), modified more recently than file **/usr/robin/admin/tech**.

The *numeric* options available to form search criteria on a **find** command line are as follows:

$$\textbf{-size}\left\{\begin{array}{l}\textbf{+b}\\ \textbf{b}\\ \textbf{-b}\end{array}\right\}\qquad \text{Find file(s) with}\left\{\begin{array}{l}\text{more than } b \text{ blocks}\\ \text{exactly } b \text{ blocks}\\ \text{fewer than } b \text{ blocks}\end{array}\right\}$$

$$\textbf{-links}\left\{\begin{array}{l}\textbf{+l}\\ \textbf{l}\\ \textbf{-l}\end{array}\right\}\qquad \text{Find file(s) with}\left\{\begin{array}{l}\text{more than } l \text{ links}\\ \text{exactly } l \text{ links}\\ \text{fewer than } l \text{ links}\end{array}\right\}$$

$$\left\{\begin{array}{l}\textbf{-ctime}\\ \textbf{-mtime}\\ \textbf{-atime}\end{array}\right\}\left\{\begin{array}{l}\textbf{+d}\\ \textbf{d}\\ \textbf{-d}\end{array}\right\}\qquad \text{Find file(s)}\left\{\begin{array}{l}\text{changed}\\ \text{modified}\\ \text{accessed}\end{array}\right\}\left\{\begin{array}{l}\text{more than } d \text{ days ago}\\ \text{exactly } d \text{ days ago}\\ \text{fewer than } d \text{ days ago}\end{array}\right\}$$

Here is an example of a command line that uses all of these options:

```
$ find /tmp -size +25 -links 3 -mtime -4 -print
/tmp/special/enter
/tmp/display/format
/tmp/team/schedule
$ _
```

Find in directory **/tmp** and display any file(s) with more than 25 blocks, exactly three links, last modified fewer than than four days ago.

Search criteria can be logically combined and grouped in much the same way search patterns for **grep** or **awk** can be. Here are the conventions:

| | |
|---|---|
| **Grouping:** | **(***option-1 option-2 option-3*\**)** (1 and 2 before 3) |
| **AND:** | *option-1 option-2 option-3* (as in the examples above) |
| **OR:** | **(***option-1* **-o** *option-2*\**)** *option-3* (1 or 2) |
| **NOT:** | **(***option-1 option-2*\**)** **!** *option-3* (not 3) |
| | **!** **(***option-1 option-2 option-3*\**)** (not 1 and 2) |

Here is an example that illustrates the use of logical grouping:

```
$ find . \! \( -user bill -o -newer /admin/enter \) \
> -ctime +5 -print
/usr/paul/misc/trial
$ _
```

Find in the current directory and display any file(s) that meet these criteria: they neither belong to *bill* nor have been modified more recently than **/admin/enter** and they were changed more than five days ago.

## Action statements

There are five action statements you can use in a **find** command line:

**-print** Display the pathname of each file matched.

**-exec** *command* Execute *command* on each file matched unconditionally.

**-ok** *command* Execute *command* on each file matched (with confirmation).

**-depth** Used before **-cpio**, copy the entries of a directory, then the directory itself—useful if you don't have read permission for the directory

**-cpio** *device* Copy to *device* each file matched, using **cpio** format.

The **-exec** and **-ok** options are identical, except that **-ok** asks for a confirmation (**y** for yes) before proceeding to execute the UNIX command. With either option, if the UNIX command ordinarily requires a filename for an argument, then you must type a pair of empty braces, followed by an escaped semicolon, after the name of the command. The **find** command will use the empty braces as a place-holder for the name(s) of the file(s) to be processed.

## Examples of using **find**

As you can see, you can use the **find** command as both a way of displaying information and also a tool for taking action. In the examples that follow we'll illustrate both functions of **find**, starting with a procedure for files that are undesirable. If you've used UNIX for any length of time, you've seen messages like the following appear from time to time:

```
Core dumped
```

Then, if you looked around your directory, you found a large file called **core**. For someone who is debugging the system, this can be a useful tool (a copy of a process in memory at the moment when an error occurred); but for most users, it's just a large file taking up valuable disk space (and often going unnoticed). As system administrator, it is your task to carry out a search and destroy mission against **core** files:

1. Find the locations of the **core** files:

   □ First use **-print** to display all of them:

   ```
   $ find / -name core -print
   /usr/ann/c.progs/core
   /usr/bill/text/core
   /usr/bill/devel/core
   /usr/carol/misc/core
   /usr/dan/C.progs/core
   $ _
   ```

   □ Chances are that you'll find quite a few of them. At this point, you may want to talk to the users about the files before proceeding to eliminate them.

2. Get rid of the **core** files:

   □ Now use **rm** to remove those that have lingered around for more than five days:

   ```
   $ find / -name core -atime +5 -ok rm {} \;
   <rm ... /usr/ann/c.progs/core> ? y
   <rm ... /usr/bill/text/core> ? y
   <rm ... /usr/dan/C.progs/core> ? y
   $ _
   ```

   □ We used **-ok** to allow confirmation, but the best way is to use **-exec** and execute this automatically with **cron** (see later in this chapter).

3. Look for files containing source code for C programs that have been idle for some time:

   □ We'll start again by using **-print** to locate files that haven't been modified for ninety days, and send the list to the printer in the background (quoting *.c to allow the shell to pass it to **find**):

   ```
   $ find /usr -name '*.c' -mtime +90 -print | lp &
   Printing request R-1257
   $ _
   ```

   □ The list will probably be longer than you imagined.

4. Archive these C files to tape:

□ Use a command line similar to the one used above, with **-exec cpio** in place of **-print**:

```
$ find /usr -name '*.c' -mtime +90 \
> -exec cpio /dev/rmt/mt1
$ _
```

□ Now all the files in your printed list have been archived to tape. You may want to give copies of the list to all the users affected.

## 31.2 Programs that run automatically

To allow you to perform certain tasks at regular fixed intervals, UNIX offers a type of program, called a *daemon process* (or *daemon*), that runs by itself automatically in the background. All you have to do is start the process once, and it takes over without any further human interaction.

### UPDATING THE DISK **update**

This process updates the disk every thirty seconds; all you have to do is execute the following command line once to get it started:

```
# /etc/update &
# _
```

### WRITING YOUR OWN SCRIPT **cron**

Another command, called **cron**, allows you to design your own set of periodic tasks and specify the exact times for having them carried out. Using **cron** is fairly similar to programming a VCR (video cassette recorder) to record television programs at different times of the day.

Briefly, the procedure is to enter lines into a file called **/usr/lib/crontab**. (As of System V, Release 2.0, there is a directory called **/usr/lib/cron**, where an ordinary user may store an individual file—provided that the user is named in another file called **/usr/lib/cron/cron.allow** On some systems, this directory may be called **/usr/spool/cron**).

Each line entry, which handles one task, contains five fields for designating the time (minute, hour, day of month, month, and day of week), along with a sixth field for entering the UNIX commands needed to perform the task. Use the wild card character (*) in a time field to indicate all (every minute, every hour, every day, or every month).

The values allowed for the five time fields are shown here. Hours are based on a 24-hour clock (1:00 A.M. is 0100, 1:00 P.M. is 1300); the day-of-week field uses a numbering scheme (0 = Sunday, 1 = Monday, ..., 6 = Saturday); all other fields are self-explanatory.

| Minute | Hour | Day of Month | Month | Day of Week |
|---|---|---|---|---|
| 0-59 | 0-23 | 1-31 | 1-12 | 0-6 |

You can use either blank spaces or tabs to separate fields, with the number-sign character (#) for any comments. Here's an example of a line entry, which displays a message on your screen (terminal 7) every weekday morning at 8:15:

```
15 8 * * 1-5  echo "Good morning, Robin\!" > /dev/tty07
```

Once you've set up **/usr/lib/crontab** and started **cron**, **cron** then reads a copy of the file every minute of the day, seven days a week, and performs any tasks requested for that minute. Here's the command line for starting **cron**:

```
# /etc/cron
# _
```

Here's a brief example, with a few of the things that you'd probably want to include. As you can see from the times, this is a convenient way to schedule tasks for the wee hours of the morning when the system is idle.

```
$ cat /usr/lib/crontab
#mm hh day mo dwk                          Command
#
 0  2   *   *   *    find /tmp /usr/tmp -atime +3 -exec rm {} \;
 5  2   *   *   2,4  find / -name core -atime +5 -exec rm {} \;
10  2   *   *   3    find / -type d -size +5 -print | lp
 0  8   1   *   *    echo "Monthly report" | mail robin
$ _
```

Explanations:

```
0  2   *  *   *    find /tmp /usr/tmp -atime +3 -exec rm {} \;
```

Every night at 2:00 A.M., remove all temporary files that haven't been accessed for at least three days.

```
5  2   *  *   2,4  find / -name core -atime +5 -exec rm {} \;
```

Every Tuesday and Thursday night at 2:05 A.M., remove from all directories all **core** files that haven't been accessed for at least five days.

```
10  2   *  *   3    find / -type d -size +5 -print | lp
```

Every Wednesday night at 2:10 A.M., print a list of all directories that occupy more than 5 kbytes of space.

```
0  8   1  *   *    echo "Monthly report" | mail robin
```

On the first day of every month at 8:00 A.M., mail yourself a reminder to submit a monthly report.

### Starting the processes automatically

In our descriptions of **update** and **cron** above, we used command lines to start the processes manually. However, you can also start them automatically if you prefer. To do this, simply include them in the UNIX initialization file /etc/rc. Any command lines found in this file are automatically executed whenever you start up UNIX.

## 31.3 System efficiency

If you have to partition drives into separate file systems, partition each disk identically. This allows you to replace one disk with another if you have replaceable disks.

Try to keep the number of users evenly distributed among the various drives in your system. If disk usage becomes unbalanced, move user directories from one disk to another to even out the usage.

Use the name of each user's login directory (given by the shell variable **HOME**), rather than rely heavily on pathnames. If a working group has an elaborate file system structure, set aside a shell variable to make it easy to refer to the name of the file system.

### Moving directories

If you have to move user directories to rearrange the file system, use **find** and **cpio**. The following sequence of commands will move, via magnetic tape, directories john and marsha from file system skimpy to file system generous:

1. Save the old files:

   □ Move to directory skimpy:

   ```
   # cd /skimpy
   # _
   ```

   □ Write john and marsha to tape:

   ```
   # find john marsha -cpio /dev/rmt0
   # _
   ```

2. Set up the new directories:

□ Move to directory **generous**:

```
# cd /generous
# _
```

□ Create new directories:

```
# mkdir john marsha
# _
```

□ Change the ownerships:

```
# chown john john
# chown marsha marsha
# _
```

3. Save the old files:

```
# cpio -idmB </dev/rmt/0
# _
```

4. Remove the original files:

□ Return to **skimpy**:

```
# cd /skimpy
# _
```

□ Remove the old directories:

```
# rm -rf john marsha
# _
```

To make sure the new directories are all right, change the login directories for **john** and **marsha** in the **/etc/passwd** file. Whenever you move users in this way, be sure to keep users with common interests in the same file system (in case they have linked files). Also, be sure to move groups of users with linked files with a single **cpio** command; otherwise, you will unlink and duplicate the linked files.

## Keeping directory files small

Directories larger than 10 kbytes are inefficient because of the indirection required. You can use the following command to locate oversized directories. (Better yet, have **cron** run it automatically.)

```
# find / -type d -size +10 -print
# _
```

Merely removing files from a directory will not make the directory itself smaller, since UNIX files never contract naturally. However, you can use the following sequence of commands to compact a directory (/usr/mail):

1. Backup the old directory:

   □ Copy temporarily to /usr/tmail:

   ```
   # mv /usr/mail /usr/tmail
   # _
   ```

   □ Re-create a new /usr/mail:

   ```
   # mkdir /usr/mail
   # _
   ```

   □ Grant all permissions:

   ```
   # chmod a+rwx /usr/mail
   # _
   ```

2. Set up the new directory:

   □ Move to the temporary directory:

   ```
   # cd /usr/tmail
   # _
   ```

   □ Copy the old to the new:

   ```
   # find . -print | cpio -plm ../mail
   ```

3. Remove the temporary directory:

   □ Move to the parent directory:

   ```
   # cd ..
   # _
   ```

   □ Remove the temporary directory

   ```
   # rm -rf tmail
   # _
   ```

### Files and directories that grow

The following files and directories accumulate information constantly. They must be watched, and the excess clutter thrown out regularly. You can have the **cron** program do this automatically if you like.

| | |
|---|---|
| /usr/adm/wtmp | Login information |
| /usr/adm/pacct | Process accounting |
| /usr/lib/spellhist | Misspellings found by **spell** |
| /usr/lib/cronlog | Log of commands executed by **cron** |
| /usr/spool | Spooling directories |
| /usr/news | News items |
| /usr/mail | Messages |
| /usr/tmp, /tmp | Temporary files |

## 31.4 Summary

In this chapter you learned about providing disk space, using daemon processes, and contributing to system efficiency.

### Providing disk space

To make sure that users have enough disk space to work with, you have the following programs available to you:

| | |
|---|---|
| **df** | Display the number of free blocks available. |
| **du** | Display the number of blocks used by each file. |
| **find** | Find a file by size, date, owner, or date of last access. |

### Using daemon processes

Daemons are processes that run automatically in the background. The **update** process updates the disk every thirty seconds. The **cron** process executes commands placed in a file called /usr/lib/crontab. You can either start a daemon manually or place it in /etc/rc to have it started automatically whenever the system goes multi-user (explained in Chapter 32, "Startup and Shutdown").

### Contributing to system efficiency

A system administrator can help to make the system run efficiently by moving directories from one file system to another, keeping directory files from growing too large, monitoring files and directories that have a tendency to grow daily, and running time-consuming processes after hours.

# 32

# Startup and Shutdown

This chapter discusses the procedures for starting up a UNIX system and shutting it down, drawing on what you've learned in the previous four chapters.

## 32.1 Starting up a UNIX system

### DISCLAIMER

Throughout this book we've tried to explain how commands work. But UNIX systems differ from from one site to another, and we can never be sure what will work at any given site and what will not. This is even more so when it comes to procedures for starting up a UNIX system, which depend on the design of the hardware. In this chapter we present a typical procedure and hope that it comes fairly close to resembling the procedure at your installation.

### TURNING ON THE POWER

The first step in starting up your UNIX system is to turn on the power switch. It may also be necessary to turn on your terminals and printers separately. At many installations, the equipment is left on twenty-four hours a day, seven days a week. If so, you will have to turn on the power only the first time you use the system. Turning on the power may activate a set of diagnostic programs to determine whether your system's hardware is functioning properly. In any event, your system is now ready for the next step, which is called *booting.*

### BOOTING THE SYSTEM

Booting the system is the act of copying the kernel from disk, where it is stored, to the computer's memory, where it can operate. The term booting (or bootstrap loading)—not unique to UNIX—comes from the expression, "pulling yourself up by the bootstraps." One small segment of code is first loaded into memory; then this segment calls in a larger module, which performs the actual transfer of data.

Some systems boot themselves automatically when you turn on the power; others do not. If yours does not, then you will have to go to the *console* and do something manually (press (RETURN), type *boot*, or something else).

The *console* is the terminal that serves as the primary device for interacting with the system. In earlier version of UNIX, the console was always the terminal that was next to, or on top of, the computer. But System V allows you to use any terminal as the console (called the *virtual console*, **/dev/syscon**).

After you boot the system (or it boots itself), you may see a prompt (or several prompts) asking you to press other keys or provide other information. Press whatever keys and provide whatever information the system asks for. At this point, the system is running in *single-user mode*, which means that `root` (/) is the only file system known to UNIX. Always perform system maintenance functions in single-user mode (also known as *maintenance mode*).

## Setting the clock

Once booting has been accomplished, the system will ask you to enter the date, which you must type in a standard format. If it is now 9:24 A.M. on April 16, 1987, you can enter the date and time as shown below. (If the year hasn't changed since the last boot, you can omit it from the command line.)

```
# date 0416092487
Mon Apr 16 09:24:01 PST 1987
# _
```

Once you've started the clock, you can display the date and time either in the standard format (**date** alone) or in a format of your own choosing. To format the date and time, begin your format with a plus sign (+) and enclose it within single quotation marks, like this:

```
# date '+Today is %w, %h %d, 19%y. %tIt's now %r'
Today is Mon, Apr 16, 1987.   It's now 09:25:42 AM
# _
```

For complete information about formatting, refer to `date(1)` in the *UNIX User's Manual.* Records for clock-settings, as well as for all logins, are kept in a System V login file called **/etc/wtmp**. (For XENIX and earlier versions of UNIX, the file is called **/usr/adm/wtmp**.)

## Checking the File Systems **fsck**

The last thing you need to do before leaving single-user mode is to check your file systems with the **fsck** (file system check) command. If you have not included **fsck** in your initialization script (described below), here is the command command line, with the output that you will see if no errors are found. (This program was described in some detail in Chapter 29, "File Systems").

```
# /etc/fsck
** Phase 1 - Check Blocks
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Free List
596 files  3987 blocks  1243 free
# _
```

## Logging in

After you enter the date and check the file systems, type the following to enter *multi-user mode*:

```
# init 2
```

The system may now prompt you to press (CTRL-D) to enter multi-user mode and log in. If you have special tasks to perform, you could log in as root at this point (left); otherwise, simply log in as an ordinary user (right):

```
login: root                                   login: robin
Password:                                     Password:
# _                                          $ _
```

## Initializating the system

Any time the system changes states, such as when it enters multi-user mode, the shell may run an initialization script that is stored in a file called /etc/rc. (This depends on what the initialization command calls for—see "The Initialization Command (Release 3.0)," page 496.) You can place in this file any shell commands you wish to have executed before the system begins multi-user mode. A very simple /etc/rc script might contain the following basic commands:

```
# cat /etc/rc
PATH=/bin:/usr/bin              [User commands are in /bin and
                                 /usr/bin]
cat dev/null > /etc/utmp        [Empty the record of logged-in
                                 users]
```

```
/etc/fsck                        [Check the file systems for errors]
/etc/mount /dev/dsk/d0s2 /usr    [Mount the user directory /usr]
rm -f /usr/tmp/*                 [Remove temporary space directories]
rm -f /tmp/*                     [Remove more temporary space directories]
/etc/update                      [Start the update process]
/etc/cron                        [Start the cron process]
# _
```

Actually, in System V, the single file **/etc/rc** has been split into three separate files, as follows:

| **File** | **Function** |
|---|---|
| **/etc/brc** | Restore the mount table file **/etc/mnttab**. |
| **/etc/bcheckrc** | Set the date and check the file systems with **fsck**. |
| **/etc/rc** | Mount file systems; start the accounting processes and daemons; back up the super-user and **cron** logs; remove all temporary files; start the **lp** (or **lpr**) scheduler. |

These three files are invoked directly in the **/etc/inittab** file (described later in this chapter). In the remainder of this subsection, we'll discuss each of these files in detail, using sample scripts with comments interspersed:

1. Examine the boot read command file **/etc/brc**:

```
# cat /etc/brc
# This file restores the mount table file /etc/mnttab
if
      [ -r /etc/mnttab ]
then
      rm -f /etc/mnttab
fi
# Put root into mnttab
/etc/devnm / | /etc/setmnt
# _
```

2. Examine the boot check run command file **/etc/bcheckrc**:

```
# cat /etc/bcheckrc
# This file will set the system clock and check the file
# systems.  You can run this file from any terminal
# (see comments below).

trap "" 2
```

```
# Set and export the time zone

TZ=PST8PDT
export TZ

# ***** Check date

while :
do
     echo "Is the date `date` correct? (y or n) \c"
     read reply
     if      [ "$reply" = y]
     then
         break
     else
         echo "Enter the correct date (mmddhhmmyy): \c"
         read reply
         date "$reply"
     fi
done

# ***** Check the file system if necessary:

while :
do
  echo "Check the file systems? (y or n) \c"
  read reply

  case "$reply" in
    Y|y ) ;;
    N|n ) break ;;
    *   ) echo "Type either y or n.  Try again."
          continue ;;
  esac
  trap "echo Interrupt" 2

  /etc/fsck

  trap "" 2
  break
done

# Finally, display a message with the date included:
echo "System booted on \c"; date
# _
```

2. Examine the main run command file /etc/rc:

```
# cat /etc/rc
# This file will set the date, mount file systems,
# remove accounting files, back up the super-user and
# cron logs, start the daemons, and remove
# temporary files

# Set the time zone and command paths here:
TZ=PST8PDT
export TZ PATH

# This script is executed when going to
# multi-user (2) state:

set `who -r`
if [ $7 = 2 ]
then
         /etc/mount /dev/dsk/0s2 /usr

         # remove temporary files
         rm -f /tmp/* /usr/tmp/*
         rm -f /usr/spool/uucp/LCK*
         echo Temporary files removed

         # discard accounting information
         rm -f /usr/adm/acct/nite/lock*
         echo Old accounting files deleted

         # back up the super-user and cron logs
         mv /usr/adm/sulog /usr/adm/OLDsulog
         mv /usr/lib/cron/log /usr/lib/cron/OLDlog
         echo Super-user and cron logs backed up

# Set up the login file
if    [ ! -f /etc/wtmp ]
then
         > /etc/wtmp
         chmod 644 /etc/wtmp
         chgrp adm /etc/wtmp
         chown adm /etc/wtmp
fi
      # start the daemon processes
      /etc/cron
      echo cron started
      /usr/lib/errdemon
      echo errdemon started

      # start process accounting
      /bin/su - adm -c /usr/lib/acct/startup
      echo Process accounting started
fi
# _
```

## 32.2 Multi-user mode

Multi-user mode is the usual working mode for UNIX, the mode in which all the system's devices and file systems are ready for operation. Let's take a look at what happens when a user logs in.

### Logging in

Before a user can log into a UNIX system in multi-user mode, a chain of events must take place. First, during the transition to multi-user mode, the **init** process reads a file called **/etc/inittab** to determine which terminal lines are active. For each active line it forks a different **getty** process.

The **getty** process reads a file called /etc/gettydefs to obtain information about the terminal, then displays the `login:` prompt (or whatever is set in etc/gettydefs) at its terminal and waits for someone to enter a user name. As soon as someone does this, the **getty** process invokes another process called **login** and passes the user name to **login**.

The **login** process looks up the user's name in a file called /etc/passwd to find the user's password. If it finds one, it displays the `Password:` prompt, and waits for the user's response. If the password is incorrect, **login** displays a message and returns to **getty**; if the password is correct, **login** refers to /etc/passwd again to determine the user's group, home directory, and login program (usually one of the shells). Finally, **login** invokes the login shell, which then takes over.

If the login shell is the Bourne shell, it begins by running a program called /etc/profile to perform any opening tasks set up for all terminals following login. For example, /etc/profile displays the message of the day, which is stored in a file called /etc/motd. The system administrator can modify /etc/profile to include any shell script desired for the installation. If the shell script called .profile exists in the user's home directory, the Bourne shell then executes it and displays the shell prompt.

If the login program is the C shell, it begins by executing either a system cshrc file (not common) or a shell script called .cshrc, which must exist in the user's home directory. The .cshrc script provides the C shell with any secondary prompt the user may have selected, a history setting, and any aliases defined by the user. Next the C shell executes a .login script, which must also exist in the user's home directory. This script provides the primary shell prompt (% by default), terminal settings, a command search path, and possibly a message.

Here is a brief summary of the login processes:

| Screen Display | Process | Files Consulted |
|---|---|---|
| | **init** | /etc/inittab |
| `login: _` | **getty** | /etc/gettydefs |
| `Password: _` | **login** | /etc/passwd |
| `$ _` (default) | **sh** | /etc/profile, **$HOME**/.profile |
| `% _` (default) | **csh** | **$HOME**/.cshrc, **$HOME**/.login |

## The initialization command (Release 3.0) **init**

This section describes UNIX and XENIX System V, Release 3.0; for earlier versions of XENIX, see the section that follows. The **init** command establishes the *modes*, or *levels*, at which the UNIX system may run (primarily single-user and multi-user modes). It does this through a table stored in a file called **/etc/inittab** (initialization table). This table contains a set of line entries that describe all the processes to be activated at each level (or in each mode). For example, consider this line entry:

```
05:2:respawn:/etc/getty tty05
```

If we spread out the individual fields and add headings, we see this:

```
Identifier      Level      Action              Process

    05            2        respawn         /etc/getty tty05
```

This tells **init**, "Any time the system enters level 2 (multi-user mode), see if **getty** is running for terminal 5. If it isn't, respawn it (refork it) now." So when you enter **init 2**, this line entry is activated. Here is an entire sample table:

```
$ cat /etc/inittab
is:s:initdefault:
bt:2:bootwait:rm -f /etc/mnttab > /dev/console   # avoid dups
bl:2:bootwait:/etc/bcheckrc > /dev/console 2>&1  # bootlog
wt:2:bootwait:/etc/wtmpclean > /dev/console      # wtmpclean
bc:2:bootwait:/etc/brc > /dev/console            # bootrun
rc:2:wait:/etc/rc 1>/dev/console 2>&1          # run commands
pf::powerfail:/etc/powerfail 1>/dev/console 2>&1 # powerfail
ka:s:sysinit:killall                             # killall on shutdown
co:23:respawn:/etc/getty console console
00:23:respawn:/etc/getty tty00 9600
01:23:respawn:/etc/getty tty01 9600
02:2:respawn:/etc/getty tty02 1200
03:23:off:/etc/getty tty03 9600
$ _
```

Here are descriptions of the fields:

**Identifier** One or two unique characters to identify the line entry.

**Level** The number(s) (1-6) that indicate the level(s) at which the specified action is to take place. For multiple levels, type the numbers consecutively (for example, `23` for levels 2 and 3) or use a hyphen (for example, `1-6` for levels 1-6). You can also use `s` to indicate single-user mode. An empty field is equivalent to `1-6`. As of System V, Release 3.0, run level `3` is defined as Remote File Sharing mode (see Chapter 39, "RFS Maintenance," for details).

**Action** A word from the following list to describe what to do when **init** is executed at this level:

*initdefault*—Enter this level by default after booting (no process).
*sysinit*—Run this process before interacting with the system console.
*bootwait*—Run this process whenever the system enters a numeric run level after booting; wait for the process to complete.
*wait*—Wait for the process to complete.
*powerfail*—Run this process after a power failure.
*respawn*—Start the process, then restart it each time it completes.
*off*—Don't do anything (terminate the process if it's running).

**Process** A UNIX command line to be executed, subject to any restrictions specified by the action.

## Terminal Processes (Release 3.0)

If you need to activate or deactivate a terminal while the system is running in multi-user mode, there's an easy procedure for starting or stopping the associated terminal process. Just call up the **/etc/inittab** file with a text editor and change the *action* field from `off` to `respawn` (to activate a terminal) or from `respawn` to `off` (to de-activate a terminal). The change will take effect as soon as you execute the following:

```
# init q
# _
```

The **q** argument retains the current level of **init**, but starts processes for any lines in **/etc/inittab** that have been modified. If the **getty** process for this terminal is still active, you may have to terminate it with the **kill** command.

## THE INITIALIZATION COMMAND (BEFORE RELEASE 3.0) **init**

The **init** command under versions of XENIX V before Release 3.0 reads a file called **/etc/ttys** (instead of **/etc/inittab**). The file used by XENIX also contains a set of line entries; however, the format is different. For example, consider this line entry:

```
13tty05
```

If we spread out the individual fields and add headings, we see this:

```
Enable/Disable       Line Speed      Terminal ID
      1                   3              tty05
```

This tells **init** that terminal is currently enabled (1) for a line speed of either 300 or 1200 bit/s (3). Here is an entire sample table:

```
$ cat /etc/ttys
14console
13tty00
13tty01
13tty02
13tty03
14tty04
14tty05
03tty06
03tty07
$ _
```

Here are descriptions of the fields:

**Enable/Disable** One digit to indicate whether the terminal line is enabled (1) or disabled (0)

**Line speed** One digit to indicate the line speed and type of connection, according to the following table:

| Digit | Line Speed | Notes |
|---|---|---|
| 0 | 110,150,300,1200 | Variable speed for a line used for more than one terminal. |
| — | 110 | Teletype Model 33 or 35 |
| 1 | 150 | Teletype Model 37 |
| 2 | 9600 | On-line terminal |
| 3 | 300,1200 | Terminal with a Bell 212 modem |
| 4 | 300 | On-line terminal such as the DECWriter LS36 |
| 5 | 300,1200 | Terminal with a Bell 103 modem |

**Terminal ID** The name of the terminal

### Terminal Processes (XENIX before Release 3.0)

If you need to activate or deactivate a terminal while the system is running in multi-user mode, there's an easy procedure for starting or stopping the associated terminal process. Just call up the **/etc/ttys** file with a text editor and change the `Enable/Disable` field from `0` to `1` (to activate a terminal) or from `1` to `0` (to deactivate a terminal). The change will take effect the next time the terminal is logged on (or off).

## 32.3 Shutting down a UNIX system

Many UNIX systems continue running around the clock. However, it may be necessary to shut down the system occasionally for one reason or another. For such cases, System V provides a shell script in a file called **/etc/shutdown** that will shut down your system properly.

All you have to do is enter the following:

```
# /etc/shutdown
```

This shell script will perform the following functions automatically:

1. Check the identity of the user issuing the command.
2. Check to see whether anyone else is still using the system.
3. Warn all users of the impending shutdown with **wall** (write all).

4. Stop process accounting, daemons, and the lp spooler.
5. Conclude all unfinished disk activity with the **sync** command.
6. Unmount all mounted devices with the **umount** command.
7. Return the system to single-user mode with **init s**; all processes will be killed with the **killall** command in /etc/shutdown.

Once **shutdown** has run, you can then turn the power off:

1. Turn off the printer(s).
2. Turn off the terminals.
3. Turn off the main computer.

Here is a sample shutdown script, with comments interspersed:

```
$ cat /etc/shutdown
# This script will shut the system down

# Update the disks:
sync

# Set the command path names:
PATH=:/etc:/bin:/usr/bin; export PATH

# Check the name of the user who executed /etc/shutdown:
if    [ "`telinit q 2>&1`" ]
then  echo "$0: you must be root to use shutdown"
      exit 1
fi

# Set the grace period and display shutdown on console:
grace=${1-60}
echo "\nSHUTDOWN PROGRAM\n"
date; echo "\n"

# Change to directory root, see if anyone else is still
# logged in; if so, warn the other users of the impending
# shutdown:
cd /
users="`who | wc -l`"
if    [ $users -gt 1 ]
then  echo "Do you want to send your own message? (y or n): \c"
  read reply
  if    [ "$reply" = "y" ]
  then  echo "Type your message, followed by CTRL-D: \n"
    su adm -c /etc/wall
  else  su adm -c /etc/wall << !

    System maintenance is about to begin.  Please log off now.
    All processes will be killed in $grace seconds.
    !
  fi
  sleep $grace
fi

# Issue the final warning before proceeding with the actual shutdown:
/etc/wall << !
The system is now being shut down!
All processes will be killed.
```

```
!
sleep 5

# Proceed with the actual shutdown:
echo "Do you want to continue with shutdown? (y or n): \c"
read reply
if    [ "$reply" = "y" ]
then
  # empty the contents of the console buffer
  cat /dev/console > /dev/null

  # stop process accounting
  /usr/lib/acct/shutacct
  echo "Process accounting stopped"

  # stop error logging
  /etc/errstop
  echo "Error logging stopped"

  # stop the lp spooler
  /usr/lib/lpshut
  echo "lp spooler stopped"

  # update all disk activity for the last time
  sync; sync; sync

  # return to single-user mode
  init s

  # if the system is now in single-user mode,
  # exit immediately
  if    [ "$3" = "S" ]
  then  exit 1
  fi

  # if the system is still in multi-user mode, idle it
  # until the transition to single-user mode is complete

  sleep 10000

# If you decide not to proceed with the shutdown, the
# following message will appear on the console:
else
  echo "For help, call your system administrator."
fi
```

This concludes our discussion of the shutdown script **/etc/shutdown**.

## 32.4 Summary

In this chapter you learned how to start up and UNIX system and how to shut it down.

### Starting Up a UNIX System

Starting up a UNIX system involves turning on the power, booting the system (to enter *single-user mode*), entering the date, checking the file systems, logging in (in *multi-user mode*), and letting UNIX run the initialization script.

## Shutting Down a UNIX System

Shutting down a UNIX system involves notifying all users, killing all processes, unmounting all file systems, backing up the day's activities, possibly protecting your disks from damage, and turning off the power.

# 33

# Terminals

In this chapter you will learn how to adjust the operation of terminals for various purposes.

## 33.1 Identifying your terminal

Default settings for each terminal are placed in the **/etc/gettydefs** file. However, it is possible for any user to modify settings. The tool that UNIX provides for adjusting and viewing your terminal's settings is the **stty** (set terminal) command. Few people in the 1980s still use printing Teletype machines for terminals, but the term has stuck from the early days of UNIX, like bubblegum on the sole of your tennis shoes. If you type ***stty*** and press RETURN after the UNIX shell prompt, you will probably see something like this on the screen:

```
$ stty
speed 1200 baud; -parity
erase = ^h; kill = ^d;
brkint -inpck icrnl onlcr tab3
echo -echoe echok
$ _
```

### Terminal description

The four lines following the **stty** command contain information that describe the present settings of your terminal. Here are explanations of the items on the first two lines:

**speed 1200 baud** The speed, data rate, transmission rate, or baud rate is currently 1200 bit/s (bits per second). The rates possible are 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, and 38400. You must select the one that matches the rate for your computer.

**-parity** Disable parity detection (error checking) and set the character size to eight binary digits (or bits).

**erase = ^h** The character for backspacing over, and erasing, a character that you have just typed is Control-H (or ^h for short). Hold the CTRL with one finger and press H with another to enter this character. You can change this if you wish.

**kill = ^d** The character for erasing an entire command line and starting over is Control-D (or ^d). You can change this to another character if you wish.

## Changing the settings

To change the settings for your terminal, simply type the **stty** again, adding as many options as necessary. For example, to change the data rate to 9600, the erase character to CTRL-G, the kill character to CTRL-A, and enable parity detection, you could enter this command and press RETURN:

```
$ stty  9600  char erase ^g  char kill ^a  parity
$ _
```

This merely informs the system of the changes. The terminal's actual data rate, of course, must match what you set here (9600). This involves making a change in either a set of DIP switches or a screen menu. To see the new settings displayed on the screen, type ***stty*** by itself and press RETURN:

```
$ stty
speed 9600 baud; parity
erase = ^g; kill = ^a;
brkint -inpck icrnl onlcr tab3
echo -echoe echok
$ _
```

Now you see the new settings: 9600 bit/s, CTRL-G to erase a character, CTRL-A to discard an input line, and parity detection enabled. While these are the most common settings for a terminal, the **stty** command also allows over 60 others. But they all work just like the examples you have seen here.

Among these sixty options are six preset terminal selections for specific terminals. If you're using one of the terminals that most people use today, like the VT100, the Wyse 50, or the TeleVideo 950, you can forget it. But if you happen to be using one of the following "classics" from years ago, you're in luck:

| | |
|---|---|
| **tty33** | (Teletype Model 33) |
| **tty37** | (Teletype Model 37) |
| **vt05** | (DEC VT05) |

| | |
|---|---|
| **tn300** | (General Electric TermiNet 300) |
| **ti700** | (Texas Instruments 700) |
| **tek** | (Tektronix 4014) |

If you are using one of these terminals, just type its abbreviation after the **stty** command and press RETURN. UNIX will then make all the settings for you.

## 33.2 The **stty** command

The general format of the **stty** command for displaying options and existing settings is

```
$ stty [-a] [-g]
```

**-a** Display all options

**-g**
Display settings in argument format

The general format for changing the settings is shown here, with descriptions of the most common options following:

```
$ stty [ options]
```

### General settings

These settings control the overall operation of the terminal:

**cs***n* Set character size to *n* bits (5-8)

**line** *l* Set line discipline to *l* (0-127)

*char c* Set *char* to *c*, where *char* is (with default value at right):

| | | |
|---|---|---|
| **erase** | Erase preceding character | # |
| **kill** | Delete entire line | @ |
| **swtch** | Switch to shell control layer | SUB, or CTRL-Z |
| **eof** | Generate an end-of-file | EOT, or CTRL-D |
| **intr** | Generate an interrupt signal to terminate processes | DEL, or Rubout |
| **quit** | Generate a quit signal to create a core image file | FS, or CTRL-\| |

*rate* Set the data rate to *rate* (110, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400)

**sane** Reset all modes to values that make sense for a majority of terminals (character size 7, enable and generate parity, even parity, enable modem control, and so on)

*term* Set modes for one of the default terminals (tty33, tty37, vt05, tn300, ti700, tek)

**ek** Set erase to # and kill to @

**0** (zero) Hang up the phone line now

| | | | |
|---|---|---|---|
| **raw** | Enable raw data | **-raw** | Disable raw data (cooked) |
| **tabs** | Preserve tabs | **-tabs** | Convert tabs to spaces |
| **lcase** | Allow lower case (-LCASE) | **-lcase** | Allow upper case (LCASE) |
| **LCASE** | Allow upper case (-lcase) | **-LCASE** | Allow lower case (-lcase) |
| **nl** | No RETURN at end of line | **-nl** | Allow RETURN at end o line |
| **parenb** | Enable parity detection | **-parenb** | Disable parity detection |
| **parodd** | Select odd parity | **-parodd** | Select even parity |
| **parity** | Enable parenb; set cs to 7 | **-parity** | Disable parenb; set cs to 8 |
| **evenp** | Enable parenb; set cs to 7 | **-evenp** | Disable parenb; set cs to 8 |
| **oddp** | Enable parenb; set cs to 7 | **-oddp** | Disable parenb; set cs to 8 |
| **cread** | Enable receiver | **-cread** | Disable receiver |
| **clocal** | Disable modem control | **-clocal** | Enable modem control |
| **cstopb** | Select two stops bits | **-cstopb** | Select one stop bit |
| **hupcl** | Hang up on last close | **-hupcl** | Do not hang up on last clos |
| **hup** | Hang up on last close | **-hup** | Do not hang up on last clos |
| **loblk** | Block output from another shell layer | **-loblk** | Do not block output fron another shell layer |

With *raw* output, the system accepts exactly what you type literally, including any editing keystrokes such as backspace characters; with *cooked* output, the system accepts the edited result of what you type, allowing you to insert or delete characters on the command line before pressing RETURN. For example, suppose a user accidently types **lpr** instead of **lp** and then presses BACKSPACE to erase the `r` before pressing RETURN. Here are the two character strings that will be transmitted to the system in each mode:

| | |
|---|---|
| `l p r` CTRL-H | Raw output |
| `l p` | Cooked output |

## OTHER SETTINGS

There are about forty more options for **stty**, which concern various ways of handling parity, newline characters, tabs, line-feeds, backspaces, delays, fill characters, echoing, and so on. For details, refer to the **stty** command in Section 1 of the *UNIX User's Manual.*

## 33.3 Describing a terminal for **vi**

### Preliminary setup procedures

In versions of UNIX derived from Berkeley, the administrative directory **/etc** contains a file called **termcap** (terminal capability). This file contains abbreviated descriptions of all the terminals capable of running on your UNIX system. In System V, there is a different arrangement: a directory called **/usr/lib/terminfo** contains separate files with terminal information. The **termcap** descriptions are similar, but not identical, to the **terminfo** descriptions. (For an extensive list, see Appendix L, "Summary of **termcap** and **terminfo**").

Each description provides alternate names for a given terminal, the size of its screen, whether or not it supports basic functions like backspacing and automargins, and the codes required to clear the screen, home the cursor, move the cursor in different directions, and so on. A description may contain only one line for a simple terminal or over thirty lines for a sophisticated terminal.

### Sample entry for ***termcap***

For example, to describe the Lear Siegler ADM-3 terminal, **/etc/termcap** will contain an entry that looks something like this:

```
Names for terminal
|                |
13|adm3|3|lsi adm3:bs:am:li#24:co#80:cl=^Z
                   |                     |
                   Capabilities of terminal
```

This one-line entry (with descriptions added here) gives four alternate names for the terminal (separated by vertical bars):

```
13 (el three)     adm3      3       lsi adm3
```

It also gives five basic capabilities of the terminal (separated by colons):

| | |
|---|---|
| **bs** | Backspacing (ability to move the cursor back and erase) |
| **am** | Automargins (*wrapping* from the edge of the screen) |
| **li#24** | Twenty-four lines of display |
| **co#80** | Eighty columns per line |
| **cl=^Z** | Clearing (erasing the screen) with (CTRL-Z) |

## Sample entry for *terminfo*

For the same terminal, /usr/lib/terminfo/a/adm3 contains a record compiled from an entry that looks like this (with additional features on the second line):

```
Names for terminal
|              |
13|adm3|3|lsi adm3,cubl=^H,am,lines#24,cols#80,clear=^Z,
                 cud1=^J,ind=^J,cr=^M,bel=^G
                 |                                  |
                 +----- Capabilities of terminal-----+
```

There are three differences between this entry and the one described above for **termcap**: 1) the abbreviations are different; 2) **terminfo** uses commas to separate features instead of colons; 3) the **terminfo** entry has to be compiled instead of inserted directly. Note that `cubl=^H` here is the same as `bs` for termcap. The four additional **terminfo** features are as follows:

| | |
|---|---|
| **cud1=^J** | Cursor down is CTRL-J |
| **ind=^J** | Index is CTRL-J |
| **cr=^M** | Carriage return is CTRL-M |
| **bel=^G** | Bell (beep) is CTRL-G |

## Description already entered

If /etc/termcap or /usr/lib/terminfo already contains an entry for your terminal, all you have to do is identify your terminal to the UNIX shell. This involves setting the shell variable **TERM** (terminal). For example, if you are using a Lear Siegler ADM-3, all you have to do is select one of the alternate names and assign it to the shell variable **TERM**:

```
 Bourne Shell                     C Shell

$ TERM=adm3; export TERM         % setenv TERM adm3
```

Alternately, use **ed** or **vi** to call up a file in your home directory called either .profile (Bourne shell) or .login (C shell) and place this statement on a line by itself in that file. Then the UNIX shell will automatically treat your terminal as a Lear Siegler ADM-3 every time you log in.

## No description entered

If /etc/termcap or /usr/lib/terminfo does *not* contain an entry for your terminal, then (sigh) you will have to add an entry yourself. The sigh has been added because, with certain exotic terminals, this task can sometimes get very involved. In rare instances, it has even been known to result in "wailing and gnashing of teeth." If it becomes necessary for you to write your own terminal description, read the sections that follow. As soon as you have done that (and assigned your terminal), you are ready to start using **vi**.

### Summary of setup

Before you can even think about using **vi**, you have to make sure that two things have been done:

1. Someone has placed an entry in the file /etc/termcap or the directory /usr/lib/terminfo that describes your terminal.
2. You have assigned the terminal described in /etc/termcap or /usr/lib/terminfo to the shell variable **TERM** (either at the command line or in your directory's login file) and also *exported* this variable.

## 33.4 Designing an entry

Suppose you check /etc/termcap or /usr/lib/terminfo on your system and find that there is no entry for your terminal. Now what do you do? If you're really adventuresome, you can write one yourself. Otherwise, you can ask someone else for help. You may find that /etc/termcap is a read-only file on your system. If so, you will have to act as system administrator to allow yourself temporary write permission to write your entry to the file. In the meantime, you can set up your own little private termcap file while you are developing your entry for the main file. (You can also continue to use your private file, which may actually be faster and more efficient.)

### Setting up a private file

Let's say you decide to call your one-entry file /usr/robin/termcap. Begin an editing session with this file and enter the information for your terminal. How does the shell know that your entry is there instead of in the main public file? The answer is simple: you tell the shell. You have to name to the shell the file that contains the description of your terminal—unless the description is in /etc/termcap (or /usr/lib/terminfo). You also have to inform the shell of the name of the description itself (wherever it may be located).

Here are the commands to type at your terminal (assuming your terminal entry is called **newterm**):

```
 Bourne Shell                      C Shell
$ TERMCAP=/usr/robin/termcap     % setenv TERMCAP ~/termcap
$ export TERMCAP
$ TERM=newterm                   % setenv TERM newterm
$ export TERM
  or
```

```
$ TERMINFO=/usr/robin/terminfo % setenv TERMINFO ~/terminfo
$ export TERMINFO
$ TERM=newterm                 % setenv TERM newterm
$ export TERM
```

In this pair of examples, you are placing a description called **newterm** in a file called termcap (or terminfo). In the C shell, the **setenv** command includes assigning the shell variable and exporting it in a single command.

Typing these commands on the command line like this allows the assignments to remain in effect only for the duration of your current session with UNIX. As soon as you log off, the assignments are lost. If you would like the assignments to go into effect every time you log in, you can place them in your login file (.profile in the Bourne shell, .login in the C shell). Just add the command statements shown above to this file, and they will be activated automatically.

## Constructing the entry

The easiest way to construct a new entry is to look at the existing entries, find one for a terminal that is similar to yours, make a copy of it, and modify the copy. If none is similar to yours, then you will have to construct it yourself from scratch. Here are some basic pointers.

The process of actually describing your terminal in a termcap or terminfo entry involves looking at the terminal manual and finding the codes that operate your terminal. Then you find the name of the appropriate termcap or terminfo variable and type an assignment statement, separated from other assignment statements by either colons (termcap) or commas (terminfo).

For example, suppose your terminal manual tells you that the code that clears your terminal's display to the end of the line is CTRL-N. Looking under termcap (or terminfo) in Section 5 of the *UNIX Programmer's Manual*, you find that the name of the variable that describes this action is `ce` for termcap (`el` for terminfo). So you type this:

| termcap | terminfo |
|---|---|
| `ce=^N` | `el=^N` |

Suppose your terminal's code to clear to the end of the screen is CTRL-O. Since termcap's variable is called `cd` (`ed` in terminfo), you add another assignment to your "string of beads":

| termcap | terminfo |
|---|---|
| `ce=^N:cd=^O` | `el=^N, ed=^O` |

For control characters like these, type the caret symbol (^), followed by the character. To type an escape sequence (such as ESC**L**), use capital `E` preceded by a backslash to represent the ESC character: `\EL`. For example, suppose your terminal's code to insert a character is ESC**P**. Using either **termcap**'s variable (`ic`) or **terminfo**'s (`ich1`), you add another feature:

| termcap | terminfo |
|---|---|
| `ce=^N:cd=^O:ic=\EP` | `el=^N, ed=^O, ich1=\EP` |

## 33.5 Examples of entries

### The ADM-3A Terminal termcap

This is the actual entry as you enter it into the file:

```
la|adm3a|3a|lsi adm3a:\
        :li#24:co#80:am:bs:cm=\E=%+ %+ :ho=^^:\
        :ma=^K^P:nd=^L:up=^K:cl=1^Z:cm=\E=%+ %+ :\
        :MP=^Z:MR=^Z:sg#1:so=[:se=]:EG#1:ES=#:\
        :EE=#:ME=^G:BS=^U:CL=^V:CR=^B:RK=^L:UK=^K:\
        :LK=^H:DK=^J:HM=\036:
```

### terminfo/a/adm3a

This is the source entry, which must be compiled with a program called **tic** to produce a coded entry. You then place the coded entry in the database.

```
adm3a|3a|lsi adm3a,
        lines#24, cols#80, am,
        cup=\E=%p1%' '%+%c%p2%' '%+%c,
        cr=^M, home=^^, bel=^G, clear=^Z$<1>,
        cub1=^H, cud1=^J, ind=^J,
        cuu1=^K, cuf1=^L,
adm3a+|3a+|lsi adm3aplus|lsi adm3a+,
        use=adm3a, kcub1=^H, kcud1=^J,
        kcuu1=^K, kcuf1=^L,
```

## Additional information

This is about all we can say about the process of constructing an entry for **termcap** or **terminfo**. For a more detailed discussion, see the series of excellent articles by Bill Tuthill in the first three issues of *UNIX/WORLD*.[1]

The second article gives actual **termcap** entries for the TeleVideo 925 and Heath/Zenith 29. The third article gives entries for TRS-80, TRS-80 II, Apple II, Apple IIe, Heath/Zenith 89, Osborne 1, IBM PC with Kermit communications package (VT52 emulation), along with the Freedom, Micro Decision I, and Ann Arbor Ambassador. You may also be interested in *Reading and Writing Termcap Entries* by John Strang.[2]

Although **terminfo** is considered to be the "official" terminal database for System V, many systems are still using **termcap**. Both **termcap** and **terminfo** allow entries for printers as well as terminals. The rules for typing the entries are the same.

Note that XENIX also contains a file called **/etc/ttytype** that relates terminal types in **termcap** or **terminfo** to terminal ports in **/etc/ttys**. Each line in **/etc/ttytype** contains a **termcap** (or **terminfo**) name, a space, and the name of the corresponding terminal port.

[1] Tuthill, William, *UNIX/WORLD*, vol. 1, no. 1, pp. 59-61, Jan/Feb 1984; vol. 1, no. 2, pp. 53-56, Mar/Apr 1984); vol. 1, no. 3, pp. 53-56, May/Jun 1984 (*UNIX/WORLD*, 444 Castro Street, Suite 1220, Mountain View, CA 94041, (415) 940-1500).

[2] Strang, John, *Reading and Writing Termcap Entries*, Newton, MA: O'Reilly & Associates ($7.50).

## SETTING TABS **tabs**

In the absence of any explicit instruction otherwise, UNIX sets the tab stops on your terminal to every eight column (1, 9, 17, 25, ...). For terminals with hardware tabs that can be controlled from the computer, you can set the tab stops on your terminal for special purpose applications. The command that permits this is called **tabs**, and has the following general format:

```
$ tabs [ tabs ] [ +m n ] [ -T type ]
```

The *tabs* option can be selected from the following list. Column 1 always means the leftmost column, even if the terminal's columns are numbered from 0. If *tabs* is omitted, the standard columns (**-8**) are used by default.

**-a** IBM S/370 assembler (first format):
1, 10, 16, 36, 72

**-a2** IBM S/370 assembler (second format):
1, 10, 16, 40, 72

**-c** COBOL (normal format):
1, 8, 12, 16, 20, 55

**-c2** COBOL (compact format):
1, 6, 10, 14, 49

**-c3** COBOL (expanded compact format):
1, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62, 67

**-f** FORTRAN:
1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61

**-p** PL/I:
1, 10, 55

**-s** SNOBOL:
1, 10, 55

**-u** UNIVAC 1100 assembler:
1, 12, 20, 44

Three other types of entries are also allowed for *tabs*:

-*n* Every *n*th column. Here are several examples:

**-5** Every fifth column:
1, 6, 11, 16, 21, 26, 31, 36, 41, 46, 51, 56, 61, 66, 71, 76

**-8** Every eighth column (the default, required for the **nroff -h** option):
1, 9, 17, 25, 33, 41, 49, 57, 65, 73

**-0** No tabs

**a,b,**... Arbitrary tab stops selected by the user (up to a maximum of 40)

*file* Tabs to be read from the first line of the file named

The next option allows you to select a left margin for some terminals:

**+m***n* Offset the margin *n* columns from the left side. Here are some examples:

**+m5** Offset the left margin to column 6
**+m10** Offset the margin to column 11 (same as **+m** with *n* omitted)
**+m0** Do not offset the margin (normal for most terminals)

Finally, you have an option for identifying the type of terminal:

**-T***type* Select terminal *type* as listed in the directory /usr/lib/term. If you omit this option, **tabs** will use the value of **$TERM**.

## 33.6 Summary

In this chapter you learned how to set the basic operational features of a terminal, how to describe a terminal for **vi**, and how to set the tab stops on a terminal.

### Setting a terminal's features

The **stty** command allows you to set parity, stop bits, upper and lower case, and many other features. Six terminals that are rarely used by anyone today have pre-set selections: Teletype Model 33, Teletype Model 37, DEC VT05, General Electric TermiNet 300, Texas Instruments 700, and Tektronix 4014.

## DESCRIBING A TERMINAL FOR **vi**

Descriptions of terminal and printers are kept in one of two places:

| | |
|---|---|
| **/etc/termcap** | Still supported by System V (one file) |
| **/usr/lib/terminfo/*** | Standard for System V (a directory of files) |

The individual descriptions of features in **terminfo** are based largely on the names given in the ANSI X3.64 terminal specification of 1979, and are therefore different from the original descriptions used in **termcap**. Except for the difference in field separators (colons for **termcap**, commas for **terminfo**), abbreviations, and compilation for **terminfo**, the two systems are very similar. (See Appendix L, "Summary of **termcap** and **terminfo**").

## SETTING TABS

If your terminal has hardware tabs that can be controlled from the computer, you can set the tab stops on your terminal with the **tabs** command, which has a number of preset options for various programming languages like COBOL, FORTRAN, and different assemblers.

# 34

# Printers

With the advent of System V, the print spooler command has changed from **lpr** (lineprinter) to **lp**. This simple spelling change doesn't even hint how radically the UNIX print spooler has changed internally. While **lpr** supported only one system printer, **lp** supports a number of printers and allows you to group them together as classes by type.

For example, your system could have four dot-matrix printers, three daisy-wheel printers, and two laser printers. Then you could group your nine printers into three classes: dot matrix, daisy wheel, and laser. This would allow users to send text to a type of printer, rather than to a specific printer. The print spooler could then select a printer of the type you selected, routing your job to a printer that isn't currently busy.

This arrangement gives users and administrators considerable flexibility, but it also requires a new suite of UNIX commands, adding to the complexity of the printing process. Before describing the commands themselves, let's take a brief look at the procedure for connecting a printer to a computer and the spooling process.

## 34.1 Printer basics

### Connecting a printer

A printer can be connected to a computer using either a *serial* connection or a *parallel* connection. If the printer has a parallel connector, you just plug in the two ends of the cable and the job is done.

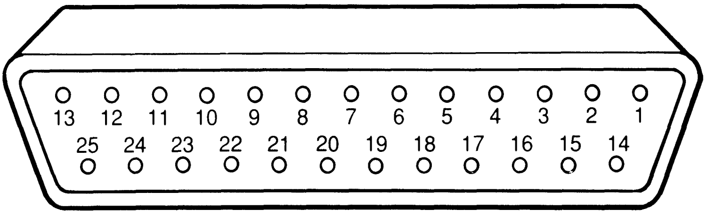
If the printer has a serial connector, the job may be quite a bit more involved. Most serial cables used today are RS-232C cables. However, unlike parallel cables, serial cables do not conform to any fixed configuration.

### RS-232C serial connection

While RS-232C cables may contain as many as 25 individual wires, with metallic pins at each end, different printers require different combinations of the 25 allowable wires and 50 allowable pins. The result is that the RS-232C standard is anything but standard.

Nevertheless, here's an attempt at explaining the basics of RS-232C serial connections. One thing RS-232C cables usually have in common is the size and shape of their connector plugs. The 25-pin version has 25 numbered pins, as shown in Figure 34.1.

FIGURE 34.1. 25 pin configuration



Pin 1 at one end may be connected to pin 1 at the other end, pin 2 to pin 2, pin 3 to pin 3, and so on; but not necessarily. A given pin on one side may be connected to a pin with another number on the other side.

Before discussing pin assignments, let's review a basic concept that determines the direction of the flow of information. Some years ago, before RS-232C became a general-purpose definition, its primary purpose was to connect a terminal to a modem. The terminal was referred to as *data terminal equipment* (DTE), while the modem was referred to as *data communications equipment* (DCE). In a nutshell, anything designated DTE sends on pin 2 and receives on pin 3, while anything designated DCE sends on pin 3 and receives on pin 2.

Today, computers and printers also use RS-232C connectors, but there is, unfortunately, no consensus as to how they should be designated. If a given computer is configured as DCE and a given printer is configured as DTE, it may be possible to connect the two machines with as few as three wires (and six pins), as shown in Table 34.1.

TABLE 34.1. RS-232C connection of opposites

| Computer (DCE) | | | Printer (DTE) | |
|---|---|---|---|---|
| Pin | Function | | Pin | Function |
| 2 | Receive data | ← | 2 | Transmit data |
| 3 | Transmit data | → | 3 | Receive data |
| 7 | Signal ground | | 7 | Signal ground |

In this example, it's possible to use "straight-through" wiring. In other words, since pin 2 is receiving on the computer side and transmitting on the printer side, it's possible to connect the computer's pin 2 to the printer's pin 2. Likewise, you can connect pin 3 to pin 3 to allow data to flow the other way.

On the other hand, if both machines are configured as DCE, you will have to cross pins 2 and 3 to achieve connection, as shown in Table 34.2.

TABLE 34.2. RS-232C connection of like machines

| Computer (DCE) | | | Printer (DCE) | |
|---|---|---|---|---|
| Pin | Function | | Pin | Function |
| 2 | Receive data | ← → | 2 | Receive data |
| 3 | Transmit data | | 3 | Transmit data |
| 7 | Signal ground | | 7 | Signal ground |

In this example, since both machines are DCE, pins 2 and 3 have to be cross-connected (the computer's pin 2 to the printer's pin 3 and the printer's pin 2 to the computer's pin 3).

While three wires may be sufficient for many connections, machines often use other signals as well: data set ready (incoming), data terminal ready (outgoing), clear to send (incoming), ready to send (outgoing), and data carrier detect (outgoing)—abbreviated DSR, DTR, CTS, RTS, and DCD. The pins commonly used to carry these signals are shown in Table 34.3.

TABLE 34.3. The Most Common RS-232C Signal Assignments

| | | | Direction | |
|---|---|---|---|---|
| Pin | Name | Abbreviation | DTE | DCE |
| 1 | Frame Ground | FG | N/A | N/A |
| 2 | Transmit/Receive Data | TD/RD | Out | In |
| 3 | Receive/Transmit Data | RD/TD | In | Out |
| 4 | Request to Send | RTS | Out | In |
| 5 | Clear to Send | CTS | In | Out |
| 6 | Data Set Ready | DSR | In | Out |
| 7 | Signal Ground | SG | N/A | N/A |
| 8 | Data Carrier Detect | DCD | In | Out |
| 20 | Data Terminal Ready | DTR | In | Out |
| 22 | Ring Indicator | RI | In | Out |

Unfortunately, there is no agreement on whether or not these additional signals are required for any particular machine. There is also no agreement on whether these various signals should be allowed to change freely or held at pre-determined voltages. Finally, there is no agreement on the functional purpose of any given signal.

Because of these inconsistencies, a system administrator may have to read manuals, experiment with wires, and do some detective work to figure out how the pins on one machine should be connected to the pins on another.

Once the basic wiring has been established, you have to make sure that both serial ports are set up for the same data rate, the same number of bits per data word, the same number of stop bits per data word, and the same parity scheme. A printer also has to be set up to interpret new lines and tabs correctly. You can take care of many of these details using the **stty** command, described in the previous chapter—provided that the actual switch settings agree with the **stty** settings.

## Print spooling

Like many UNIX tasks, printing requests do not cause text to be sent directly from user's directories to the destination devices. Instead, the print spooler places a copy of the file to be printed in a directory designated as the print spooling area. This copy of the file remains in the directory until the printer selected (or a printer of the class selected) becomes available. When the printer becomes available, a daemon (**lpsched**) sends the text from the directory (/usr/spool/lp) to the appropriate device file (/dev/lp).

## A sample system

To illustrate the lineprinter spooling system, let's assume that your system has the nine printers mentioned at the beginning of this chapter. Suppose you have four dot-matrix printers (all made by Epson), three daisy-wheel printers (all made by NEC), and two laser printers (both made by Hewlett-Packard). The members of a class of printers do not have to share a common brand name. But since they do in this example, we'll use the brand names to identify the classes; then we'll use the model names to identify the individual printers.

Class: **epson** (dot matrix)
  Printer 1: lx800
  Printer 2: fx86e
  Printer 3: ex1000
  Printer 4: lq2500

Class: **nec** (daisy wheel)
  Printer 1: sw3500
  Printer 2: sw5500
  Printer 3: sw7700

Class: **hp** (laser)
  Printer 1: ljet1
  Printer 2: ljet2

## 34.2 Features for users

An ordinary user can direct printed output either directly to one of the nine individual printers or to one of the three classes. If a user does not name a particular printer or class (and the system administrator has designated a default printer), the text will be routed to the default printer. Here are some examples, with and without the **-d** (destination) flag:

**Example 1**. Direct the printed output to a particular printer (in this instance, the Epson FX-86e).

```
$ lp -dfx86e ch.5
request-id is fx86e-3107 (1 file)
$ _
```

**Example 2**. Direct the printed output to a class of printers (in this instance, the Epson dot-matrix printers). Note that the system's response indicates the name of the next available printer in this class.

```
$ lp -depson ch.1 ch.2 ch.3
request-id is lx800-3124 (3 files)
$ _
```

**Example 3**: Direct the printed output to the default printer (in this instance, the Epson EX-1000). Again, the system provides the name of the printer implied in the **lp** command.

```
$ lp ch.4 ch.5
request-id is ex1000-3129 (2 files)
$ _
```

Now suppose that, after issuing the **lp** command in example 1, you realize that you gave the wrong filename. To remove the file from the print queue, you can enter the **cancel** command, using the request-id provided by the system after you originally requested printing. Then you can enter another **lp** command with the correct filename:

```
$ cancel fx86e-3107
request "fx86e-3107" cancelled
$ lp -dfx86e ch.6
request-id is fx86e-3138
$ _
```

Finally, to check the status of your printing requests, you can enter the **lpstat** command:

```
$ lpstat
total 28
mx80-217        robin          17462    Apr 6 09:09 on mx80
lx1000-346      robin           3685    Apr 6 09:12
fx86e-297       robin           8931    Apr 6 09:13
lx1000-408      robin           2366    Apr 6 09:17
mx80-329        robin           6328    Apr 6 09:19
lx1000-453      robin          23697    Apr 6 09:21
$ _
```

## 34.3 Features for system administrators

The commands for system administrators are stored in directory /usr/lib. To use these commands, you must be logged in as the lineprinter administrator (**lp**), with home directory /usr/spool/lp. To avoid having to type the full pathname of each command, it's also desirable to include /usr/lib in **lp**'s default execution path.

The commands provided for administrators make it possible to set up and configure the entire spooling system, start and stop the spooling system, suspend and release printing jobs at different stages of the system, and obtain detailed status information. Let's begin with the commands for allowing and disallowing output intended for a specific printer or class.

### Suspending printing

If one of your printers is malfunctioning, you can prevent users from queueing jobs for that printer without disrupting the flow of text to any of the other printers. For example, if you have to repair your LX-800, you can turn it off and issue the following command:

```
lp  reject lx800
destination "lx800" now rejecting requests
lp  _
```

Now you are free to work on the printer, and the system will reject any requests that name this printer. After the printer is working again, you can issue the following command to allow queueing again:

```
lp  accept lx800
destination "lx800" now accepting requests
lp  _
```

On the other hand, if this same printer simply has a paper jam or runs out of paper or needs a new ribbon, you can defer printing (but leave jobs queued for the printer) by issuing the following command:

```
lp  disable lx800
printer "lx800" now disabled
lp  _
```

Now you can unjam the paper, add new paper, or replace the ribbon while jobs queued for this printer remain suspended in the queue. With the problem fixed, you can issue the following command to resume operation:

```
lp  enable lx800
printer "lx800" now enabled
lp  _
```

## TURNING THE PRINT SPOOLER ON AND OFF **lpsched**

The commands that turn the entire print spooling system on and off are lpsched (on) and **lpshut** (off). It is customary to place **lpshed** in the **rc** file to start the print spooler with the UNIX system. When the spooler is running (and the spooler daemon is active), a lock file called **SCHEDLOCK** is created in **/usr/spool/lp**. The purpose of this file is to prevent the print spooler from being accidently initiated after it's already running.

## CONFIGURING THE SPOOLING SYSTEM **lpadmin**

Before starting the spooling system, you have to make each printer known to the UNIX system and associated with a logical device—that is, with a printer interface program stored in directory **/usr/spool/lp/interface**. The printer interface program can be either a default program provided by the UNIX system (called a **model**) or a custom program written in the form of a shell script.

The command that you use to associate the hardware (the printer) with the software (the printer interface program) is **lpadmin** (actually, **usr/lib/lpadmin**). You can also use this program to identify the type of device and to designate a default printer for the system, which will be discussed in a moment. You have to turn off the print spooler with **lpshut** before you can use this command.

The **lpadmin** command associates a device (**-v**) with a printer (**-p**) and assigns a printer interface program in one of three ways, using the following syntax:

$$\texttt{lp}\ \ \textbf{\textit{lpadmin}}\ \ \textbf{\textit{-p}}\ \textit{printer}\ \ \textbf{\textit{-v}}\ \textit{device}\ \left\{\begin{array}{l}\textbf{-m}\ \textit{model}\\ \textbf{-e}\ \textit{printer}\\ \textbf{-i}\ \textit{custom}\end{array}\right\}$$

Here, *printer* is a unique name of 1-14 alphanumeric characters, including underscores and *device* is the pathname of a hard-wired printer, a login terminal, or another file that can be written by **lp**. When making this assignment, you must also designate a printer interface program, which you can do in one of three ways, as indicated by the following examples:

1. Designate a printer interface program (mandatory):

   □ Select it from a list of model interfaces provided with **lp** (here we select a plain model called *dumb*):

   ```
   lp /usr/lib/lpadmin -pmx80 -v/dev/tty26 -mdumb
   ```

   □ Use the same interface already selected for an existing printer (here we choose *fx86e* for a subsequent printer):

   ```
   lp /usr/lib/lpadmin -pfx86e -v/dev/tty27 -emx80
   ```

   □ Use a custom program written by the system administrator (here we use a program called *prc*):

   ```
   lp /usr/lib/lpadmin -plq2500 -v/dev/tty28 -iprc
   ```

When assigning a printer name to a device, you also have the option of indicating that the printer is either hard-wired or a login terminal, in addition to the option of adding it to a *class*. Here are three examples:

2. Identify the printer further (optional):

   □ Indicate that the printer's device is hard-wired:

   ```
   lp /usr/lib/lpadmin -psw3500 -v/dev/tty21 -h
   ```

   □ Indicate that the printer's device is a login terminal:

   ```
   lp /usr/lib/lpadmin -psw5500 -v/dev/tty22 -l
   ```

   □ Add the printer to class *hp*:

   ```
   lp /usr/lib/lpadmin -pljet1 -v/dev/tty31 -chp
   ```

You can also use the **lpadmin** command to remove a printer from a class, designate a default class, or remove either a printer or a class of printers. Here are some examples:

3. Use the **lpadmin** command in other ways:

   □ Remove printer *ljet2* from class *hp*:

   ```
   lp /usr/lib/lpadmin -pljet2 -v/dev/tty32 -rhp
   ```

□ Designate ex1000 as the default printer:

```
lp /usr/lib/lpadmin -dex1000
```

□ Remove printer *mx80*, then remove class *epson*:

```
lp /usr/lib/lpadmin -xmx80
lp /usr/lib/lpadmin -xepson
```

After you have run the **lpadmin** program, run **accept**, **enable**, and **lpsched** before attempting to use the print spooler. Then log out as **lp**.

## System information

This section, which is keyed to the previous section, tells you where information about your printers is stored by the system. First, each time you attach a printer with **lpadmin**, the system creates a file in /usr/spool/lp/member that contains the name of the device file. For example, for the MX-80 printer, there will be a file called /usr/spool/lp/member/mx80 that contains the entry /dev/tty26.

Each time you create a printer class with **lpadmin**, the system stores a file in /usr/spool/lp/member, with additional details in /usr/spool/lp/class. For example, for class epson, there will be a file called /usr/spool/lp/class/epson that contains the following four entries:

```
/dev/tty26
/dev/tty27
/dev/tty28
/dev/tty29
```

The system stores its default interface programs in /usr/spool/lp/model. The original location of a custom interface program can be any suitable directory in the system. However, the system copies all active interface programs (default or custom) to /usr/spool/lp/interface.

Each time a user requests printing, the system stores a copy of the file(s) to be printed in a subdirectory of /usr/spool/lp/request. For example, suppose you request printing of your weekly report on the MX-80. Then a copy of your report will be held temporarily in /usr/spool/lp/request/mx80 until the MX-80 is available.

At the same time, the sequence number of the request (say mx80-2361) is stored in a file called /usr/spool/lp/seqfile. In addition, a log entry (sequence number, user identifier, device name, date, and time) is placed in a file called /usr/spool/lp/log. (The log file for the most recent UNIX run is called /usr/spool/lp/oldfile). If there is a default printer, its name is stored in /usr/spool/lp/default.

We'll conclude this section with two more files. An output queue for the spooling system is kept in a file called **/usr/spool/lp/outputq**, while status information is stored in another called **/usr/spool/lp/output**.

## CHECKING STATUS **lpstat**

The system administrator can obtain much more information about the printer spooler than an ordinary user by including the **-t** flag. The system will show dates, times, and information about the configuration of the system. Here is an example:

```
   # lpstat -t
   scheduler is running
   system default destination: ex1000
   members of class epson:
          mx80
          fx86e
          ex1000
          lq2500
   device for mx800: /dev/tty26
   device for fx86e: /dev/tty27
   device for ex1000: /dev/tty28
   device for lq2500: /dev/tty29
⋮

   mx80 accepting requests since Mar 21 09:36
   fx86e accepting requests since Mar 25 11:08
   ex1000 accepting requests since Apr 2 14:42
   lq2500 accepting requests since Apr 11 08:57
⋮

   printer fx86e is idle.  enabled since Sep 15 09:12
   printer lq2500 is idle.  enabled since Jul 19 13:17
⋮

   # _
```

## PRINTER INTERFACE PROGRAMS

Each printer in the spooling system must have a printer interface program to carry out the actual printing. The system provides a set of shell scripts to be used as model interfaces in directory **/usr/spool/lp/model**. (You can use a shell script, a C program, or some other kind of program as an interface program.)

A simple example of a shell script is given below. In reading this script, note that the data line that is passed to the interface program contains at least seven fields, which contain information derived from the **lp** command line:

**$0** The name of the interface program (including the subdirectory of **/usr/spool/lp** and the program name), such as **interface/prc**
**$1** The sequence number of the printing job, such as `mx80-2137`
**$2** The identifier of the user who entered the **lp** command, such as `robin`
**$3** An optional title for the print request such as `Weekly Report`
**$4** Optional number of copies to be printed (entered with the **-n** option of the **l** command), such as **2**
**$5** Additional options, such as **c** (copy) or **m** (mail)
**$6** File(s) to be printed, using full path name(s), such as **/usr/robin/reports/weekly**

Here is the shell script:

```
# cat /usr/spool/lp/interface/prc
#
# Simple lp(1) interface
#
# invoked by lpsched(1M) as
#   interface/printer request user [title] [copies] \
#                [options] file(s)
#
# echo form feed
echo "\014\c"
#
# banner user name
banner "$2"
echo
echo "Request id: $1  Printer: `basename $0`\n"
date
echo "\n"
#
# banner title, if given
if [ -n "$3" ]
then
        banner $3
fi
copies=$4
echo "\014\c"
shift; shift; shift; shift; shift
files="$*"
i=1
while [ $1 -le $copies ]
do
        for file in $files
        do
                cat "$file" 2>&1
                echo "\014\c"
        done
```

```
                i = `expr $1 + 1`
done
exit 0
# _
```

## 34.4 Summary

In this chapter you learned how to use the **lp** spooling system.

### Features for users

The new spooler system includes three new commands for all users: **lp** to queue a file for printing, **lpstat** to display printing status, and **cancel** to cancel printing either by request or by printer.

### Features for system administrators

The system includes these new commands for system administrators: **accept** and **reject** to accept and reject requests to a printer, **lpmove** to reroute requests from one printer to another, **enable** and **disable** to enable and disable routing of requests to a printer, **lpsched** to start the printing scheduler, **lpshut** to stop it, **lpstat** to display status, and **lpadmin** to configure the system.

To set up a spooling system for an installation, a system administrator has to configure the system and make sure each printer has a printer interface program.

# 35

# System Security

In this chapter, you will learn how to account for authorized users and reduce (or prevent) unauthorized use of your UNIX system.

## 35.1 Information about users and groups

### Users and groups

Each user authorized to use UNIX receives an identifier and a password. Logging in gives a user access to the system via the user's home directory. However, UNIX includes a system of protections for all directories and files. An ordinary user is not allowed to access directories and files outside that user's own directory unless the necessary permissions have been given.

To allow sharing of directories and files among users working on a common project, you can form a working group under UNIX. Then you can allow access within the group to common files, while withholding it from other users.

Each user is described in a single line entry in a password file called /etc/passwd; each group may be described on a line entry in a group file called /etc/group. These two files are described below.

UNIX allows three levels of access permission: to the individual user, to the user's group, and to all other users. Since this has been covered in Chapter 3, "The UNIX File System," it will not be repeated here.

### Information about users

The identifier and password of each new user have to be entered in a file called /etc/passwd, which can be modified only by a super-user, and contains the following seven information fields for each user:

- Identifier
- Encrypted password
- Numerical user identifier
- Numerical group identifier

- Any comments
- Home directory
- Default shell or dedicated program

Here are a few lines from a typical /etc/passwd file:

```
root:/H4Gql5HCW7uk:0:50:Super User:/:
daemon:x:1:50::/:
cron:x:1:50::/:
sys:qu.v0r0z90H1c:2:50:/sys:
bin:0b3cNIVqpk2A:3:50::/bin:
robin:MWxG24o.118fM:10:50:Rob Russell:/usr/robin:/bin/sh:
```

Taking a closer look at the sixth entry (robin), you can see the different fields, separated by colons, broken down as follows:

| | |
|---|---|
| `robin` | Login name |
| `MWxG24o.118fM` | Encrypted password |
| `10` | Unique user identifier, associated with each file |
| `50` | Unique group identifier, associated with each file |
| `Rob Russell` | Additional identification (optional) |
| `/usr/robin` | Home directory |
| `/bin/sh` | Login program: the Bourne shell |

Note that XENIX also has a file called /etc/default/passwd that specifies the minimum and maximum number of weeks a user must wait before changing passwords (MINWEEKS and MAXWEEKS) and the minimum number of characters allowed in a password (PASSLENGTH).

## INFORMATION ABOUT GROUPS

The name and password of each group is entered in a file called /etc/group, which contains the following four information fields for each user:

- Name of the group
- Encrypted password
- Numerical group identifier
- Members of the group, separated by commas

Here are a few lines from a typical /etc/group file:

```
root:/H4Gql5HCW7uk:1:root,daemon
sys:qu.v0r0z90H1c:2:bin,sys
secret:XyWK31p.723sP:10:robin,stan,robert,willy,tom
special:LuGf7e5.0bRfY:50:henry,janice,paul,dan,lisa
```

Taking a closer look at the third entry (secret), you can see the different fields, separated by colons, broken down as follows:

| | |
|---|---|
| `secret` | Name of the group |
| `XyWK31p.723sP` | Encrypted password |
| `10` | Numerical group identifier, associated with each file |
| `robin,stan,...` | Users who belong to this group |

### Adding a new user

To add a new user to the system, you have to enter a single-line entry into **/etc/passwd** like the one you just looked at. You also have to provide a new directory and set up an initialization file. Here are the steps:

1. Enter a line for the user in the password file:

   □ Log in as super-user:

   ```
   login: root
   Password:
   # _
   ```

   □ Begin an editing session with **vi** at the end of **/etc/passwd**:

   ```
   # vi + /etc/passwd
   ```

   □ Type a line entry for the new user:

   ```
   a
   pat::20:50:Pat Wyman:/usr/pat:/bin/sh
                (Bourne shell)
   pat::20:50:Pat Wyman:/usr/pat:/bin/csh
                (C shell)
   ESC
   ```

   □ End the editing session with **vi** and return to the shell:

   ```
   :wq
   # _
   ```

2. Provide a new directory for the user:

   □ Create a new home directory for the user:

   ```
   # mkdir /usr/pat
   # _
   ```

   □ Change the name of the owner from `root` to `pat`:

```
# chown pat /usr/pat
# _
```

□ Change the owner's group number to `50`:

```
# chgrp 50 /usr/pat
# _
```

3. Set up the user's initialization file in the user's own directory:

□ Log into the user's new directory:

```
# login pat
Password:
$ _
```

□ Begin an editing session with **vi** using one of two files, depending on which shell will be assigned to the new user:

```
$ vi + .profile             (Bourne shell)
$ vi + .login               (C shell)
```

□ Type lines to set and identify the user's terminal:

```
a
stty erase '^h' kill '^u'
TERM=vt100
```

□ Type lines to specify the user's pathnames for mail and for commands:

```
MAIL=/usr/mail/pat                          (mail)
PATH=:/bin:/usr/bin:/usr/pat/bin:           (commands)
```

□ In the Bourne shell, export the variables just assigned:

```
export TERM, MAIL, PATH
ESC
```

□ End the editing session and return to the shell:

```
:wq
$ _
```

The new user can easily create a password with the **passwd** command. Note that in XENIX, you can accomplish all of this with one simple command, which presents a sequence of prompts to guide you through the process:

```
# mkuser
```

### Removing a user

If a user leaves the system, you can write the user's files to tape, and then remove the user's directory:

1. Back up the user's files:

   □ Change to the root directory:

   ```
   # cd /
   # _
   ```

   □ Write all files owned by this user to tape:

   ```
   # find -user pat -exec tar cv {} ;
   # _
   ```

2. Remove the user's directory and login entries:

   □ Remove the user's directory:

   ```
   # rm -fr /usr/pat
   # _
   ```

   □ Remove the user's line entry in /etc/passwd

   □ Remove the user's name from the line entry in /etc/group

Again, XENIX allows you to remove a user with a single command, which prompts you for the user's login name:

```
# rmuser
```

## 35.2 Restricted accounts

So far we have discussed only ordinary user accounts. In this section we'll talk about accounts with restricted purposes.

### One-command accounts

For an ordinary user, the login program is the shell. In the last field of the user's entry in /etc/passwd the system administrator typically enters either /bin/sh (Bourne shell) or /bin/csh (C shell). However, it's possible to enter the pathname of any program here. You can use this to advantage to create accounts that serve no purpose other than to execute one command.

For example, suppose you'd like to be able to find out quickly which users are active on the system. You could enter a line like this in /etc/passwd:

```
who:uYe,7op!wq:20:1:Sys Admin:/:/bin/who
```

Then you could type ***who*** on any terminal and find out quickly. The system would execute this one command, and nothing more. As soon as the display of users appeared, you would be logged out again. Here are some more examples:

```
sync::20:1:Admin(0000):/:/bin/sync
tty::53:2:tty(0000):/:/bin/tty
lp:3iGv&ncW:71:2:lp(0000):/usr/spool/lp:/usr/bin/lp
```

If your company has developed an application program to run under UNIX, this is one way to make your program the only program that appears when your customers log into UNIX.

## The restricted shell **rsh**

By following the same kind of procedure, you can set up an account that presents the user with a *restricted shell.* This shell restricts the user to a selected set of UNIX commands and one directory, and prohibits redirection of output. Here's a typical entry in /etc/passwd, showing /usr/small as the login directory and /bin/rsh as the login program:

```
tiny::120:120::/usr/small:/bin/rsh
```

By modifying the user's .profile or .login start-up file, the system administrator can select the the commands permitted. Here's a typical example of a .login file:

```
PATH=/usr/little/bin:$HOME/bin
SHELL=/bin/rsh
export PATH SHELL
```

The user is now confined to one directory (/usr/small), and allowed to execute only the commands in /usr/little/bin, which the system administrator can copy from /bin and /usr/bin. The restricted directory /usr/little/bin may contain only half a dozen UNIX commands or many more, depending on what the system administrator decides to copy into it.

## Changing permissions **chmod**

### Another way to indicate permissions

In Chapter 3, "The UNIX File System," you learned how to change permissions for files and directories with the **chmod** (change mode) command using this kind of symbolic notation:

```
$ chmod u+x,o-w new.file
$ _
```

This command would add *execute* permission for the user and remove *write* permission for others (beside the user and the user's group). While you may prefer to use this notation, which is clear and simple, you should at least be aware of a numeric notation that can also be used.

Here is how the numeric notation works: Each of the nine permissions is assigned a binary value, depending on whether the permission is granted (`1`) or denied (`0`), so that the nine-character string translates into a nine-digit binary number. Here is an example:

| Owner | Group | Others | |
|---|---|---|---|
| r w - | r - - | r - - | [Permission string] |
| 1 1 0 | 1 0 0 | 1 0 0 | [Binary equivalent] |
| 6 | 4 | 4 | [Octal equivalent] |

As you can see here, each `r`, `w`, or `x` translates into a 1 (indicating permission granted), while each - translates into a 0 (indicating permission denied). So now we have a binary number (110 100 100) that is equivalent to the permission string (`rw- r- r-`). This binary number, in turn, can be read in octal notation as a number in base 8 (644).

## Converting to octal notation

In case you're a little rusty on octal numbers, here's a quick refresher. Each group of three binary digits translated into one octal digit, always following the same pattern:

| Binary | Octal | Binary | Octal |
|---|---|---|---|
| 0 0 0 | 0 | 1 0 0 | 4 |
| 0 0 1 | 1 | 1 0 1 | 5 |
| 0 1 0 | 2 | 1 1 0 | 6 |
| 0 1 1 | 3 | 1 1 1 | 7 |

One easy way to remember this is to think of the numerical value of each of the three digits of the binary number. Reading from left to right, the values are 4, 2, and 1. Simply adding the value for each 1 digit gives the desired octal value. Here's an example:

| | | | | | |
|---|---|---|---|---|---|
| r | | w | | - | [Symbolic permission string] |
| 1 | | 1 | | 0 | [Binary equivalent] |
| 4 | | 2 | | 1 | [Value of each binary digit] |
| 4 | + | 2 | + | 0 = 6 | [Octal equivalent] |

### Changing permissions with numbers

What all this means is that the following two commands are equivalent:

```
$ chmod u+rw,g+r,o+r text.file
$ _
```

```
$ chmod 644 text.file
$ _
```

Here are some of the more common sets of permissions for files and directories translated into their numerical equivalents:

```
Files

r-- r-- r--          [Read permission for everyone]
100 100 100    444

rw- r-- r--          [Read and write for user, read for everyone else]
110 100 100    644

Directories

rwx r-x r-x          [Read, write, and execute for user; read and exe-
111 101 101    755    cute for everyone else]

rwx rwx rwx          [Read, write, and execute for everyone]
111 111 111    777
```

## 35.3 Setting special file modes

In addition to the ordinary nine modes for files and directories, there are three special modes, which can be set only by the super-user and which apply only to executable files. Using binary notation, each mode is represented by one of three extra binary digits (or *bits*) placed in front of the ordinary nine digits, as shown here:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | [Set user ID bit: mode 4000] |
| 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | [Set group ID bit: mode 2000] |
| 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | [Sticky-bit: mode 1000] |

### Set user ID bit **Mode 4000**

This mode permits a user to execute programs which the user could not otherwise execute. It does this by allowing the user to borrow the user ID of the file's owner when executing the process. In other words, it changes the effective user identifier for the process from that of the user executing the process to that of the file's owner. When this mode is set, any user can execute a file that could otherwise be executed only by the super-user—which represents a security risk for the system. Here is a command to set the user ID bit and grant read, write, and execute permission to the user, with read and execute for everyone else:

```
# chmod 4755 track
# _
```

## SET GROUP ID BIT **Mode 2000**

This mode is analogous to Mode 4000 (Set User ID Bit), but applies to groups instead of users. This command would set the group ID bit and grant all permissions to the user and the group, but only read permission to others:

```
# chmod 2774 enter
# _
```

## STICKY BIT **Mode 1000**

This mode, which allows programs that are in constant demand to run more quickly, reserves swap-space for the file, regardless whether or not the file is actually being executed at the moment. You could say that having this mode set is something like having a downtown parking space held for you every day of the week, even when you stay home. This command would set the sticky bit and grant all permissions to the user and the user's group, with read and write to others:

```
# chmod 1776 independ
# _
```

## 35.4 More on permissions

### SUMMARY OF **chmod** PERMISSIONS

Here is a summary of the notation used by **chmod** to indicate permissions:

| Octal Code | Meaning | Symbolic |
|---|---|---|
| 4000 | Set user ID (UID) on execution | u+s |
| 2000 | Set group ID (GID) on execution | g+s |
| 1000 | Set the sticky bit (super-user) | u+t |
| 0400 | Allow read by owner | u+r |
| 0200 | Allow write by owner | u+w |
| 0100 | Allow execute by owner* | u+x |
| 0040 | Allow read by group | g+r |
| 0020 | Allow write by group | g+w |
| 0010 | Allow execute by group* | g+x |
| 0004 | Allow read by others | o+r |
| 0002 | Allow write by others | o+w |
| 0001 | Allow execute by others* | o+x |

* Execute a file or search a directory.

Here is the general syntax for a symbolic **chmod** command line:

$$\$\ \textbf{\textit{chmod}}\ \begin{bmatrix}u\\g\\o\\a\end{bmatrix} \begin{Bmatrix}+\\-\\=\end{Bmatrix} \begin{bmatrix}r\\w\\x\\s\\t\end{bmatrix}\ \textit{file}$$

Comments on symbolic notation:

1. The default for users is **a** (all), which is equivalent to **ugo**.
2. The absolute symbol (**=**) grants the permission explicitly shown, but removes all others; use it alone to remove all permissions.
3. Only the owner of a file (or the super-user) can change the file's permissions or set the user ID (UID).
4. Only a member of a group (or the super-user) can set the group ID (GID).
5. Only the super-user can set the sticky bit.

## SETTING DEFAULT PERMISSIONS **umask**

It is possible for the system administrator to establish default permissions for all files created by any user within the UNIX system. To do this, all you have to do is to place a command called **umask** (user file creation mask) in /etc/profile, using an octal code as an argument. The thing to be aware of is that the codes used with **umask** have the opposite effect of the codes used by **chmod**. In other words, while **chmod** grants permissions, **umask** denies permissions.

For example, here are two parallel commands:

| | |
|---|---|
| **chmod 444 text** | Grant permission to read to the owner, the group, and others |
| **umask 444** | Deny permission to read to the owner, the group, and others (that is, grant only permission to execute) |

This takes a little getting used to. Here are a few more examples:

| | |
|---|---|
| **umask 001** | Deny permission to execute to others |
| **umask 031** | Deny permission to write or execute to the group and deny permission to execute to others |
| **umask 066** | Deny permission to read or write to anyone except the owner |
| **umask 067** | Deny permission to read or write to the group and deny all permissions to others |
| **umask 077** | Deny all permissions to anyone except the owner |

Here is a summary of the codes for denying permissions with **umask**:

| Octal Code | Meaning |
|---|---|
| 0400 | Deny read by owner |
| 0200 | Deny write by owner |
| 0100 | Deny execute by owner |
| 0040 | Deny read by group |
| 0020 | Deny write by group |
| 0010 | Deny execute by group |
| 0004 | Deny read by others |
| 0002 | Deny write by others |
| 0001 | Deny execute by others |

Note that **umask** merely establishes default permissions for newly created files. Any user is free to change these with **chmod**, and any user can override the **umask** setting in /etc/profile by placing another **umask** command in .profile (Bourne shell) or .login (C shell). Here is where **umask** can be placed:

| | |
|---|---|
| /etc/profile | Establish default permissions for all users |
| .profile | Establish default permissions for one user (Bourne shell) |
| .login | Establish default permissions for one user (C shell). |

## 35.5 Maintaining security

Computer systems have a way of attracting people the way honey attracts bears. Most of the people who invite themselves into computer systems are just curious, but, unfortunately, a few are destructive. The basic strategy of an individual who seeks to do harm is two-fold: First, find a way to get into a system and then, having entered, perform the desired mischief.

To counter these efforts, a system administrator must also have a two-fold strategy:

- Try to keep unauthorized users out of the system.
- Try to prevent any who do enter from obtaining full access to the system.

Your first line of defense is effective use of passwords. Make sure every authorized user has a password. Having user accounts without passwords is like leaving a window or door open in your house. Anyone so inclined now has an easy way to get inside.

If unauthorized users still manage to enter your system, you must do whatever you can to keep them from obtaining super-user status. Allow only one user to log in as the super-user, and change the super-user's password from time to time. The safeguards built into UNIX are not absolutely fool-proof, but the relative security they provide is much better than no protection at all.

Plans for counteracting specific strategies, such as the Trojan Horse, can be found in the reference named at the end of the chapter.

## 35.6 Summary

In this chapter you learned about adding and removing users, setting up restricted accounts, changing permissions, setting default permissions for new files, and monitoring system activity.

### Information about users and groups

Information about users is kept in a file called /etc/passwd (the password file), and provides the user's identifier, password, user and group ID numbers, comments, home directory, and default shell. Information about groups is kept in a file called /etc/passwd (the group file), and provides the name of the group, the password, the group number, and the identifiers of members of the group.

To add a new user, add a new one-line entry for the user in **/etc/passwd**, create a new home directory for the user, change ownership of the directory to the user, log into the directory, and set up **.profile** in the user's new directory. (In XENIX, this can all be done with the **mkuser** command.) To remove a user, copy the user's files to tape, then remove the user's directory. (In XENIX, this can be done with **rmuser**.)

By making the appropriate entries in the **/etc/passwd** file, you can set up either a one-command account or a restricted shell account. Entries for these things appear in the file right along with the entries for ordinary users.

## PERMISSIONS

The **chmod** (change mode) command can be used with either symbolic strings (`rw-r-r-`) or equivalent octal (644) notation. Three special modes, which apply only to executable files, are set user ID bit (mode 4000), set group ID bit (mode 2000), and the sticky-bit (mode 1000).

The **umask** command, placed in **/etc/profile**, sets the default permissions for all files created by all users, using octal codes that deny permissions. Individual users can override the **umask** placed in **/etc/profile** by placing their own **umask** commands in either **.profile** (Bourne shell) or **.login** (C shell).

## FURTHER READING

For further reading on the subject of security in much greater detail, refer to the following. They contain many shell scripts for protecting your system from intruders.

Kochan, Stephen G. and Patrick H. Wood, *UNIX System Security*, Hasbrouck Heights, NJ: Hayden Book Company, 1985

Rieken, Bill and Webb, Jim, *Adventures in UNIX System Administration*, Santa Clara, CA: .sh consulting inc., 1988

# 36

# System Accounting

In this chapter you will learn how to set up and operate the process-accounting programs and how to monitor system activity. These will help you maintain security and measure the performance of your system to permit fine-tuning. They may also allow you to implement monthly billing of users.

## 36.1 Process accounting

### Record-keeping

Each UNIX system performs a considerable amount of record-keeping automatically. For example, the file **/usr/adm/pacct** contains one record for every process completed by the kernel. A file called **/etc/wtmp** contains records of date changes, reboots, shutdowns, and individual terminal connections. There are commands you can use to take advantage of this record-keeping and obtain detailed information about what is going on inside your system.

### All processes run today **acctcom**

For example, the **acctcom** (accounting command) provides you with a log of all system activity for a given day. It will tell you every command executed, showing the user's name, terminal, start and end time, real and CPU time, and the average amount of memory occupied. Here's part of a sample listing:

```
$ acctcom
COMMAND                          START    END          REAL      CPU     MEAN
NAME          USER   TTYNAME    TIME     TIME         (SECS)   (SECS)  SIZE(K)
#accton     root     ?          04:00:18 04:00:18      0.02      0.02     1.17
csh         bill     tty03      05:29:08 05:29:08      0.02      0.02     0.93
stty        bill     tty03      05:29:09 05:29:09      0.02      0.03     1.05
pwd         bill     tty03      05:29:08 05:29:09      1.00      0.18     4.12
sh          bill     tty03      05:29:08 05:29:09      1.00      0.08     5.26
who         bill     tty03      05:29:10 05:29:10      0.02      0.12     0.18
passwd      bill     tty03      05:29:19 05:29:51     32.00      2.17    18.57
echo        root     console    06:00:04 06:00:04      0.02      0.02     0.24
awk         bill     tty03      06:02:52 06:04:02     70.00     12.05    10.65
cron        root     ?          09:50:02 09:50:02      0.02      0.02     0.19
dmesg       root     ?          09:50:02 09:50:03      1.00      0.47     3.27
sh          root     ?          09:50:02 09:50:03      1.00      0.05     2.98
awk         beth     tty08      09:55:22 09:57:26    124.00     14.70     8.69
```

```
$ _
```

A question mark (?/) indicates a process initiated internally, such as **cron**. Note how much more time the two **awk** commands took than any of the others. An inordinate amount of execution time for a command could be the first clue leading to the detection of unauthorized activity. However, commands like **awk** and **nroff**, which perform numerical operations, can be slow on systems that lack hardware floating-point assist.

## 36.2 System activity accounting

So far in this chapter, we've taken a look at process accounting, which gives us information about system usage in terms of terminals, commands, and users. To probe into the inner workings of the system and measure internal performance, UNIX offers a set of commands that report on system activity. To optimize the performance of your system, you may want to use these to obtain statistics about activity of the CPU, terminals, disk and tape drives, buffers, system calls, swapping, file-access, queueing, messages, and semaphores. The commands we'll be discussing in this section are summarized in Table 36.1.

TABLE 36.1. Programs for Monitoring System Activity

| Command | Function |
|---|---|
| **timex** | Time a command and also provide information about system about system activity and process accounting. |
| **sar** | Report on system activity in tabular format and save the information in a file. |
| **sag** | Report on system activity in graphical format (histogram). |
| **sadp** | Check disk activity every second for a given period of time, then provide information about usage and seek distances in either tabular or graphical format. |

### TIMING A COMMAND **timex**

The **timex** command is an enhanced version of **time**, which works like a stopwatch for UNIX commands. When used without any options, **timex** is identical to **time**, as shown in this example:

```
$ timex ed file
?file
a
This is a new file to test some things out.
```

```
Here is the second line of this new file.
.
w
86
q

real        24.98
user         0.00
sys          0.28

$ _
```

Here, **timex** gives the time required for this brief editing session in three forms: real (24.98 seconds), user (0.00), and system (0.28). However, **timex** also has three options and six suboptions available:

```
         ┌-o  ┐          [Number of characters and blocks transferred]
$ timex  │-p x│ command  [Show process activity—see suboptions below]
         └-s  ┘          [Show all system activity during execution]
```

Process activity:

**-pr** Display user time divided by (system time + user time)
**-pt** Separate user time from system CPU time
**-ph** Display CPU time divided by elapsed time
**-pm** Display mean core size
**-pk** Display Kcore-minutes
**-pf** Display the fork/exec flag and exit status of the command

Here are a few examples of using **timex** with these options, in each case measuring a brief formatting process:

1. Time the **mm** command and display the number of characters transferred, along with the number of blocks read and written:

```
$ timex -o mm file


                                          - 1 -


        This is a new file to test some things out.  Here is
        the second line of this new file.


real        6.38
user        1.26
sys         2.46
```

```
CHARS TRNSFD = 498480
BLOCKS READ  = 93

$ _
```

2. Time the **mm** command and display the mean core size:

```
$ timex -pm mm file

                                              - 1 -


       This is a new file to test some things out.  Here is the
       second line of this new file.



real         5.80
user         1.18
sys          2.60

START AFT: Wed Feb  5 09:17:26 1988
END BEFOR: Wed Feb  5 09:17:31 1988
COMMAND                      START    END          REAL     CPU    CHARS BLOCKS    MEAN
NAME     USER TTYNAME        TIME     TIME       (SECS)  (SECS)   TRNSFD   READ  SIZE(K)
mesg    robin  tty07         09:17:26 09:17:26    0.02    0.22    28680      2    0.00
mesg    robin  tty07         09:17:26 09:17:26    0.02    0.22    28672      1    0.00
col     robin  tty07         09:17:27 09:17:31    4.00    0.13      380      1    0.00
nroff   robin  tty07         09:17:27 09:17:31    4.00    2.38   410112     61   34.80
mesg    robin  tty07         09:17:31 09:17:31    0.02    0.22    28672      3    0.00
sh      robin  tty07         09:17:26 09:17:31    5.00    0.45     1964      2    0.00
$ _
```

## SYSTEM ACTIVITY REPORT

**sar**
**sag**

To generate reports of system activity in tabular form and save the results in a file, use the **sar** (system activity report) command. The simplest way to use **sar** is to give the command with a single time interval in seconds. The command will report user time, system time, and idle time, as shown here for a one-second interval (09:20:57-09:20:58):

```
$ sar 1

colus colossus 2.5 Col/2000    05/16/87

09:20:57     %usr     %sys   %idle
09:20:58        1       15      83

$ _
```

If you specify more than one time interval, **sar** will also display averages. In the following example, we request three intervals of two seconds each:

```
$ sar 2 3

colus colossus 2.5 Col/2000    05/16/87

09:21:11     %usr     %sys   %idle
09:21:13        0        5      95
09:21:15        3       20      77
09:21:17        2        9      89

Average         2       11      87

$ _
```

If you would like **sar** to save the samples in a file, add the **-o** option in front of the time options, as shown here for the previous example:

```
$ sar -o samples 2 3
$ _
```

Here is the general form of the **sar** command for sampling current system activity and possibly saving the samples in a file:

```
         ┌-A┐                             [All system activity]
         │-a│                             [File access]
         │-b│                             [Buffers]
         │-c│                             [System calls]
         │-d│                             [Block devices]
$ sar    │-m│  [-o  file]  t  [ n]       [Messages and semaphores]
         │-q│                             [Queues]
         │-u│                             [CPU usage—the default]
         │-v│                             [Text, processes, i-nodes, and files]
         │-w│                             [Swapping and switching]
         └-y┘                             [Terminal devices]
```

Time is specified as follows:

**t** The length of each sample in seconds
**n** The number of samples to take (default: 1)

Here are some examples using options:

1. Sample terminal activity in five two-second intervals:

```
$ sar -y 2 5

colus colossus 2.5 Col/2000    05/16/87

09:25:01 rawch/s canch/s outch/s rcvin/s xmtin/s mdmin/s
09:25:03     1.3     0.0    47.5     0.0     2.7     0.0
```

```
09:25:05     0.9     0.0     32.3     0.0     1.4     0.0
09:25:07     3.2     0.0     30.7     0.0     3.7     0.0
09:25:10     1.8     0.0     28.4     0.0     2.3     0.0
09:25:12     0.9     0.0     46.3     0.0     1.8     0.0

Average      1.6     0.0     37.1     0.0     2.4     0.0

$ _
```

The abbreviations used have the following meanings:

| | |
|---|---|
| rawch | Number of characters in the raw queue |
| canch | Number of characters in the canonical queue |
| outch | Number of characters in the output queue |
| rcvin | Number of hardware interrupts from the receiver |
| xmtin | Number of hardware interrupts from the transmitter |
| mdmin | Number of hardware interrupts from the modem |

2. Sample file access in four one-second intervals:

```
$ sar -a 1 4

colus colossus 2.5 Col/2000    05/16/87

09:22:27  iget/s namei/s dirbk/s
09:22:28       2       1       1
09:22:29       2       2       1
09:22:30       4       3       2
09:22:31       4       2       0

Average        3       2       1

$ _
```

The abbreviations used have the following meanings:

| | |
|---|---|
| iget | Number of times the system located the i-node entry of a file |
| namei | Number of times the system searched the file system for a path |
| dirbk | Number of times the system read directory blocks in a search |

Once you've directed sample output to a file with the **-o** option, you can then extract the samples with a subsequent **sar** command using the **-f** option, as shown in here:

```
$ sar -f samples
   . . .
$ _
```

The general form of the **sar** command for extracting samples from a file is as follows (the data options, which were shown above, will not be repeated here):

```
$ sar [ options] [-s  start] [-e  end] [-i  sec] [-f  file]
```

The abbreviations have the following meanings:

| | |
|---|---|
| *options* | The options listed above |
| *start* | Starting time |
| *end* | Ending time |
| *sec* | Time interval in seconds (default: interval in the file) |
| *file* | The file from which to extract the samples |

If you prefer to display the sample information in graphical format (histogram), use **sag** instead of **sar**.

## DISK ACTIVITY REPORT **sadp**

You can run similar samples on disk activity with **sadp**. This command samples disk usage and seek distances every second for a given interval, then displays the results in either tabular or graphical format. Here is the general form of the **sadp** command:

```
       [-t]
$ sadp [  ]  [-d  device[- drive]]  s  [ n]
       [-h]
```

The options are as follows:

| | |
|---|---|
| **-t** | Use tabular format—the default |
| **-h** | Use histogram format |
| **-d** | Device (and optional drive number) |
| *s* | Length of each sample in seconds |
| *n* | Number of times to repeat sampling |

Here are some examples:

1. Report on disk activity for a one-second interval in tabular format:

```
$ sadp -t 1

CYLINDER ACCESS PROFILE

disk-1:
Cylinders        Transfers
  0 -   7        25
  8 -  15        2
 24 -  31        4
 32 -  39        1
 40 -  47        1

Sampled I/O = 33, Actual I/O = 386
Percentage of I/O sampled = 8.55


SEEK DISTANCE PROFILE

disk-1:
Seek Distance    Seeks
       0         19
  1 -  8         1
  9 -  16        2
 25 -  32        3
Total Seeks = 25

# _
```

2. Report on disk activity for a one-second interval in histogram format:

```
$ sadp -h 1

CYLINDER ACCESS HISTOGRAM
 [Histogram not shown here]
```

## Collecting information on system activity

It is generally desirable to collect information on system activity, even if you don't intend to produce reports regularly. Then the information will be there if you need it. The procedure for setting this up involves making sure the system activities counters restart from zero whenever the system is restarted and making sure that **cron** runs the data collection routines. Here is the procedure in detail:

1. Provide for automatic starting:

   □ Move to directory /etc:

   ```
   # cd /etc
   ```

   □ Make sure that /etc/rc contains a start-up command in the area for changing to multi-user mode:

   ```
   /bin/su - adm -c "/usr/lib/sa/sadc/ /usr/adm/sa/sa'date +%d'"
   ```

2. Provide for regular execution of the data collection commands:

   □ Move to directory /usr/lib:

```
# cd /usr/lib
```

□ Begin an editing session with **crontab**:

```
# vi + crontab
```

□ Add the following three lines to take care of daily processing:

```
0  *     *  *   0,6   /usr/lib/sa/sa1
0  18-7  *  *   1-5   /usr/lib/sa/sa1
0  8-17  *  *   1-5   /usr/lib/sa/sa1 1200 3
```

Here is a brief description of the commands involved:

1. **sadc**, which starts every time your system changes to multi-user mode, reads system counters from /dev/kmem and writes them to a file called /usr/adm/sa/sa*dd*, where *dd* represents the day of the month.

2. **sa1**, which is executed by **cron** at different intervals, depending on the time of day, invokes **sadc** to write system counters.

   This shell script may be invoked with a sampling interval in seconds and a number of iterations as arguments. For example, **sa1 1200 3** gives a sampling interval of 20 minutes (1200 seconds) and three iterations.

   In the crontab lines shown above, **sa1** is called hourly on weekends and during off hours on weekdays (6:00 p.m. to 7:00 a.m.), but is called every twenty minutes during normal business hours (8:00 a.m. to 5:00 p.m.).

3. **sa2**, which is executed manually instead of by **cron**, invokes **sar** to use the daily data file /usr/adm/sa/sa*dd* to generate a daily report in /usr/adm/sa/sar*dd*. The shell script **sa2** also deletes all data and report files after one week (seven days).

## 36.3 Summary

In this chapter you learned how to set up and operate both the process accounting and system accounting packages. The process accounting package, which gives you information about processes executed on your system, includes this command:

**acctcom** List every process executed today

The system accounting package, which gives you information about internal activities, includes these commands:

**timex** Time a UNIX command and give system information

**sar** Generate system activity reports in tabular format, with the option of saving the information in a file

**sag** Generate system activity reports in graphical format

**sadp** Generate disk activity reports in either tabular or graphical format

To collect system activity data, you have to use **/etc/rc** to start **sadp** when the system starts and use **/usr/lib/crontab** to have **sadc** invoked at regular intervals.

# Part VII

# Network Administration

In Part VII you will learn how to set up and operate the **uucp** system, which allows you to communicate with other UNIX systems and become part of a world-wide UNIX network. Although other communication programs are available for UNIX, **uucp** is the most widely used because it's a standard part of every UNIX installation. Because of changes in **uucp**, there will be one discussion for versions before Release 3 (Chapter 38) and one for versions after Release 3 (Chapter 39). (The rewritten version of **uucp**, part of the new Basic Networking Utilities, is often called Honey-DanBer after its authors Peter Honeyman, David A. Nowitz, and Brian E. Redman.)

You will also learn how to set up, maintain, and operate the Remote File Sharing (RFS) system, the major feature of Release 3, which allows different UNIX systems to share resources with each other.

While AT&T has been promoting its RFS, Sun Microsystems has been promoting a competing Network File System (NFS). At the time this book was going to press, there was a great deal of discussion about the future of networking under UNIX. NFS has gained an early lead in the race, but some observers see RFS as the eventual long-term winner.

# 37

# Introduction to Networking

Before describing **uucp** and the Remote File Sharing (RFS) system, we'll take a look at the underlying hardware and software that allow these applications to operate. Application programs like **uucp** and RFS rest on a number of layers of equipment, materials, and other programs.

## 37.1 Connecting computer systems

Before you can transfer files from one computer to another (or from a computer to a printer), you have to set up some kind of connection between them. This connection allows one of the two machines to transmit text to the other and also allows control information to pass back and forth between the machines. The connection may be in the form of a cable that connects the machines directly, a telephone line that connects them via AT&T's telephone network, or a network of computer systems (possibly a local area network).

### DIRECT CONNECTION

Two computers in the same room are usually connected with *serial* cables through their respective *serial ports.* Near the beginning of Chapter 34, "Printers," we discussed serial connection via RS-232C interface. Connecting two computers is about the same as connecting a computer to a printer. If the serial port on one computer is configured as DCE and the serial port on another is configured as DTE, you may be able to connect them with straight-through wiring, as shown in Table 37.1.

TABLE 37.1. RS-232C connection of opposites

| Computer A (DCE) | | | Computer B (DTE) | |
|---|---|---|---|---|
| Pin | Function | | Pin | Function |
| 2 | Receive data | ← | 2 | Transmit data |
| 3 | Transmit data | → | 3 | Receive data |
| 7 | Signal ground | | 7 | Signal ground |

On the other hand, if both computers are configured as DCE (or DTE), you may have to cross pins 2 and 3 to accommodate data flow in each direction, as shown in Table 37.2).

TABLE 37.2. RS-232C connection of like machines

| Computer (DCE) | | | Printer (DCE) | |
|---|---|---|---|---|
| Pin | Function | | Pin | Function |
| 2 | Receive data | ← → | 2 | Receive data |
| 3 | Transmit data | | 3 | Transmit data |
| 7 | Signal ground | | 7 | Signal ground |

In this example, since both machines are DCE, pins 2 and 3 have to be cross-connected (A's pin 2 to B's pin 3 and B's pin 2 to A's pin 3). A serial cable that is cross-connected in this way is sometimes called a *null modem.*

As pointed out in Chapter 34, whether you are connecting printers or computers, you may have to do some detective work to figure out how the pins on one machine should be connected to the pins on another.

Once the basic wiring has been established, you have to make sure that both serial ports are set up for the same data rate, the same number of bits per data word, the same number of stop bits per data word, and the same parity scheme.

## Telephone lines

If two computers are equipped with modems, they can dial each other and send information back and forth over the telephone lines. The modems are required to modulate the information (convert it from numbers to pitches) before transmission and demodulate it (convert the pitches back to numbers) after reception. If you are using an external modem, you must connect the computer to the modem as described in Chapter 34. Also, before you can begin dialing from either side, you have to make sure that both modems agree in data rate, bits per word, stops bits, and parity.

## Local area network

If a group of computers are all equipped with the circuit boards and connectors required, they can be linked in a *local area network (LAN)*, which allows sharing of the resources connected to the network. The section that follows describes networks in greater detail.

## 37.2 Some basics of networking

Networks, which connect computers and other devices to allow sharing of resources and information, play a large role in System V, Release 3. We'll begin with the hardware (physical link and data link), then cover the software (communication protocol and application protocol). As in any computer application, the hardware provides a foundation, over which the software is built like the structure of a building. As you will soon learn, a networking building is one of several stories, each built upon the story below it.

### The physical link

A network begins with a physical link. The physical link, or *medium*, is what provides the basic connection for a network. It is characterized by the speed at which it can transfer data (*data rate*) as well as its capacity (*bandwidth*). The greater the bandwidth, the more channels can be transferred simultaneously (or the more information can be conveyed over the same number of channels). *Bounded* media involve wires or cables, while *unbounded* media operate through the airwaves. The bounded media in use today include *twisted pair*, *coaxial*, and *fiber optic*.

Twisted pair, which originated in telephone networks, consists of a pair of copper wires wound around each other. Optimized for voice signals, it provides speeds of about 2400 bit/s over circuit-switched lines or 4800 bit/s over leased lines. The oldest and most developed medium, twisted pair is best suited for low-speed devices like terminals.

Coaxial cable, which consists of an inner copper wire, surrounded by an insulator and copper mesh, surrounded by an outer shield, has been widely used in cable television. This technology is supported by a variety of couplers, splitters, repeaters, taps, and controllers. With a typical bandwidth of 300-400 MHz, this medium can support over fifty color TV channels or thousands of voice or low-speed data channels. It can also support a single channel at very high data rates (12 Mbit/s).

Fiber-optic cable, made of glass or plastic, is the newest and most expensive of the bounded media. This cable must be supported by devices to convert electricity to light, then light back to electricity again. Its performance is astonishing: it has a bandwidth up to 3.3 GHz (3.3 trillion Hz) and a data rate up to 1 Gbit/s—with almost no interference. Around 2000 A.D. a fiber-optic Integrated Services Digital Network (ISDN) for voice, data, and video will probably replace the present analog voice network in businesses and homes.

Very briefly, the unbounded media include *radio* and *microwave*, which can be used in satellite communication, and *infrared.*

## The data link

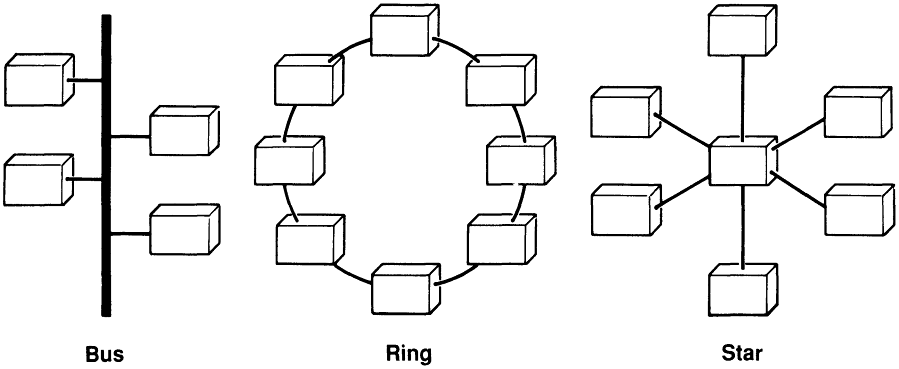
The next *layer* in a network is the data link. The data link, which relies on the physical link, is characterized by the method for transferring the data (*transmission method* or *signaling method*), the configuration of the *nodes* and *lines* in a network (*network topology*), the method for establishing a path (*switching*), and the method for allowing access to a network to multiple users (*access method*). Two transmission methods are the following:

- *baseband*—high-speed digital transmission on a single channel
- *broadband*—medium-speed digital, voice, or video transmission over multiple channels

There are three configurations, called network topologies, for connecting stations (see Figure 37.1):

- *bus*—stations connected along an open route
- *ring*—stations connected within a closed route
- *star*—secondary stations radiating from a central station

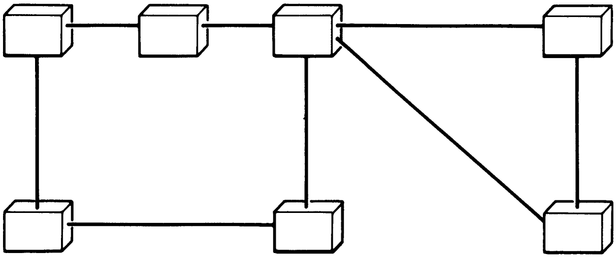
FIGURE 37.1. Network topologies.



An actual network is likely to represent a combination of these configurations, or a *hybrid* topology, as shown in Figure 37.2.

The method for establishing a path is called the *switching method. Circuit-switching*, which involves selecting a physical link and routing information through it continuously for a period of time, is the method used in telephone networks. This method is suitable for voice communication, but not for data communication, which typically occurs in short bursts. A method more suitable for computer systems is *packet-switching*, which employs groups of data called packets (usually about 64–256 bytes long). Each packet contains not only data, but also routing information.

FIGURE 37.2. Hybrid topology.



Use of a single channel by many users is determined by where control of allocation and access resides in the network, which user is allowed access to the channel, and how much use of the channel the user is allowed. The method for determining which user is allowed access to the channel is called the *access method.* Some methods in use today are the following:

- *frequency division multiplexing (FDM)*—allocating different frequencies, within the available bandwidth, to each user
- *time division multiplexing (TDM)*—allocating time slices in quick succession
- *token-passing*—a polling method that involves passing a token around to each station on the network)
- *Carrier Sense, Multiple Access with Collision Detection*, or *CSMA/ CD*—allowing any user to bid for the line at any time, then resolving contentions by allowing bidders to retry, after a randomly determined short delay, until one succeeds

## Low-level protocols

The third layer in a network is called a low-level protocol. Network standards can be adopted either by an independent organization or a private company. The independent organizations most widely recognized today are the following:

- *IEEE*—Institute of Electrical and Electronic Engineers
- *ANSI*—American National Standards Institute
- *ECMA*—European Computer Manufacturers' Association
- *ISO*—International Standards Organization
- *CCITT*—Consultative Committee on International Telephony and Telegraphy

Several hardware standards, or low-level protocols, which include a physical link and a data link, have been adopted for various applications:

- *Ethernet (IEEE 802.3)*—any medium; baseband or broadband; bus; CSMA/CD (joint standard of Xerox, DEC, and Intel)
- *Starlan (IEEE 802.3)*—twisted-pair cable; baseband; bus and star; CSMA/CD (AT&T standard)
- *Token bus (IEEE 802.4*—coaxial cable; broadband; bus; token-passing (General Motors standard)
- *Token ring (IEEE 802.5)*—twisted-pair or fiber-optic cable; baseband; ring; token-passing (IBM standard for larger computers)
- *PC Network*—coaxial cable; broadband; bus; CSMA/CD (IBM standard for personal computers)

This completes our discussion of the hardware aspect of networking, which includes a physical link (or medium) and a data link, with various combinations of characteristics used in different low-level protocols. Now we'll consider the software aspect of networking, high-level protocols, including communication protocols and application protocols.

## Communication protocols

With a low-level protocol in place, we have a foundation for the next level, the communication protocol, which takes care of data-routing, error-checking, process-connection, and formats. Here are six major protocols:

- *SNA* (System Network Architecture)—originally developed by IBM for hierarchical networks; recently enhanced for peer-to-peer networks with Advanced Program-to-Program Communications (APPC)
- *DNA* (Digital Network Architecture)—developed by Digital Equipment Corporation for its DECnet
- *XNS* (Xerox Network Systems)—developed by Xerox Corporation
- *TCP/IP* (Transmission Control Protocol/Internet Protocol)—developed by the United States Department of Defense's Advanced Research Projects Agency (ARPA) and later incorporated into 4.2BSD
- *ISO/CCITT OSI* (Open System Interconnection)—not yet fully developed, but supported by the U.S National Bureau of Standards and ECMA
- *ISO/CCITT X.25*—used for public networks like Tymnet and Telenet

IBM's APPC, which relies on new physical unit and logical unit protocols (PU 2.1 for transport (physical) level and LU 6.2 for upper session and presentation levels), is eventually expected to replace IBM's 3270 protocols.

## Application protocols

With a low-level protocol and a high-level protocol in place, it's possible to construct an application protocol at the highest level. Here are a few that have been developed for three major communication protocols:

For SNA, IBM has developed *DCA* (Document Content Architecture) for document formatting, *DIA* (Document Interchange Architecture) for document distribution, and *SNADS* (SNA Distribution Services) for delayed delivery services. *Disoss* (Distributed Office Support Systems) is an IBM application for host access via DCA and DIA.

For TCP/IP, the Department of Defense has approved *Telnet* for terminal access, *FTP* (File Transfer Protocol) for file transfer, and *SMTP* (Simple Mail Transfer Protocol) for electronic mail. In addition, Sun Microsystems has developed *NFS* (Network File System) to interconnect a variety of UNIX and non-UNIX systems.

For ISO/OSI, these have been adopted: *VTP* (Virtual Terminal Protocol) for terminal access, *FTAM* (File Transfer Access Method) for file transfer, *MMFS* (Multiple Message Forwarding System) for messaging, *CCITT X.400* (Message Handling Standard) for electronic mail, and *RFS* (Remote File Sharing) for resource-sharing in System V, Release 3.

## The ISO reference model

To summarize our discussion of networking concepts, let's take a look at the ISO Open System Interconnection (OSI) reference model, which is used to describe networks in terms of seven separate areas, or layers—the seven layers we've just discussed. SNA and DNA were originally derived from this model.

| **Layer** | **ISO Name** | **Function** | **General Layer** |
|---|---|---|---|
| 1 | Physical | Provide medium | Low-level protocol |
| 2 | Data Link | Connect stations | |
| 3 | Network | Route data | |
| 4 | Transport | Keep data intact | High-level protocol |
| 5 | Session | Connect processes | |
| 6 | Presentation | Provide data formats | |
| 7 | Application | Provide actual use | User application |

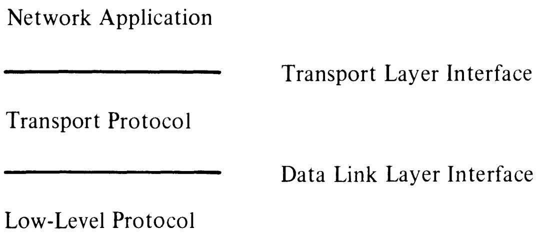
## 37.3 System V, Release 3

System V, Release 3 continues the process of opening UNIX to the rest of the computing world. Before this version, most attempts to connect UNIX to other systems resulted in an inconsistent assortment of unrelated programs, each relying on a different protocol and a different communication medium. Release 3 now lays the groundwork for an ever-growing set of networks by providing a structured character input/output system and a standard, modular model for networking systems.

The first step was to adopt Dennis Ritchie's Stream Input/Output System (STREAMS) as the standard character input/output system for System V. The second step was to adopt the reference model of Open Systems Interconnection (OSI) as the standard definition of protocol service interfaces for System V. The third step was to implement the OSI reference model within STREAMS, using the principles of *functional layering* and *service interfaces.*

Functional layering means separating a networking system into three different functional areas (or general layers): (1) application, (2) high-level protocol, and (3) low-level protocol. Each layer interacts with the next through a service interface, which provides a standard set of services to each upper layer. In this way, the application is independent of the protocol, and the protocol is independent of the medium. The Transport Layer Interface (TLI) provides protocol support to applications, while the Data Link Layer Interface (DLLI) provides hardware support to protocols, as shown in Figure 37.3.

FIGURE 37.3. Layers and interfaces.



Both **cu** and **uucp** have been re-coded to conform to this scheme, and therefore operate independently of any protocol or medium (Chapter 5, "Communication"). A new feature of Release 3, Remote File Sharing (RFS) has been designed from the start to conform to this scheme (Chapters 40, "Basic Resource Sharing," 41, "RFS Maintenance," and 42, "RFS Security").

In a sense, System V now provides for networking systems what UNIX as a whole provides for computer users. The UNIX shell provides the user with a standard interface, which shields the user from the kernel; while the kernel provides the shell with a standard interface, which shields the shell from the computer hardware. The layering of the OSI Reference Model performs similar functions. The Transport Layer Interface shields the application from the protocol, while the Data Link Layer Interface shields the protocol from the communication medium. For a summary of these and other new features of Release 3, see Appendix M, "UNIX versus XENIX."

## 37.4 Summary

This chapter describes how computers may be connected to each and to other machines and introduces networking.

### Connecting computers

Computers can be connected directly with cables, usually using RS-232C serial cables. They can also be connected via the telephone network, provided that both computers are equipped with modems. Finally, if a group of computers are equipped with the proper circuit boards and connectors, they can be linked within a network.

### Basics of networking

A network is constructed in layers, each resting on the one beneath, including a medium (bounded or unbounded), a low-level protocol, a high-level protocol, and a user application. Many of the features of System V, Release 3 rely on the ISO/OSI reference model for networks. This version of UNIX offers a standardized approach to networking applications by combining STREAMS with the reference model of Open Systems Interconnection (OSI). One of the major features of Release 3, Remote File Sharing, is based on this approach.

### For further reading

If you would like to learn more about networking, refer to the following:

Friend, George E., John L. Fike, H. Charles Baker, and John C. Bellamy, *Understanding Data Communications*, Dallas: Texas Instruments, 1984.

Tanenbaum, Andrew S., *Computer Networks*, Engelwood Cliffs, NJ: Prentice-Hall, 1981.

# 38

# Communication Before Release 3

## 38.1 Hardware requirements for **uucp**

To run **uucp**, you need a minimum of two UNIX systems and something to connect them together. You will also have to update three files to make sure your system is aware of the method of connection that you are using:

```
/etc/inittab
/usr/lib/uucp/L-devices
/usr/lib/uucp/L.sys
```

### Direct connection

If two systems are close to each other, you can connect them directly with a *null modem* (a serial cable with the send and receive pins crossed over). This is also referred to as a *hard-wired* connection, and can be used for systems up to several hundred feet apart, depending on the data rate used.

### Telephone lines

To operate over ordinary telephone lines, you will probably have to use an auto-dial modem. You can then operate the modem via any full-duplex communication line in the system, provided that the line has been disabled for ordinary use. The modem you select must operate at the same speed at which the modem on the other end of the line is operating (usually 1200 bit/s). For larger operations, you can install a special automatic call unit (ACU), which takes the place of an integral modem.

### Disabling the terminal

If you've connected your modem through an ordinary terminal port, you'll have to disable the getty process for that terminal in /etc/inittab. For example, suppose you plan to attach your modem to tty02. When you look at /etc/inittab, you'll see something like this:

```
02:2:respawn:/etc/getty tty02 1200
```

Use your editor (**vi** or **ed**) to change `respawn` in the third field to `off`. Now no one can use this terminal for logging into your UNIX system, but only for dialing out. This is how the entry should look after you modify it:

```
02:2:off:/etc/getty tty02 1200
```

### Informing the system

Now that you have a modem attached to a terminal port (or possibly a special dialing device) and you have the terminal port disabled, you have to make this known to **uucp**. To do this, you have to add one line entry in the file **/usr/lib/uucp/L-devices**. The line entry must identify the type of connection, the device name for the line being used, the associated ACU, the data rate, and (optionally) a protocol name. Here are several examples:

| | |
|---|---|
| `DIR tty01 0 9600` | Hard-wired device on tty01 at 9600 bit/s |
| `ACU tty02 tty02 1200` | Dial-up device on tty02 at 1200 bit/s |
| `DIR x25.3 0 300 x` | Hard-wired device using X.25 protocol |

Here is the general format of a line in **/usr/lib/uucp/L-devices**:

*type* *line* *call-unit* *speed* *[protocol]*

where

*type* indicates whether the line is hard-wired (DIR) or operating a dialing device (ACU).

*line* is the name for the device that is given in **/dev**.

*call-unit* is 0 for hard-wired devices (DIR), the same as *line* for dialing devices (ACU).

*speed* is the data rate (up to 9600 bit/s for DIR, usually 300 or 1200 for ACU).

*protocol* (optional) is `x` if you are using X.25 protocol.

If you select X.25 protocol, then *type* must be `DIR`, *call-unit* is `0`, and *speed* is ignored.

## 38.2 Software setup for **uucp**

In this section we'll discuss the **uucp** programs required to operate the system, along with the procedures for identifying your system in your password file and identifying other systems for your system.

## The programs

The programs required to run the system are stored in two directories: one for the operational commands (**/usr/bin**) and one for the maintenance commands and daemons (**/usr/lib/uucp**). Here is a summary of the modules:

**/usr/bin**

| | |
|---|---|
| **uucp** | Command for transfering files |
| **uux** | Command for remote execution of UNIX commands |
| **uustat** | Command for checking network status |
| **uuname** | Command for listing the names of systems in the network |
| **uulog** | Command for printing a log of **uucp** actions |

**/usr/lib/uucp**

| | |
|---|---|
| **uucico** | Daemon invoked by **uucp** or **uux** to interact with other systems |
| **uuxqt** | Daemon invoked by **uux** to execute UNIX commands remotely |
| **uuclean** | Command for cleaning up the **uucp** spool directory |
| **uusub** | Command for creating and monitoring a subnetwork |

There are also shell scripts that can be invoked to maintain the network each week (**uudemon.wk**), each day (**uudemon.day**), and each hour (**uudemon.hr**). The entire package comes with each standard UNIX system, and is usually installed along with the rest of the UNIX commands.

Once the system is installed, each **uucp** or **uux** command places the user's request in a queue and invokes the **uucico** daemon. The local **uucico** calls the other system and performs any file transfer requested. The system invokes a remote **uucico** to receive the file at the other end. When you request remote execution of UNIX commands with **uux**, **uucico** transfers a command file to the other system and invokes the **uuxqt** daemon to execute the command file and return any output to your system.

## The files

The record-keeping files that result from operation of the system are stored in the spool directories: **/usr/spool/uucp**). These files are used to store information about remote file transfers or command executions, lock devices, hold temporary data. Here is a summary of the files:

**/usr/spool/uucp**

C.*sysnxxxx* Work files are created in a spool directory whenever file transfers or remote command executions have been queued for a remote system. The suffixes are the name of the remote system (*sys*), the priority of the work (*n*), and sequence number (*xxxx*). Each file contains the following:

- Full pathname of the file to be sent or requested
- Full pathname of the destination or filename
- User ID
- Options
- Name of associated data file in the spool directory (`D.0` if **uucp -c** or **uucp -p** was given)
- Mode bits of the source file
- ID of remote user to be notified upon completion

X.*sysnxxxx* Execute files are created in the spool directory before remote command executions take place. The suffixes are the name of the remote system (*sys*), the priority of the work (*n*), and sequence number (*xxxx*). Each file contains the following:

- Requestor's login ID and system name
- Name(s) of the files requested for execution
- Standard input for the command string
- System and filenames for standard output
- Command string
- Option lines for return status requests

D.*sysnxxxxyyy* Data files are created whenever the command line specifies copying the source file to the spool directory. The suffixes are the name of the remote system (*sys*), the priority of the work (*n*), sequence number (*xxxx*), and sequence number extension (*yyy*). An extension is appended whenever there is more than one data file (D) for a given work file (C).

LCK.*name* Lock files, created in the /usr/spool/uucp directory for each device being used, prevent more than one user from accessing the same device at one time. The suffix .*name* is the name of either a system or a device.

TM.*pid.ddd* Temporary data files, created in /usr/spool/uucp whenever a file is received from another system, use the name of the other system (*name*). The suffixes are process ID (*pid*) and sequence number (*ddd*). The cleanup program **uuclean** removes TM files automatically.

## The directories

### /usr/spool/uucp

| | |
|---|---|
| **LOGFILE** | Directory of transaction log files |
| **SYSLOG** | Directory of system log files |
| **ERRLOG** | Directory of error log files |
| **LCK.*** | Directory of lock files |

## Identifying your own system

To allow users from other UNIX systems to log into yours, you have to set up a user identifier called **uucp**. Specify **/usr/spool/uucppublic** as the home directory and **/usr/lib/uucp/uucico** as the login shell. Then, after you've entered this user line, use the **passwd** command to give **uucp** a password. This is how the line in **/etc/passwd** should look before a password is given:

```
uucp::5:1:UUCP:/usr/spool/uucppublic:/usr/lib/uucp/uucico
```

You also need an identifier for the **uucp** administrator, the owner of all **uucp** spooled data files and programs. For this entry, make **/usr/lib/uucp** the home directory. With the standard name, here is how the line in **/etc/passwd** should look before a password is given:

```
uucp::5:1:UUCP:/usr/lib/uucp:
```

## Identifying other systems

Now that you have your own system identified for remote users, you have to identify other systems for your system. This involves finding out the phone numbers, identifiers, passwords, data rates, and calling hours of the other systems and entering them in a file called **/usr/lib/uucp/L.sys**. You can enter more than one line for the same system; **uucp** will treat them as alternates. Each line entry in this file has the following general format:

*name* *time* *device* *speed* *phone* *login*

where

*name* is the name of the other system.

*time* indicates when the system will accept calls, using these conventions: `Any` (any day of the week); `Wk` (any week day); `Su`, `Mo`, `Tu`, `We`, `Th`, `Fr`, `Sa` (days of the week); `0000-2400` (time of day). You can also specify an optional minimum retry period by typing a comma immediately after the time.

*device* is the device name (ACU or hard-wired).

*speed* is the data rate for the other system (usually 300, 1200, or 9600).

*phone* is the phone number of the other system (or `device` for hard-wired systems). You can enter either actual phone numbers or abbreviations that you have stored in a file called **/usr/lib/uucp/L-dialcodes**.

*login* describes the login process on the other system, using a series of prompts and responses. You can use the following conventions in a login sequence when required:

| | |
|---|---|
| `BREAK` | Send a `BREAK` character |
| `BREAKn` | Send *n* `BREAK` characters (1-9) |
| `EOT` | End-of-text |
| *string--string* | "If *string* doesn't appear, send a null and keep waiting for it." |
| `""` `(null string)` | No prompt expected |
| `\s` | Send a space character |
| `\d` | Delay one second |
| `\c` | Suppress newline at end of string |
| `\N` | Send a null character |

Here is an example of a line entry:

```
name   time device speed phone    login sequence

zebra Any  ACU    1200  325-1000 login:--login: uucp word: uusecret
```

Note that you can save space in the login sequence by abbreviating prompts (for example, `in:` for `login:`, `word:` for `password:`)

If you choose to use mnemonic abbreviations instead of entire phone numbers, you can assign them in **/usr/lib/uucp/L-dialcodes**. Here is an example, followed by another line in **/usr/lib/uucp/L.sys** that invokes the abbreviation: **/usr/lib/uucp/L-dialcodes**

```
hal 818-347-9000
```

**/usr/lib/uucp/L.sys**

```
gato  Any2300-0800  ACU  1200  hal  login:--login: uucp word: cat
```

Here `hal` is an abbreviation for 818-347-9000, as specified above. Note also the hours specified for incoming calls at this installation (11:00 pm to 8:00 a.m.). You can request a file transfer to this system with **uucp** or remote execution with **uux** any time of the day, but **uucico** will look up the time specified and defer the request until 11:00 p.m. Speaking of **uucico**, let's go on to the subject of setting up the files that control and maintain the system.

## 38.3 Control and maintenance of **uucp**

At this point you have the hardware and software in place and you've identified your system to others and other system to yours. Now we'll discuss the things you have to do to control and maintain your **uucp** system from day to day.

### Providing for startup functions

The **uucp** system records one line entry for every request submitted and every request processed. This usually amounts to hundreds or even thousands of lines per day. To keep this mountain of record-keeping from overwhelming you, it's a good idea to get rid of lock files each time you start up UNIX. The way to do this is to include these lines in your **/etc/rc** file:

```
    rm -f /usr/spool/uucp/LCK*
```

This is shown in Chapter 32, "Startup and Shutdown," in the listing of **/etc/rc**, page 491.

### Programming periodic functions

It's essential to make sure that the system invokes the **uucico** daemon at least hourly to process requests that have been queued. It's also important to keep **uuclean** running at regular intervals to get rid of any requests that have been around longer than 72 hours. This command will check the **uucp** directory (**/usr/spool/uucp**) and delete any files (with certain filename prefixes) beyond the time limit.

On systems with a lot of traffic, just like the downtown areas of large cities, you find a lot of congestion. The backlog of unprocessed files can become overwhelming if you don't take address it regularly. You can let two daemons, **uudemon.hr** and **uudemon.day**, take care of these two tasks.

The following entry will exercise the connection to each system on the network and collect statistics on traffic for the past 24 hours (for details on syntax, refer to Appendix J, "Summary of System Administration"):

```
0  8  *  *  *    /bin/su - uucp -c /usr/lib/uucp/uusub -c all -u 24
```

### Controlling user access

We'll conclude this section by discussing how you can control access to your system by your own users and by users of other systems. This involves setting up a file that describes user access (**/usr/lib/uucp/USERFILE**). This file allows you to control user access by specifying

- Files that a user of your system can access
- Files that a user of another system can access
- The login name for another system
- Whether to call another system back to verify its legitimacy

Each line in this file is entered in the following format:

*login,system [c] path [path] . . .*

where

*login* is the login name for either a user or another system.
*system* is the other machine's system name.
*c* (if entered) indicates that your system must call the other system back before any requests can be honored.
*path* is the prefix of a pathname allowed for *system*.

Omission of one of these removes all related restrictions. Here are some examples, with and without restrictions:

| | |
|---|---|
| `al,max /usr/bin` | This line permits a user from machine `max` to log in with login name `al` and transfer files that belong to **/usr/bin**. |
| `bo,win c /usr/lib` | This line permits a user from machine `win` to log in with login name `bo` and transfer files that belong to **/usr/lib**, but only after your system first calls back to verify the identity of the caller. |
| `tu,vaq /usr/bin` | These two lines allow a user from any machine to |
| `tu, /bin /usr/bin` | log in as `tu`. System `vaq` is restricted to files in **/usr/bin**, but all others have access to files in both **/bin** and **/usr/bin**. |
| `root, /` | These two lines allow `root` to transfer any file |
| `, /usr/test` | in your system, but restricts all others to files in **/usr/test**. |
| `, /` | This line removes all restrictions, allowing any user from any system to transfer any file in your system. |

## 38.4 Trouble-shooting **uucp**

Once you have **uucp** installed, you may feel like breathing a big sigh of relief. But don't breathe too deeply because your problems have just begun. (Just kidding!) In this section we'll discuss a few of the most common difficulties you may encounter.

### Defective equipment

From time to time a faulty ACU or modem may hinder users on your system from reaching other systems or accepting files. If so, entries in your LOGFILE will make repeated references to the line in question. You can then take the following measures to isolate the problem:

1. Try calling another system over the suspected line using **cu**.
2. See whether calling attempts by either **uucp** or **cu** have resulted in lock files (files of the form /usr/spool/uucp/LCK.... ).
3. Carry out hardware self-tests on the ACU or modem in question.
4. Make sure that you have set the correct options for the ACU or modem.
5. If you have a special ACU interface, make sure that it's sending data to its ACU.
6. Make sure you have the correct phone number for the other system.
7. Test the modem by having someone dial into system through it.
8. Make sure the cable is firmly connected and free of defects.

### Running out of disk space

The traffic on your system may be flooding your spooling area with files. If the area fills up completely, then your **uucp** system will come to a standstill until additional space becomes available. Then it will become inundated by another wave of requests. There are several ways you can deal with this problem:

1. Set aside a larger file system to handle spooling.
2. Schedule your daemons more frequently (possibly every half hour instead of every hour).
3. Set a shorter deadline for disposing of old files (possibly every 48 hours or even every 24 hours instead of 72 hours).

### Losing contact

Other systems may be changing their phone numbers, login sequences, or passwords without telling you. Then your equipment will waste time trying to call through to them. If you suspect this problem, you can begin by examining the LOGFILE and ERRLOG entries associated with another system. You can also check out the other system by invoking its **uucico** daemon from your system. Here's the procedure:

1. Queue a job on the other system (cougar):

   □ Execute the following command from your system:

   ```
   $ uucp -r test.file cougar!~/phil
   ```

   □ This command will cause the other system to queue the job without invoking **uucico** to process the job.

2. Invoke the processing daemon for system cougar directly:

   □ Execute the following command from your system:

   ```
   $ /usr/lib/uucp/uucico -r1 -x4 -scougar
   ```

   □ This command line will start the daemon in master mode with level 3 debugging.

Debugging levels 1-4 are suitable for ordinary users and part-time administrators; higher levels require knowledge of the internal workings of **uucico**.

Some systems simply act as nodes in the network without actively participating. If such a system has any files intended for your system, you can poll the system with the **uusub** command. For example, suppose there is a passive system called bambi. You can poll bambi with a command like the following to initiate a call:

```
$ /usr/lib/uucp/uusub -cbambi
```

This is useful in the **uucp uudemon.hr cron** job to poll systems automatically.

## 38.5 Direct networking (XENIX only)

You can set up direct networking among as many as twenty-five XENIX hosts over RS-232 serial ports with Micnet (pronounced Mike-net). You can also configure Micnet with a **uucp** gateway for communications over telephone lines. XENIX provides one command for setting up a Micnet system via screen menus (**netutil**), another for remote command execution (**remote**), and a third for file transfer, or remote copy (**rcp**).

## 38.6 Summary

In this chapter you learned the procedures for setting up and operating the **uucp** system, which allows you to exchange files with other UNIX systems and execute commands on those systems.

### Hardware requirements for **uucp**

The basic hardware requirements are two UNIX machines connected by either a hard-wired connection or an auto-dial modem attached to a terminal port. (Larger systems may employ a special ACU in place of a modem.) You usually have to disable the terminal, if used, in /etc/inittab, then place a line entry in /usr/lib/uucp/L-devices that describes what you are using.

### Software setup for **uucp**

The **uucp** system includes nine basic modules, five in /usr/bin (**uucp**, **uux**, **uustat**, **uuname**, **uulog**) and four in /usr/lib/uucp (**uucico**, **uuxqt**, **uuclean**, **uusub**). It also includes three shell scripts for maintenance (uudemon.wk, uudemon.day, and uudemon.hr). The two primary commands for ordinary users are **uucp** (file transfer) and **uux** (remote execution). These programs invoke two daemons, **uucico** (calling other systems) and **uuqxt** (executing remote commands).

You have two tasks to perform here: 1) identifying your system to others by placing a line entry in /etc/passwd (plus another entry for the administrator); 2) identifying other systems to yours by placing a series of line entries in /usr/lib/uucp/L.sys. Each line in this file must give the name of the system, calling times, device name, speed, phone number, and login sequence. You can set up abbreviations for phone numbers in /usr/lib/uucp/L-dialcodes if you like.

### Control and maintenance of **uucp**

To keep your **uucp** system running smoothly, processing new jobs and getting rid of old jobs regularly, you need to add lines to key files. Your /etc/rc file should contain lines to remove lock files, log files, and system logs; your crontab file should contain lines to invoke **uucico** and **uclean** at least hourly.

To control user access to your system, you need to add lines to a file called /usr/lib/uucp/USERFILE. Each line indicates which users and systems can and cannot access which files; it can also require a callback to verify the caller's identity.

## TROUBLE-SHOOTING **uucp**

This section concludes with a few suggestions for solving the problems of defective equipment, running out of disk space, and losing contact with other systems. You can test a faulty ACU or modem by trying to call another system using **cu** instead of **uucp**. Then you can look for lock files, perform self-testing, check the options, verify the phone number, dial into the suspected modem, and examining the cable.

You can deal with the problem of running out of disk space by assigning a larger file system to handle spooling, scheduling your daemons to run more often, and getting rid of old files more frequently.

You can handle the problem of losing contact with other systems by queueing a job on the system, then invoking **uucico** directly from your system. For systems that do not actively participate in the network, you can use the **uusub** command to poll them for files.

# 39

# Communication After Release 3

## 39.1 Hardware requirements for **uucp**

To run **uucp**, you need a minimum of two UNIX systems and something to connect them together. You will also have to update three files to make sure your system is aware of the method of connection:

```
/etc/inittab
/usr/lib/uucp/Devices
/usr/lib/uucp/Systems
```

### Direct connection

If two systems are close to each other, you can connect them directly with a *null modem* (a serial cable with the send and receive pins crossed over). This is also referred to as a *hard-wired* connection, and can be used for systems up to several hundred feet apart.

### Telephone lines

To operate over ordinary telephone lines, you will probably have to use an auto-dial modem. You can then operate the modem via any terminal port in the system, provided that the terminal port has been disabled for ordinary use. The modem you select must operate at the same speed at which the modem on the other end of the line is operating (usually 1200 bit/s). For larger operations, you can install a special automatic call unit (ACU), which takes the place of an integral modem.

### Local area network

The third possibility is to have your system linked to other systems in a local area network (LAN). In this case, you will be able to send and receive data to the other systems according to the protocol defined for the network.

## Disabling the terminal

If you've connected your modem through an ordinary terminal port, you will have to disable the **getty** (or **uugetty**) process for that terminal in **/etc/inittab**. For example, suppose you plan to attach your modem to **tty02**. When you look at **/etc/inittab**, you'll see something like this:

```
02:2:respawn:/etc/getty tty02 1200    [one-way connection]
```

or

```
02:2:respawn:/etc/uugetty tty02 1200 [bidirectional connection]
```

Use your editor (**vi** or **ed**) to change `respawn` in the third field to `off`. Now no one can use this terminal for logging into your UNIX system, but only for dialing out. This is how the entry should look after you modify it:

```
02:2:off:/etc/getty tty02 1200        [one-way connection]
```

or

```
02:2:off:/etc/uugetty tty02 1200      [bidirectional connection]
```

## Informing the system

With a modem or ACU attached to a terminal and the terminal disabled if necessary, you have to make this known to **uucp**. To do this, you have to add one line entry in the file **/usr/lib/uucp/Devices**. The line entry must identify the type of connection, the device name for the line being used, the device name of the 801 dialer (ACU only), the data rate (and possibly a dialer code), and one or more dialer/token pairs. (Note that several fields entered in **Devices** must match fields entered in the **Dialers**, **Systems**, or **Dialcodes** files.) Here are several examples:

| | |
|---|---|
| `Direct tty01 9600 direct` | Hard-wired device on tty01 at 9600 bit/s |
| `ACU tty02 - 1200 hayes` | ACU on tty02 at 1200 bit/s, Hayes modem |
| `micom tty03 - 1200 micom \D` | LAN switch on tty03 at 1200 bit/s |
| `Starlan,e starlan - - TLIS \D` | AT&T Starlan with `e` protocol |

Here is the general format of a line in **/usr/lib/uucp/Devices**:

```
Type   Line   Line2  Class   Dialer,Token [Dialer,Token]
```

where

`Type` indicates the type of line: direct connection (`Direct`), automatic calling unit (`ACU`), or a system or local area network (name of the system or LAN switch).

`Line` is the device name for the line (port) that is given in **/dev**.

`Line2` (used only for 801-type dialers) is the device name for the 801 dialer (in this case, `line` is the device name for the modem; `Line` must be a place-holder (**-**) for non-801 dialers.

`Class` can be one of three entries: the line speed alone, the line speed preceded by a one-letter code to distinguish between classes of dialers, or `Any` to match the speed indicated in the `Systems` file.

`Dialer,Token [Dialer,Token]`

can be one of four entries, depending on the type of connection:

- `direct` for a direct connection to a computer.
- The name of the modem (token is taken from the **Systems** file) for an automatic dialing modem (one pair only).
- The name of a LAN switch and a token (token may be omitted here and entered in the **Systems** file) for connection to a local area network.
- Two dialer-token pairs for an automatic dialing modem connected to a switch—one for the switch and one for the automatic dialing modem—with one of two escape sequences allowed for tokens:

  `\T` to request translation using the **Dialcodes** file

  `\D` not to request translation.

Any name(s) entered here must match the name in the **Dialers** file.

## 39.2 Software setup for **uucp**

In this section we'll discuss the **uucp** programs, files, and shell scripts required to operate the system, along with the procedures for identifying your system in your password file and identifying other systems for your system.

### The programs

The programs required to run the system are stored in two directories: one for the operational commands (**/usr/bin**) and one for the maintenance commands and daemons (**/usr/lib/uucp**). Here is a summary of the modules:

### /usr/bin

**cu** Command for calling another UNIX system

**ct** Command for calling a terminal (callback)

**uucp** Command for transferring files

**uuto** Command for sending files to /usr/spool/uucppublic/receive (complete directories only)

**uupick** Command for retrieving files from /usr/spool/uucppublic/receive

**uux** Command for remote execution of UNIX commands

**uustat** Command for checking network status

**uuname** Command for listing the names of systems in the network

**uname** Command for reporting the **uucp** name of the local system

**uulog** Command for printing a log of **uucp** actions

### /usr/lib/uucp

**uucico** Daemon invoked by **uucp**, **uuto**, or **uux** to interact with other systems (select the device for the link, establish the link, perform login, check permissions, transfer data, execute files, log the results, and notify the user by **mail**)

**uuxqt** Daemon invoked by shell script **uudemon.hour**, which is started by **cron**, to execute commands remotely; such commands must be of the form X.file in the spool directory

**uusched** Daemon invoked by shell script **uudemon.hour**, which is started by **cron**, to schedule work that has been queued in the spool directory by starting **uucico**

**uucleanup** Command invoked by shell script **uudemon.cleanup**, which is started by **cron**, to cleaning up the **uucp** spool directory

**Uutry** Command for testing call-processing and debugging, invoking **uucico** to set up a link to any computer specified

**uucheck** Command for checking for the existence of **uucp** directories, programs, and files and checking the Permissions file

Once the system is installed, each **uucp** or **uux** command places the user's request in a queue and invokes the **uucico** daemon. The local **uucico** calls the other system and performs any file transfer requested. The system invokes a remote **uucico** to receive the file at the other end. When you request remote execution of UNIX commands with **uux**, **uucico** transfers a command file to the other system. Then **uudemon.hour** invokes the **uuxqt** daemon to execute the command file and return any output to your system.

## THE FILES

The record-keeping files that result from operation of the system are stored in the spool directories: **/usr/spool/***). These files are used to store information about remote file transfers or command executions, lock devices, hold temporary data. Here is a summary of the files:

**/usr/spool**

C.*sysnxxxx* Work files are created in a spool directory whenever file transfers or remote command executions have been queued for a remote system. The suffixes are the name of the remote system (*sys*), the priority of the work (*n*), and sequence number (*xxxx*). Each file contains the following:

- Full pathname of the file to be sent or requested
- Full pathname of the destination or file name
- User ID
- Options
- Name of associated data file in the spool directory (D.0 if **uucp -c** or **uucp -p** was given)
- Mode bits of the source file
- ID of remote user to be notified upon completion

X.*sysnxxxx* Execute files are created in the spool directory before remote command executions take place. The suffixes are the name of the remote system (sys), the priority of the work (*n*), and sequence number (*xxxx*). Each file contains the following:

- Requestor's login ID and system name
- Name(s) of the files requested for execution
- Standard input for the command string
- System and file names for standard output
- Command string
- Option lines for return status requests

D.*sysnxxxxyyy*

Data files are created whenever the command line specifies copying the source file to the spool directory. The suffixes are the name of the remote system (*sys*), the priority of the work (*n*), sequence number (*xxxx*), and sequence number extension (*yyy*). An extension is appended whenever there is more than one data file (D) for a given work file (C).

LCK.*name* Lock files, created in the /usr/spool/locks directory for each device being used, prevent more than one user from accessing the same device at one time. The suffix .*name* is the name of either a system or a device.

TM.*pid.ddd* Temporary data files, created in /usr/spool/uucp/*name* whenever a file is received from another system, use the name of the other system (*name*). The suffixes are process ID (*pid*) and sequence number (*ddd*).
If the file transfer is aborted, the TM file will remain in its original directory; if file transfer is completed, the TM file will be moved to a directory specified by the file that initiated transmission (C.*sysnxxxx*). The cleanup program **uucleanup** removes TM files automatically.

## Identifying your own system

To allow users from other UNIX systems to log into yours, you have to provide them with a network name for your own system. Use **uname** to find out what this name is; then give the name to system administrators of other systems. Their users will have to use this name to log into your system, but their requests will arrive at your system via user name **nuucp**.

To provide the login function for **uucico**, you have to set up a user called **nuucp**. Specify /usr/spool/uucppublic as the home directory and /usr/lib/uucp/uucico as the login shell. Then, after you've entered this user line, use the **passwd** command to give **uucp** a password. This is how the line in /etc/passwd should look:

```
nuucp:Wqv7b@pZas:5:1:UUCP:/usr/spool/uucppublic:/usr/lib/uucp/uucico
```

You also need an identifier for the **uucp** administrator, the owner of all **uucp** spooled data files and programs. For this entry, make /usr/lib/uucp the home directory. Here is how the line in /etc/passwd should look:

```
uucp:Ty&u2*We;Ru:5:1:UUCP:/usr/lib/uucp:
```

## Identifying other systems

Now that you have your own system identified for remote users, you have to identify other systems for your system. This involves finding out the phone numbers, identifiers, passwords, data rates, and calling hours of the other systems and entering them in three files:

/usr/lib/uucp

Dialers Character strings necessary to establish connection with non-801 ACUs on other systems

**Systems** Information (name of computer and device, hours, telephone number, login ID, and password) needed by **uucico** and **cu** to establish a link to another system

**Dialcodes** Abbreviations that can by used in the telephone number field of the **Systems** file

More detailed descriptions of these files follow. We'll begin with the **Dialers** file. Each line entry in this file has the following general format:

```
dialer   substitutions   expect-send
```

where

`dialer` is a dialer that matches the fifth (or seventh) field in the **Devices** file.

`substitutions` is a set of translation pairs, usually used to translate = into "wait for dialtone" and - into "pause."

`expect-send` is a sequence of characters to be received and sent to establish connection.

Here are some of the character strings distributed in the **Dialers** file:

```
direct
micom ""     ""\s\c NAME? \D\r\c GO
develcon "" "" \pr\ps\c est:\007 \E\D\e \007
hayes    =,-, "" \dAT\r\c OK\r \EATDT\T\r\c CONNECT
ventel   =&-% "" \r\p\r\c $ <K\T%%\r>\c ONLINE!
```

Explanations of the escape sequences are as follows:

| | |
|---|---|
| `\p` | Pause about 1/2 second |
| `\d` | Delay about 2 seconds |
| `\T` *Dialcodes* | translation for phone number or token |
| `\D` | No translation |
| `\E` | Enable echo-checking |
| `\e` | Disable echo-checking |
| `\r` | Carriage return |
| `\n` | Send newline |
| `\c` | No carriage return or newline |
| `\K` | Insert a BREAK character |
| `\`*nnn* | Send octal number *nnn* |

Other escape characters are listed under the discussion of the **Systems** file. The entry for `hayes` in the **Dialers** file is executed as follows. First translate the phone number, replacing either a = (wait for dialtone) or a - (pause) with a comma (`,`). The rest is as follows:

`""` Wait for nothing.

`\dAT\r\c` Delay for 2 seconds, then send AT, followed by a carriage return, followed by no newline or carriage return.

`OK\r` Wait for OK, followed by a carriage return.

`\EATDT\T\r\c` Enable echo-checking, then send ATDT and translated phone number or token with **Dialcodes**, followed by carriage return, followed by no carriage return or newline.

`CONNECT` Wait for CONNECT.

The **Systems** file contains the information required by the **uucico** daemon to establish a link to another system. You can enter more than one line for the same system; **uucico** will treat them as alternate communications paths. Also, you can set up **remote.unknown** so that any computer not named in **Systems** will be unable to log into your system. Finally, you can use the System Administration menu (**systemmgmt** subcommand) to manage **Systems**. Each line entry in this file has the following general format:

```
Name     Time     Type     Class      Phone      Login
```

where

**Name** is the node name of the other system

`Time` indicates when the system send accept calls to the other system, using these conventions: `Any` (any day of the week); `Wk` (any week day); `Su`, `Mo`, `Tu`, `We`, `Th`, `Fr`, `Sa` (days of the week); `0000-2400` (time of day); `Never` (other system must initiate the call); `Any` for sending calls upon receipt (any day, any time).

You can also specify an optional minimum retry period by typing a semicolon immediately after the time, followed by a number to represent minutes.

`Type` is the device type to be used to establish the link (such as ACU); this field must match the Name field of a line in the `Devices` file; you can also add a protocol type here (for example, ACU,e).

`Class` is either the data rate for the device alone (for example, 1200) or the data rate preceded by a one-letter class code (for example, C1200) (or `Any` to allow any data rate).

`Phone` is the phone number (token) of the other system for automatic dialers (LAN switches). You can enter either actual phone numbers or abbreviations that you have stored in a file called **/usr/lib/uucp/L-dialcodes**.

`Login` describes the login sequence on the other system, using a series of prompts and responses.

You can use the following conventions in a login sequence when required:

| | | | |
|---|---|---|---|
| `\p` | Pause about 1/2 second | `\d` | Delay about 2 seconds |
| `\N` | Send or expect a null character | `\b` | Send or expect a backspace |
| `\s` | Send or expect a space character | `\t` | Send or expect a tab |
| `\E` | Enable echo-checking | `\e` | Disable echo-checking |
| `\r` | Send or expect a carriage return | `\\` | Send or expect backslash |
| `\c` | Suppress newline at end of string | `\K` | Same as BREAK |
| `BREAK` | Send or expect a BREAK character | `\K` | Same as BREAK |
| `\`*ddd* | Convert the octal number to a character | `EOT` | Send or expect EOT newline twice |

Here is an example of a line entry:

```
Name  Time  Type    Class  Phone     Login

colt  Any   ACU     1200   LA5213 "" gin:-BREAK-gin: nuucp word: gallop
```

Note that you can save space in the login sequence by abbreviating prompts (for example, `gin:` for `login:`, `word:` for `password:`)

If you choose to use mnemonic abbreviations instead of entire phone numbers, you can assign them in **Dialcodes**. Here is an example, followed by another line in **Systems** that invokes the abbreviation:

**/usr/lib/uucp/Dialcodes**

```
hal 818-347-9000
```

**/usr/lib/uucp/Systems**

```
gato  Any  2300-0800  ACU  1200  hal  login:---login: nuucp word: cat
```

Here `hal` is an abbreviation for 818-347-9000, as specified above. Note also the hours specified for incoming calls at this installation (11:00 p.m. to 8:00 a.m.). You can request a file transfer to this system with **uucp** or remote execution with **uux** any time of the day, but **uucico** will look up the time specified and defer the request until 11:00 pm. Speaking **uucico**, let's go on to the subject of setting up the files that control and maintain the system.

## 39.3 Control and maintenance of **uucp**

At this point you have the hardware and software in place and you've identified your system to others and other system to yours. Now we'll discuss the things you have to do to control and maintain your **uucp** system from day to day.

### Providing for startup functions

The **uucp** system records one line entry for every request submitted and every request processed. This usually amounts to hundreds or even thousands of lines per day. To keep this mountain of record-keeping from overwhelming you, it's a good idea to get rid of any leftover lock files each time you start up UNIX. The way to do this is to include this line in your /etc/rc file (this is shown in Chapter 32, "Startup and Shutdown," in the listing of /etc/rc):

```
rm -f /usr/locks/*
```

### Programming periodic functions

The shell scripts that schedule, process, and monitor jobs and remove old files are stored in the /usr/lib/uucp directory. These scripts are started by entries in the /usr/spool/cron/crontabs/root file. The scripts and the entries that start them are summarized here:

/usr/lib/uucp

**uudemon.poll** Checks the Poll file (/usr/lib/uucp/Poll) for systems scheduled to be polled. For each one it finds, it places a work file (C) in directory /usr/spool/uucp/*name*, where *name* is the name of the system.

**uudemon.hour** Calls the **uusched** program to search for and schedule any work files (C) waiting for transfer to another system; calls the **uuxqt** daemon to search for and execute any execute files (X) that have been transferred to your system for processing.

**uudemon.admin** Executes the **uustat -p -q** command, and sends the status information to the **uucp** administrative login: **-q** reports the status of all pending work files (C), execute files (X), and data files (D). **-p** prints information on processes listed in the lock files (/usr/spool/locks).

**uudemon.cleanup** Extracts log files for individual systems from the log directory **/usr/spool/uucp/.Log**, merges them, and appends them to the storage directory **/usr/spool/uucp/.Old**; removes execute files (**X**) more than a day old and work files (**C**) and data files (**D**) more than six days old; returns undeliverable mail to senders; mails to the **uucp** administrative login a summary of the day's status information.

**/usr/spool/cron/crontabs/root**

```
1,30     * * * * "/usr/lib/uucp/uudemon.poll > /dev/null
11,41    * * * * "/usr/lib/uucp/uudemon.hour > /dev/null
48 8,12,16 * * * /bin/su uucp -c "/usr/lib/uucp/uudemon.admin" > /dev/null
45 23      * * * ulimit 5000; /bin/su uucp -c "/usr/lib/uucp/uudemon.cleanup"
                   > /dev/null 2>&1
```

## Controlling user access

Now we'll discuss how you can control access to your system by your own users and by users of other systems. This involves setting up a file that describes user access (**/usr/lib/uucp/Permissions**). This file allows you to control user access by specifying permissions for remote systems that relate to login, file access, and command execution. Each line in this file is entered in the following format:

```
option=value option=value option=value ...
```

Use spaces to delimit individual items on a line, a backslash (\) to continue to the next line, and a pound sign (#) to enter a comment line. Blank lines are ignored. You can enter items either for incoming calls (LOGNAME) or for outgoing calls (MACHINE). Login IDs used on incoming calls must appear in one and only one LOGNAME entry. If you make an outgoing call to a system not named in a MACHINE entry, the default permissions are as follows:

1. Local requests to send or receive data will be executed.
2. The incoming spool file will be your **/usr/spool/uucppublic** directory.
3. A remote system can send only default commands (such as **rmail**) to your system to be executed.

The specific options, the permission issues they relate to, and where they may be used (LOGNAME or MACHINE) are as follows:

`REQUEST` Can one system request files from another? Yes or no (LOGNAME or MACHINE)

`SENDFILES` Can another system take work that your system has queued for it after completion on your system? Yes, no, or call (LOGNAME)

`READ/NOREAD` Which directories on one system can/cannot be read by another? (LOGNAME or MACHINE)

`WRITE/NOWRITE` Which directories on one system can be written to by another? (LOGNAME or MACHINE)

`CALLBACK` When another system calls, does your system have to hang up and call back to confirm its identity? Yes or no (LOGNAME)

`COMMANDS` Which commands is another system allowed to execute on your system? (MACHINE)

`VALIDATE` Which remote systems are granted a login ID/password combination to call your system? (LOGNAME)

Here are some examples of entries with explanations:

```
LOGNAME=tiger:panther:cheetah REQUEST=yes SENDFILES=no
```

Any time one of the machines named logs into your system, it has permission to request files from your system, but not to take output from processes run on your system. (You can use `SENDFILES=call` to allow sending of files only when your system initiates the call.)

```
MACHINE=daisy:rose:tulip \
READ=/usr/spool/uucppublic WRITE=/usr/spool/uucppublic
```

Any time your system logs into one of the systems named, read and write permission are granted only for the **uucp** spool file (the default). (You can use `READ=/ WRITE=/` to allow read and write permission for all directories.)

```
LOGNAME=tiger:panther:cheetah \
READ=/etc NOREAD=/etc/bin WRITE=/usr/spool/uucppublic:/usr/jungle
```

Any time one of the machines named logs into your system, it has permission to read any files in /etc except those in /etc/bin and it has permission to write to the **uucp** spool file and /usr/jungle.

```
LOGNAME=tiger:panther:cheetah \
CALLBACK=yes
```

Any time one of the machines named logs into your system, your system hangs up and calls the system back to confirm its identity. Note that if two systems use this option, neither can log into the other.

```
MACHINE=daisy:rose:tulip \
COMMANDS=/usr/bin/news
```

Any time your system logs into one of the systems named, you grant to that system permission to execute **/usr/bin/news** on your system. (Use this option with great care.)

```
LOGNAME=bigcat VALIDATE=tiger:panther:cheetah
```

Any time one of the machines named logs into your system, it must log in as `bigcat`. This offers protection only if this ID is kept secret.

```
MACHINE=daisy:rose:tulip REQUEST=yes READ=/ WRITE=/ \
COMMANDS=/usr/bin/news

LOGNAME=flower VALIDATE=daisy:rose:tulip READ=/ WRITE=/ \
REQUEST=yes SENDFILES=yes
```

This combination provides nearly unlimited read, write, and command execution for the systems named. The only safety precautions here are the `COMMANDS` option, which restricts the systems to **news**, and the `VALIDATE` option, which requires a specific login ID of each of these systems when it calls your system.

## Requesting polling

The **Poll** file allows you to poll other systems at regular intervals. Each line in this file must have the name of a system, a (TAB) (not a space), and one or more times to attempt a call. Here is the general format:

```
Name  (TAB)  hour [hour ...]
```

For example, here is a line entry to provide for polling of system **rabbit** at 8:00 A.M., 1:00 P.M., and 7:00 P.M.:

```
rabbit   8 13 19
```

The **uudemon.poll** script will set up a polling work file called **C.file** in the spool directory. The **uudemon.hour** script will start the **uusched** daemon every hour, which will perform the polls at the hours indicated.

## Using custom files

If you would like to have different **Devices**, **Dialers**, and **Systems** files for different kinds of calls with **cu** and **uucp**, you can identify them in the **Sysfiles** file. Here is the general format of an entry in the file:

```
service=program(s)  systems=file(s)  dialers=file(s)  devices=file(s)
```

where

*program(s)* is either **cu** or **uucico** (or both).
*file(s)* is one or more filenames to be consulted by the program named.

For example, here is how you could maintain separate systems lists for **cu** and **uucico** (in addition to the standard list):

```
service=cu      systems=Systems:cu_Systems
service=uucico  systems=Systems:uu_Systems
```

Both programs will begin by searching **/usr/lib/uucp/Systems**. Then, if the desired system is not found, **cu** will then search **/usr/lib/uucp/cu_Systems**, while **uucico** will search **/usr/lib/uucp/cu_Systems**. You can print these lists using either the **uuname** command (**uucico**) or **uuname -c** (**cu**).

## Specifying STREAMS modules

If your system uses a STREAMS-based transport provider that conforms to the Transport Layer Interface (TLI), you can specify the STREAMS modules to be used for a particular TLI device in **/usr/lib/uucp/Devconfig**. The general format is as follows:

```
        ⎧cu     ⎫
service=⎨       ⎬ device=network push=module[:module ...]
        ⎩uucico ⎭
```

where

*network* is the name of a TLI network.
*module* is the name of a STREAMS module.

Use **ed** or **vi** to change the contents of **Devconfig**. For details, refer to AT&T's *STREAMS Primer*[1] and *STREAMS Programmer's Guide*[2].

[1] *STREAMS Primer*, [location not given]: AT&T, 1986.

[2] *STREAMS Programmer's Guide*, [location not given]: AT&T, 1986.

### Other files

Other files that perform various functions are as follows:

1. **Maxuuxqts** defines the maximum number of **uuxqt** programs that can run at the same time.
2. **Maxuuscheds** defines the maximum number of **uusched** programs that can run at the same time.
3. **remote.unknown** keeps track of login attempts from unknown systems, prohibiting connection and logging a record of the attempts.

You can use **ed** or **vi** to make changes to these files.

### Watching files that grow

Here are two files that continue to grow while communications are active. Feel free to delete them when they become too large:

| | |
|---|---|
| /usr/adm/sulog | (super-user commands) |
| /usr/lib/cron/log | (**cron** activities) |

## 39.4 Trouble-shooting **uucp**

Once you have **uucp** installed, you may feel like breathing a big sigh of relief. But don't breathe too deeply because your problems have just begun. (Just kidding!) In this section we'll discuss a few of the most common difficulties you may have to deal with.

### Getting information

To get a few simple facts about the network, you can use these commands:

| | |
|---|---|
| **uuname** | to list the systems that your system can call |
| **uname** | to report the **uucp** name of your own system |
| **uulog** | to display the log directories of certain host systems |
| **uucheck -v** | to determine files, directories, and permissions for **uucp** |

Other information can be found in the following directories:

| | |
|---|---|
| /usr/spool/uucp/.Status/* | status information on attempted communication to other systems, stored in files named after the systems |
| /usr/spool/uucp/.Admin/errors | ASSERT error messages, which provide file name, SCCS ID, line number, and text |

Finally, make sure that your **Systems** file contains the correct phone number, login, and password for each system entered.

## Defective equipment

You can check for defective automatic call units (ACUs) or modems by running one of the following programs:

1. **uustat -q** to give reasons for any failures detected
2. **cu -d -l***line* to call over a line and print a report on the results (no autodialer)
3. **cu -d -l***line phone number* to call over a line and print a report on the results (autodialer)

A line used by **cu** here must be defined in the **Devices** file as Direct.

## Losing contact

If you are unable to reach another system (say **cougar**), you can begin by trying to make contact as follows:

1. Try to make a connection:

   □ Execute the following command from your system:

   ```
   # /usr/lib/uucp/Uutry -r cougar
   ```

   □ This command will start the transfer daemon (**uucico**) with debugging directed to **/tmp/cougar**.

   □ Send the output to your terminal, which you can halt by pressing (BREAK).

If this doesn't work, you can also check out the other system by invoking its **uucico** daemon from your system. Here's the procedure:

2. Queue a job on the other system:

   □ Execute the following command from your system:

   ```
   # uucp -r test.file cougar!~/phil
   ```

   □ This command will cause the other system to queue the job without invoking **uucico** to process the job.

3. Try to make connection again:

   □ Execute the following command from your system:

   ```
   # /usr/lib/uucp/Uutry -r cougar
   ```

   □ Save any output you receive to show to service personnel.

## 39.5 Summary

In this chapter you learned the procedures for setting up and operating the **uucp** system, which allows you to exchange files with other UNIX systems and execute commands on those systems.

### Hardware requirements for **uucp**

The basic hardware requirements are at least two UNIX machines connected by 1) a direct (hard-wired) connection; 2) an auto-dial modem attached to a terminal; or 3) a local area network (LAN). (Larger systems may employ an ACU in place of a modem.) You usually have to disable the terminal in /etc/inittab, then place a line entry in /usr/lib/uucp/Devices.

### Software setup for **uucp**

The **uucp** system includes 15 basic modules, nine in /usr/bin (**cu**, **ct**, **uucp**, **uuto**, **uupick**, **uux**, **uustat**, **uuname**, **uname**, **uulog**) and six in /usr/lib/uucp (**uucico**, **uuxqt**, **uusched**, **uucleanup**, **uutry**, **uucheck**). The two primary commands for ordinary users are **uucp** (file transfer) and **uux** (remote execution). These programs invoke two daemons, **uucico** (calling other systems) and **uuqxt** (executing remote commands).

There are five types of files used for record-keeping: work (C), execute (X), data (D), lock (LCK), and temporary (TM) files. The shell scripts **uudemon.poll**, **uudemon.hour**, **uudemon.admin**, and **uudemon.cleanup** are started by **cron** to take care of day-to-day administrative functions.

You have two tasks to perform here: 1) identifying your system to others by placing a line entry in /etc/passwd (plus another entry for the administrator); 2) identifying other systems to yours by placing a series of line entries in /usr/lib/uucp/Dialers and /usr/lib/uucp/Systems. You can set up abbreviations for phone numbers in /usr/lib/uucp/Dialcodes if you like.

### Control and maintenance of **uucp**

To keep your **uucp** system running smoothly, processing new jobs and getting rid of old jobs regularly, you need to add lines to key files. Your /etc/rc file should contain lines to remove lock files; your **uudemon.cleanup** file should take care of removing extra files.

To control user access to your system, you need to add lines to a file called /usr/lib/uucp/Permissions. Each line indicates which users and systems can and cannot access which files; it can also require a callback to verify the caller's identity.

Here are a few more functions available: you can request polling of another system with the /usr/lib/uucp/Poll file; you can specify STREAMS modules to be used for a TLI device with /usr/lib/uucp/Devconfig; and you can change the maximum number of **uuxqt** programs (`Maxuuxqts`), the maximum number of **uusched** programs (`Maxuuscheds`), or the method of handling calls from unknown systems (`remote.unknown`).

## TROUBLE-SHOOTING **uucp**

This section concludes with a few suggestions for problem-solving. You can use **uuname**, **uulog**, and **uucheck -v** to obtain basic information about your network, and you can find error messages in /usr/spool/uucp/.Status and /usr/spool/uucp/.Admin/errors.

To check for defective equipment, you can run **uustat -q**, then **cu -d**, then **uustat -q** again if necessary.

You can handle the problem of losing contact with other systems by running **Uutry**, then queueing a job on the system and running Ûutry again.

# 40

# Basic Resource Sharing

## 40.1 Sharing resources

### An example

Suppose you work in an office that is organized as follows:

- The databases that are used by everyone are stored on a mainframe, which has two separate disk drives for backing up daily work.
- Text and graphics are handled by a minicomputer that is connected to a laser printer and a plotter.
- Statistical work is handled by another minicomputer, which also requires a laser printer and plotter.
- Each employee uses a personal computer for preparing reports, accessing information on the mainframe, and working with either text and graphics or statistics.

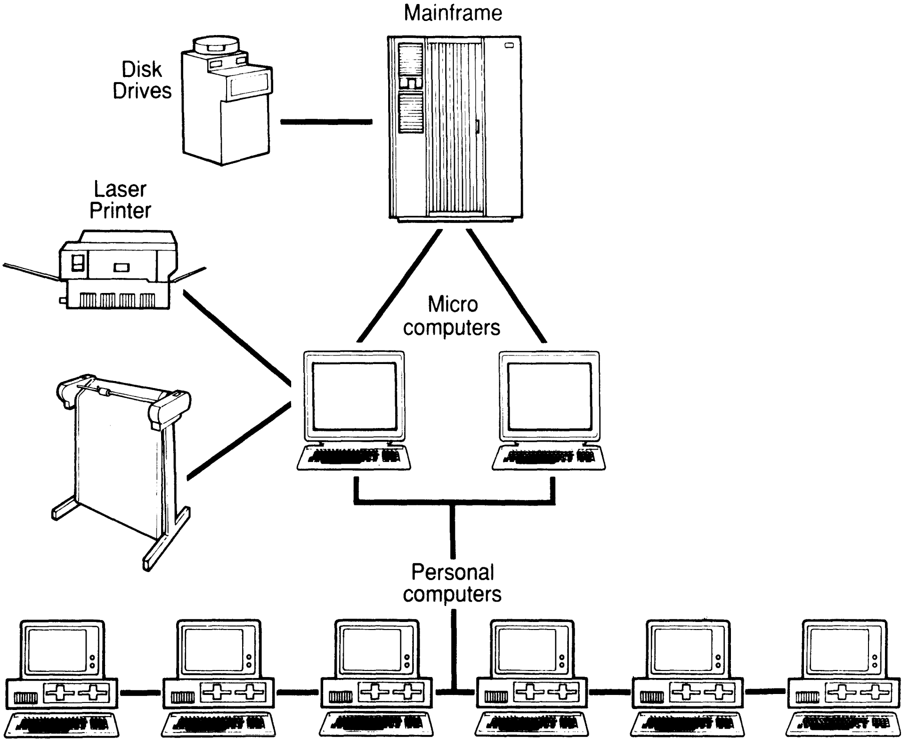
For this office to operate as smoothly as possible, it will be necessary to connect all these machines in a common network. This will allow employees to access information on each of the larger machines and send information back and forth to each other. It will also reduce the total amount of equipment required for this office. For example, with the two minicomputers linked, it won't be necessary for each to have its own laser printer and plotter. The network could look like that shown in Figure 40.1.

UNIX now allows ordinary users to take full advantage of a network like this without having to learn any new commands. To a user, it's as if the entire network has become a single UNIX system. Note that the machines do not necessarily have to be located near each other; as long as there is a network in place, the machines can be located on different continents. (If you'd like to review the basic concepts of networking, see Chapter 37, "Introduction to Networking").

### The remote file sharing system

System V, Release 3 introduces the Remote File Sharing (RFS) system, which allows users within a network of UNIX systems to access resources

FIGURE 40.1. An office network.



on other machines. With RFS, administrators of systems on the network can make known to others (*advertise*) which resources are available to be shared. Such resources can include files, directories, special files (devices), and fifo files (named pipes). Then other administrators can mount these resources on their own machines (*mount*) as if they were local resources.

As a system administrator offering resources, you can restrict access to these resources (and thereby maintain security) in three different ways:

1. Decide which remote systems are allowed to connect to your system.

2. Decide which remote systems are allowed to mount resources.

3. Set up user and group permissions using an optional mapping scheme that maps remote users and groups to local users and groups.

Likewise, administrators of other systems will be allowed to restrict access to their resources by the same methods. The system administrator of each machine will set up appropriate security measures, and decide which resources may and which resources may not be shared with other machines. (See Chapter 40, "RFS Security," for details.)

Once the security issues have been resolved and file systems have been mounted in this way, local users will be able to access remote files as if they were local files. RFS has been implemented in such a way that users are not even aware that they are accessing remote files. Since special files can be remotely mounted, networked UNIX systems can share common devices like disk drives and laser printers.

## The concept of resource sharing

When UNIX systems are grouped together to share resources with each other, the group is called a *domain* and each individual system or machine is called a *host*. To set up the RFS system initially, you have to assign to each domain at least one host (*name server*) to keep track of the names of resources being shared.

After these and a few other details have been taken care of, any host can make known the resources that it is willing to share by *advertising* them—much the same way someone might advertise an item for sale on a bulletin board. Other hosts can look at the list of resources available (maintained by the name server) and *mount* those they have permission to use.

Once a resource has been advertised by one host and mounted by another, it becomes available to users of the second machine via a *virtual circuit*. This is depicted in Figures 40.2, 40.3, and 40.4, which show two systems in the same domain before and after sharing resources, with the commands required for sharing in the middle. In this example, a host called sales shares resources with another host called mrktg.

FIGURE 40.2. Two hosts before sharing.

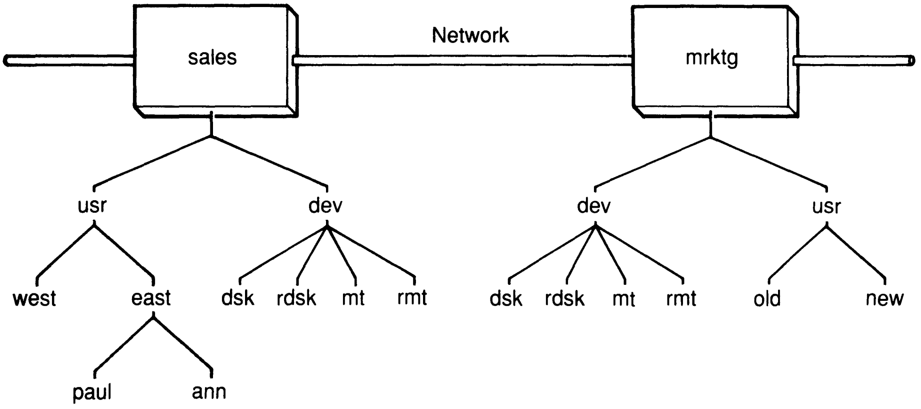
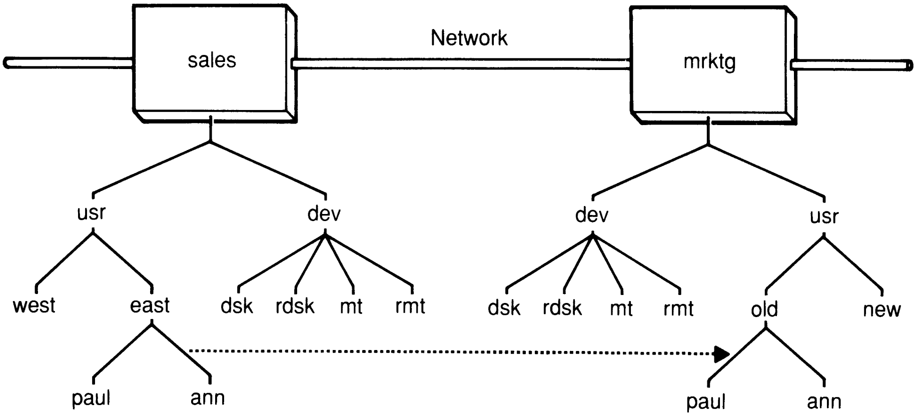


FIGURE 40.3. Commands required for sharing.

```
             sales                          mrktg
# adv STORE /dev/mt        # mount -d STORE /dev/mt
# adv SELL /usr/east       # mount -d SELL /usr/old
```

FIGURE 40.4. The Hosts after sharing.



## 40.2 Setup procedures

### Preliminary information

An RFS network is organized into *domains*. Every machine must belong to a domain, a grouping that includes at least one machine and can include all machines on a network. A machine that keeps track of the names of machines in a domain is called a *domain name server*. Each domain must have one *primary* name server and may have optional *secondary* name servers for backup. These are identified in a configuration file called **rfmaster**.

Each domain and each machine must have a name. One machine may refer to another in the same domain by its machine name only (unqualified name); to refer to a machine outside its own domain, a machine must name both the domain and the machine (fully qualified name). This is similar to the naming conventions for files and directories.

### Setting up a domain

Once you've chosen which host(s) in the network you'd like to designate as domain name server(s), installed all software (UNIX, RFS, and network), and logged in on the primary name server as **root**, you're ready to begin setting up your domain. This involves setting your system's node name, setting up a network listener, setting your domain's name, identifying the network, identifying the name servers, adding individual hosts, and starting RFS:

1. Set your machine's node name:

□ First see if your machine already has a node name by entering

```
# uname -n
```

- □ If not, give it a node name (say **sales**) by entering

  ```
  # uname -S sales
  ```

  Your name can up up to 14 characters long, including upper or lower case letters, digits, hyphens (-), and underscores (_).

2. Set up the network listener (AT&T Starlan network assumed):

- □ Initialize the files needed for the listener process by entering

  ```
  # nlsadmin -i starlan
  ```

- □ Report the status of the listener process on this machine:

  ```
  # nlsadmin -x
  starlan ACTIVE
  ```

- □ Identify the network addresses of your system for the listener:

  ```
  # nlsadmin -l sales.serve -t sales starlan
  ```

- □ Start the listener:

  ```
  # nlsadmin -s starlan
  ```

3. Set the domain name:

- □ Enter your domain name (say **ace**) by entering

  ```
  # dname -D ace
  ```

- □ The rules for domain name are the same as those for node name.

4. Identify the network:

- □ Identify the network (transport provider) to RFS by entering

  ```
  # dname -N starlan
  ```

- □ This names the device in **/dev** that will be used for the transport provider.

5. Identify the domain name server(s) in the configuration file:

- □ Move to directory **/usr/nserve** and create a file called **rfmaster** with a text editor like **ed** or **vi**.

□ Identify the primary name server for the domain (say **mrktg**), any secondary name server(s) desired (say **eng**), and then enter the network address of each server machine named:

```
# cat rfmaster
ace          p   ace.mrktg     [Primary name server]
ace          s   ace.eng       [Secondary name server]
ace.mrktg    a   mrktg.serve   [Network address of primary]
ace.eng      a   eng.serve     [Network address of secondary]
# _
```

□ For each domain with which you intend to share resources, you must supply the same information (primary and secondary name servers and network addresses).

6. Add individual hosts:

□ For each host system that will belong to the domain, enter something like

```
# rfadmin -a ace.mrktg    [Add mrktg to ace]
Enter password for mrktg:
Re-enter password for mrktg:
```

□ The password will be stored in **/usr/nserve/auth.info/ace/passwd**.

7. Start RFS:

□ Start the Remote File Sharing system manually by entering

```
# rfstart
rfstart: Please enter machine password:
```

□ Enter the password for your host that you entered in Step 6.

Step 7 is usually performed automatically by the **init 3** command, which initiates RFS, runs **rfstart**, advertises your machine's resources to others, and mounts remote resources on your machine through a series of shell scripts. For details, see Chapter 39, RFS Maintenance."

## Setting up each host

The previous section described the steps required to set up a domain from the primary name server. To set up each individual host in a domain, follow the same steps except for Steps 5 and 6.

After you've completed these steps for each host in the domain, you will have the domain set up under the default security measures for resource sharing—in addition to the usual UNIX security measures. Any users from other hosts who use resources from your system will be mapped into a special guest login with minimal access to files and directories that belong to your users. If you would like to set up more stringent security measures, you have the option of restricting access to certain domains or hosts with the *mapping* feature. For details, refer to Chapter 40, "RFS Security."

Both the system that has resources to share with others (the [file] *server*) and the system that needs them (the *client*) must issue commands that name the resources to be shared. The procedures are described in detail in the remainder of this chapter.

## 40.3 Advertising resources

### A simple example

A server advertises the availability of resources to other UNIX systems on the network with a command called **adv** (advertise); then any client that has not been restricted can mount them with a new option for the **mount** command. In general, the **adv** command assigns a resource name to a directory that you are willing to share and adds the resource to a list kept by the domain name server. For example, here is a command that assigns the name CUSTOMERS to a directory called /usr/admin/cust.list:

```
# adv CUSTOMERS /usr/admin/cust.list
```

Now this directory and all files and directories under it in the file system are listed as available for sharing. Resource names, which are customarily typed in all capitals for ease of recognition, may include up to any 14 printable characters other than slash (/), period (.), or blank space (). No name may be repeated within a domain.

### Options for **adv**

In addition, the **adv** command offers you three options: 1) making the resource read-only; 2) including a short description of the resource; and 3) restricting certain domains or hosts from accessing the resource. Here is the general form of the **adv** command:

```
# adv  [-r] [-d " description "]  resource pathname [ client(s)]
```

where

**-r** makes the resource read-only (read/write is the default).

**-d** "*description*" provides an optional 32-character description of the resource.

| | |
|---|---|
| *resource* | is the name of the resource that you choose (up to 14 characters)—typically entered in upper case letters for ease of reading. |
| *pathname* | is the full pathname of a directory in your system. |
| *client(s)* | is an optional list of domains or hosts to whom you wish to restrict access to the resource. |

## Examples of **adv** commands

Here are some examples of **adv** commands:

```
# adv -r -d "March correspondence" LETTERS /usr/letters/march
```

Advertise local directory /usr/letters/march read-only under the name LETTERS, with the description "March correspondence".

```
# adv -d "Z15 programs" PROGS /usr/bin/z15 anthony series.quotas mrktg
```

Advertise local directory /usr/bin/z15 under the name PROGS, with the description "Z15 programs" and restriction to all systems in domain anthony, system quotas in domain series, and system mrktg in your own domain. (To name a system in another domain, you must give a fully qualified name of the form domain.host.)

## Using aliases for clients

You can save time by grouping names in the *Client(s)* field under an alias; then you can enter the alias in place of the other names. Just open the file called /etc/host.alias and enter each alias in the format shown here:

```
   alias  name client1 client2 client3 ...
```

where *name* is the name of the alias and the client names can be names of domain names, host names, or aliases previously defined. In the following example

```
   alias team series eng mrktg
   alias chiefs century acctg
   alias all chiefs team
```

**team** includes all systems in domain series, along with systems eng and mrktg in your own domain; **chiefs** includes all systems in century, along with system acctg in your own domain; and **all** includes all of the above.

## MODIFYING AN **adv** ENTRY

If you'd like to modify what you've already entered in a previous **adv** command, use one of these forms, allowing you to change either the description or the list of clients or both:

```
# adv  -m   resource   -d " description"   [ client(s)]
# adv  -m   resource  [-d " description"]    client(s)
```

For example, you could use this to change the description for **LETTERS**:

```
# adv -m LETTERS -d "Letters for the month of March"
```

## ADVERTISING AUTOMATICALLY

If you'd like to have certain **adv** commands executed automatically, place them in the file **/etc/rstab**. Then, whenever your system enters Remote File Sharing mode (**init 3**), these **adv** commands will be run.

## DISPLAYING RESOURCES ADVERTISED BY YOUR SYSTEM

To display resources on your system that have been advertised for sharing with other systems, execute the **adv** command by itself, like this:

```
$ adv

LETTERS /usr/letters/march read-only "March correspondence" unrestricted

PROGS   /usr/bin/z15   read/write "Z15 programs" anthony series.quotas mrktg

$ _
```

## DISPLAYING RESOURCES ADVERTISED IN YOUR DOMAIN

To display all resources in your domain that have been advertised for sharing with other systems, execute the **nsquery** command:

```
# nsquery
RESOURCE        ACCESS          SERVER              DESCRIPTION

LETTERS         read-only       ace.training        "March correspondence"
PROGS           read/write      ace.training        "Z15 programs"
# _
```

For each resource available, **nsquery** provides the name of the resource (*resource*), permissions (*ACCESS*), the server host (*SERVER*), and the description of the resource (*DESCRIPTION*). If you'd like to omit the headings, you can include the **-h** option. If you'd like to restrict the display to a particular domain or host, you can include an optional name. Here is the general form of the **nsquery** command:

```
# nsquery [-h] [ name]
```

where *name* can be in one of three forms:

| | |
|---|---|
| *domain.* | the name of a domain |
| *domain.host* | the name of a host in a domain |
| *host* | the name of a host in your own domain |

### Unadvertising a resource

There are time when you may want to prevent additional users on other systems from sharing a resource that you have already advertised. For example, you may want to return a shared resource to local use, or you may want to unmount a file system in which a shared resource resides. Then any system in the domain can issue a command of the form

```
# unadv  resource
```

and the domain name server can issue a command of the form

```
# unadv  domain.resource
```

Here are examples:

```
# unadv LETTERS            [Issued by any system in the domain]

# unadv ace.LETTERS        [Issued by the domain name server]
```

## 40.4 Mounting resources

Once a resource from one host has been advertised, a system administrator from any other host with access to the resource can make it available to the users of that host by mounting it on that file system. Once you've decided on a mount point for the resource in your own file system, use the **mount** command with the **-d** option. If your host has permission, the resource will be added to your local mount table at the point that you've selected. For example, suppose /usr/bin/zebra is an empty directory. Then to mount resource PROGS at /usr/bin/zebra, use this command:

```
# mount -d PROGS /usr/bin/zebra
```

## Read-only option

You can mount any resource as read-only by including the **-r** option. If a resource has been advertised as read-only, however, you are required to mount it as read-only also. For example, suppose you locate a read-only resource in the **nsquery** list (say **LETTERS**):

```
# nsquery
resource       ACCESS        SERVER              DESCRIPTION

LETTERS        read-only     ace.training        "March correspondence"
PROGS          read/write    ace.training        "Z15 programs"
# _
```

If you would like access to **LETTERS**, you have no choice but to mount is as read-only, like this:

```
    # mount -r -d LETTERS /usr/admin/corr
```

The general format of the command for mounting remote resources is

```
    # mount [-r] -d resource mount_point
```

You can mount a remote resource on any *mount_point* in your UNIX system, with the following exceptions:

1. To avoid covering up local information vital to the functioning of your own UNIX system, never use any of the following directories as mount points: /, **/usr**, **/usr/net**, **/usr/nserve**, **/dev**, **/etc**.
2. To avoid conflicting file names, never use the following: **/usr/spool**.
3. You can access a device's file system only if the server has mounted the device locally and advertised the device's mount point.

## Mounting resources automatically

You can arrange to have remote resources mounted automatically by placing one line entry for each resource to be mounted in a file called **/etc/fstab** (which also contains lines for local mounting). Then your system will perform the mounts when it enters RFS mode (**init 3**). The following are the lines you would enter to perform the two mounts described above:

**/etc/fstab**

```
PROGS /usr/bin/zebra -d
LETTERS /usr/admin/corr -dr
```

The general format of a line in **/etc/fstab** for mounting a remote resource automatically is

```
    resource  mount_point -d[r]
```

## Displaying resources mounted on your system

To display remote resources that you have mounted on your own system, execute the **mount** command by itself. You will see local and remote resources in the same display, with remote resources identified by the word `remote` immediately following the permission:

```
$ mount
/ on /dev/dsk/c1d0s0 read/write on Fri Jan 18 08:43:17 1987
/usr on /dev/dsk/c1d1s0 read/write on Mon Jan 21 16:39:04 1987
/usr/bin/zebra on PROGS read/write/remote on Tue Mar 23 08:57:26 1987
/usr/admin/corr on LETTERS read only/remote on Wed Mar 24 15:04:33 1987
$ _
```

The general format of each line of display is

*directory* on *resource* *permission* on *date*

## Displaying resources mounted in your domain

To display all resources on your system that have been mounted on other systems, execute the **rmntstat** command:

```
# rmntstat
RESOURCE        PATHNAME          HOSTNAME
CUSTOMERS       /usr/adm/cust     series.operations mrktg
INVOICES        /usr/admin/inv    pub_rel mrktg eng
M_TAPE          /dev/rmt/0        field_s acctg
B_DISK          unknown           united field_s
# _
```

For each resource available, **rmntstat** provides the name of the resource (*RESOURCE*), the local pathname of the resource (*PATHNAME*), and the name of each host that has mounted the resource (*HOSTNAME*). (The pathname will be `unknown` any time you unmount a resource that is still in use by at least one other host. If you'd like to omit the headings, you can include the **-h** option. If you'd like to restrict the display to a particular resource, you can include an optional name. Here is the general form of the **rmntstat** command:

```
# rmntstat [-h] [ resource]
```

## Unmounting a resource

Unmounting a remote resource is similar to unmounting a local one. The only difference is the **-d** option. For example, to unmount resource **LETTERS**, execute the following sequence of commands:

```
# fuser -u LETTERS [Report any processes using files in LETTERS]
# umount -d LETTERS[Unmount resource LETTERS]
```

The general forms of these commands are as follows:

```
# fuser [-ku] resource
# umount -d resource
```

where the **-k** option of **fuser** allows you to kill any process that has files open in any directory or subdirectory of the resource.

### Forcibly unmounting a resource

It may become urgent to unmount a local resource that is being shared with other systems. You may not have time to notify other system administrators to unmount the resource from their systems. In such an emergency situation, you can forcibly unmount the resource from other systems by executing the **fumount** command. (Then you can proceed to unmount it from your own system with the ordinary **umount** command.)

```
# fuser -u LETTERS    [Report any processes using files in LETTERS]
# fumount -w 30 LETTERS
                      [Forcibly unmount LETTERS in thirty seconds]
```

The general forms of these commands are as follows:

```
# fuser [-ku] resource
# fumount [-w sec] resource
```

where the **-w** option of **fumount** provides for a grace period (in seconds) before proceeding with the forcible unmounts. If you include this option, all users will see messages like these at their terminals:

```
resource is being removed from the system in ss seconds.
resource has been disconnected from the system.
```

These messages are displayed by shell script **rfuadmin**, which is invoked by the daemon **rfudaemon** when your host receives a **fumount** command from the host that is forcing the unmount. (See "Recovery Procedures," page 000, in Chapter 39, "RFS Maintenance," for details.)

## 40.5 Summary

This chapter introduces the concepts and terminology of resource sharing, describes the steps for setting up a domain and individual hosts, and explains the procedures for advertising and mounting resources.

## Concepts and terminology

System V, Release 3 allows different UNIX systems to share resources (directories, files, devices, and named pipes). A group of UNIX systems (also referred to as *machines* or *hosts*) that are to share resources is called a *domain.* A host that keeps track of the names of resources being shared is called a *name server.* A *primary* name server is required; *secondary* name servers may be added for backup.

Making known the availability of a resource for sharing is called *advertising.* After one host advertises a resource and another host mounts it, the users of the second host may access the resource as if it were stored on their own system, subject to various security measures.

## Setup procedures

To set up a domain from the primary name server, set your host's node name, the network listener, and the domain name; identify the network and the domain name server(s); add individual hosts; and start the Remote File Sharing system.

To set up an individual host, set your host's node name, set up the network listener, set the domain name, identify the network, and start the Remote File Sharing system.

## Advertising resources

To advertise a resource on your host, use the **adv** command, giving a name to the resource and the pathname of the resource. Options include read-only protection, a description of the resource, adding a list of clients to whom you wish to restrict access, and modifying an entry. To have your **adv** commands executed automatically, place them in the file /etc/rstab. You can display resources advertised by your host (**adv** without arguments) or advertised in a domain (**nsquery**). Finally, you can unadvertise a resource with the **unadv** command.

## Mounting resources

To mount a resource that has been advertised, use the **mount -d** command, naming the resource and giving it a mount-point (an empty directory) within your own system. You have the option of mounting the resource as read-only. To have your **mount** commands executed automatically, place them in the file /etc/fstab. You can display resources mounted on your host (**mount** without arguments) or mounted in your domain (**rmntstat**). Finally, you can unmount a resource with the **umount** command (**fumount** for forcible unmounting).

# 41

# Remote File Sharing Maintenance

## 41.1 Introduction

In Chapter 40, "Basic Resource Sharing," you learned what the Remote File Sharing system is, how to set it up, and how to make it work. In this chapter you will learn the day-to-day tasks necessary to keep the RFS system running smoothly. These include starting and stopping run level 3, maintaining domains, maintaining individual hosts, and monitoring and adjusting performance:

1. *Run level 3*—starting and stopping
2. *Maintaining domains*—adding a new host, removing a host, resetting the current name server, adding new domains, and recovering from failures
3. *Maintaining hosts*—starting up and shutting down the RFS system
4. *Monitoring*—listing CPU time, number of system calls, processes, data transferred, and disk space
5. *Tuning*—eight adjustable maximum and minimum values that affect RFS performance

Most of the procedures that relate to the security of the Remote File Sharing system will be described in Chapter 40, "RFS Security."

## 41.2 Remote file sharing mode

### Run level 3

In Chapter 32, "Startup and Shutdown," you learned about single-user mode (run level 1) and multi-user mode (run level 2), initiated by the **init s** and **init 2** commands. A new run level has been defined for Remote File Sharing (run level 3). Whenever you enter run level 3 with the **init 3** command, the system starts all processes in the /etc/inittab file that show level 3. When you leave run level 3 with either **shutdown** or **init 2**, the system will stop all level 3 processes.

If you would like to have Remote File Sharing started automatically when you boot your system, you can change the run level for **initdefault** from 2 to 3, as shown here:

```
is:3:initdefault:
```

## Starting run level 3

Whenever you execute the **init 3** command, the system executes shell script /etc/rc3 and any other processes that show run level 3. Then /etc/rc3 executes all shell scripts in the run level 3 directory /etc/rc3.d whose names begin with S. This includes the RFS file S21rfs and any other files that you choose to place in /etc/rc3.d. The RFS file S21rfs, which is linked to /etc/init.d/rfs, performs the following tasks:

- Makes sure that a domain name has been set for your host
- Makes sure that the rfmaster file exists
- Executes **rfstart** every sixty seconds until RFS is started successfully
- Executes /etc/init.d/adv to advertise the resources named in your /etc/rstab file
- Executes /etc/rmountall to mount the resources named in your /etc/fstab file

You can add to the run level 3 directory /etc/rc3.d your own files to be linked to custom shell scripts stored in /etc/init.d. Begin the name of the /etc/rc3.d file with S (start), then include a two-digit sequence number, and lastly include the name of the linked file in /etc/init.d. Here is the format, with spaces added for ease of reading:

```
S  nn  name
```

For example, suppose you'd like to place shell scripts called **advertise** and **mount_new** in /etc/init.d. To make these the first and second scripts executed, use `01` and `02` as the sequence numbers, use the shell **start** option, and name the files as shown here:

| /etc/rc3.d | /etc/init.d |
|---|---|
| S01advertise | advertise |
| S02mount_new | mount_new |

### Stopping run level 3

Whenever you leave Remote File Sharing mode by executing an **init 0**, **init 1**, or **init 2** command, the system executes any processes that show run level 0, 1, or 2. Then the system executes all shell scripts in the run level 0 (/etc/rc0.d), run level 1 (/etc/rc1.d), or run level 2 (/etc/rc2.d) directory whose names begin with K. This includes the RFS file K65rfs and any other files that you choose to place in /etc/rc.0, /etc/rc.1, or /etc/rc.2. The RFS file K65rfs, which is linked to /etc/init.d/rfs, runs **umount**, **unadv**, and **rfstop** to stop Remote File Sharing mode.

You can add to one of the run level directories (say /etc/rc2.d) your own files to be linked to custom shell scripts stored in /etc/init.d. Begin the name of the /etc/rc2.d file with K (kill), then include a two-digit sequence number, and lastly include the name of the linked file in /etc/init.d. Here is the format, with spaces added for ease of reading:

```
K  nn  name
```

For example, suppose you'd like to place shell scripts called unadvertise and unmount in /etc/init.d. To make these the first and second scripts executed, use 01 and 02 as the sequence numbers, use the shell **stop** option, and name the files as shown here:

| /etc/rc2.d | /etc/init.d |
|---|---|
| K01unadvertise | unadvertise |
| K02unmount | unmount |

## 41.3 Maintaining domains

A typical domain contains a primary name server, possibly at least one secondary name server, and other hosts without name server duties. The responsibilities of a domain administrator include adding and removing hosts, changing and reassigning name servers, and taking care of recovery when one of the systems fails.

### Basic information

Before getting into specific tasks, let's take a look at how you can obtain basic information about a domain and its member hosts. The following four files in directory /usr/nserve contain the information indicated:

| /usr/nserve | |
|---|---|
| *domain* | The name of each host's domain |
| *netspec* | The name of each host's RFS network |

*rfmaster* The names and addresses of the primary and any secondary name servers for your domain, along with the same information on other domains with which you share resources

*loc.passwd* Each host's authentication password

The files that contain information about users, groups, and passwords are described in Chapter 40.

Here are a few procedures for obtaining information quickly:

1. Display the name of your host's domain and network type:

   ```
   # dname -a
   ```

2. Display the name of the current name server:

   ```
   # rfadmin
   ```

3. Display the names and addresses of your domain's primary and secondary name servers:

   ```
   # pg /usr/nserve/rfmaster
   ```

4. Display the names and addresses of the primary and secondary name servers for domains with which your domain shares resources:

   ```
   # pg /usr/nserve/rfmaster
   ```

Now we'll proceed to the actual maintenance tasks.

## Adding a new host

To add a new host to a domain, go to the primary name server and enter an **rfadmin** command with the **-a** (add) option. For example, to add host **mrktg** to domain **ace**, enter the following:

```
# rfadmin -a ace.mrktg
Enter password for mrktg:
Re-enter password for mrktg:
```

The password you enter (if any) will be stored in **/usr/nserve/auth.info/ace/passwd**.

## Removing a host

To remove a host from a domain, go to the primary name server and enter an **rfadmin** command with the **-r** (remove) option. For example, to remove host **eng** from domain **ace**, enter the following:

```
# rfadmin -r ace.eng
```

If **eng** is not a name server, its entry will be removed from **/usr/nserve/auth.info/ace/passwd**.

## Changing domain name servers

To reassign the responsibilities of domain name server, you must go to the current primary name server, begin an editing session with the **rfmaster** file, and make the changes there. For example, suppose **mrktg** is currently the primary name server and **eng** is currently the secondary name server for domain **ace**, as shown here:

**/usr/nserve/rfmaster**

```
ace          p   ace.mrktg        [Primary name server]
ace          s   ace.eng          [Secondary name server]
ace.mrktg    a   mrktg.serve      [Network address of primary]
ace.eng      a   eng.serve        [Network address of secondary]
```

To make **standard** the new primary name server and **training** and **field_s** the new secondary name servers, modify the file so that it looks like this:

```
ace             p   ace.standards    [Primary name server]
ace             s   ace.training     [Secondary name server]
ace             s   ace.field_s      [Secondary name server]
ace.standards   a   standards.serve  [Network address of primary]
ace.training    a   training.serve   [Network address of secondary]
ace.field_s     a   field_s.serve    [Network address of secondary]
```

Now go to each of the five machines named (**mrktg**, **eng**, **standards**, **training**, and **field_s**) and execute the following two commands to activate the changes:

```
# rfstop
# rfstart
```

## Adding a new domain

To be able to share resources with another domain in your network, find out the names and network addresses of the primary and secondary name servers and add them to **rfmaster**. The procedure is similar to the one for changing name servers for your own domain, described in the previous section, "Changing Domain Name Servers".

### Recovery procedures

The Remote File Sharing system provides for the orderly restoration of resource sharing any time one of the machines in a domain fails. The machine that fails may be a file server, which is sharing resources with other machines, or a name server, which is keeping track of the names of resources that are being shared.

1. *File server*—The **rfudaemon**, which runs continuously whenever RFS is active, monitors the system for any of these three events:

   **fuwarn** warning to users just before a forced unmount (**fumount**)

   **fumount** actual unmount of a shared resource disconnect—dropping of a link to a shared resource If any of these occurs, **rfudaemon** starts shell script **rfuadmin**, which initiates one of three recovery procedures, depending on which event triggered the recovery. You can modify **rfuadmin** if you wish.

2. *Primary name server*—The system immediately switches responsibility to a secondary name server for a temporary duration. When the primary is restored, name service responsibility is not restored automatically; the system administrator must restore it manually from the acting name server with the **rfadmin -p** command. It will not be possible to add or delete hosts or change host authentication passwords until this has been done.

3. *All name servers*—If all primary and secondary name servers fail at the same time, there will be no record of which resources have been advertised. When the primary is restored, this information will have to be restored in one of two ways from each host that is sharing resources:

   - Execute **adv -m** for each resource already advertised.
   - Go to **init 2** and back to **init 3** and readvertise all resources to restart the RFS system.

## 41.4 Maintaining hosts

The procedures for setting up a domain and individual hosts were covered in Chapter 40, "Basic Resource Sharing." In this section we'll elaborate on some of those procedures. In particular, we'll discuss starting up and shutting down a host and changing a host's authentication password.

## Basic information

Before getting into specific tasks, let's take a look at how you can obtain basic information about your domain and your host:

1. Display the name of your host's domain and network type:

   ```
   # dname -a
   ```

2. Display the name of the current name server:

   ```
   # rfadmin
   ```

   Now we'll proceed to the actual maintenance tasks.

## Starting up a host

The command used to start Remote File Sharing on your host and notify the domain name server is **rfstart**, which is usually run automatically when the host enters Remote File Sharing state (**init 3**). The general format of the command is

```
# rfstart [-v] [-p primary]
```

where

| | |
|---|---|
| **-v** | requests verification of any host that attempts to mount your resources |
| **-p** *primary* | specifies the network address of the primary name server |

The **-v** option provides mount protection by requiring a password from any host that attempts to mount one of your resources. The names and passwords of other hosts in a given domain are kept in a **passwd** file stored in directory **/usr/nserve/auth.info/domain** (where *domain* is the name of the domain). The attempted mount will fail if the password entered doesn't match, the host isn't named in the **passwd** file, or there is no **passwd** file for the host.

If you issue **rfstart** without the **-v** option, validation will depend on whether or not the password file exists and the host is listed:

1. If **domain/passwd** exists and the other host is listed, the host will still be required to enter the correct password.

2. If **domain/passwd** doesn't exist or the host isn't listed, the host will be able to mount your resources without a password.

The **-p** option, which is required only the first time you start Remote File Sharing, allows you to provide the network address of the primary name server if your host's **rfmaster** file is unable to provide it because of an error.

To verify that Remote File Sharing has started properly, use the **ps -e** command to make sure all the RFS processes are running:

```
# ps -e
listen
rfdaemon
nserve              [This is only a partial list]
rfudaemon
server
# _
```

If Remote File Sharing fails to start, here is a checklist of possible problems:

□ Are you logged in as **root**?

□ Is the network running properly?

□ Is your host's network listener runing properly?

□ Is the domain's name server up and running?

□ Has your host's name been added to the domain?

□ Is there an error in your host's **/usr/nserve/rfmaster** file?

□ Is there an error in your host's **/usr/nserve/loc.passwd** file?

## Shutting down a host

The procedure for shutting down a host, which will be carried out automatically when your host leaves Remote File Sharing state (**init 3**), is as follows:

1. Unadvertise all your resources with **unadv**.

2. Unmount all remote resources with **umount**.

3. Make sure other hosts have unmounted all your resources (using **fumount** if necessary).

4. Disconnect your host from the network with **rfstop**.

### Changing a host's password

To change your host's password, make sure Remote File Sharing is running on your host and on the primary name server (not a secondary name server). Then execute the **rfpasswd** command, which will prompt you for the old and new passwords:

```
# rfpasswd
```

## 41.5 Monitoring remote file sharing

UNIX System V, Release 3 offers five tools for monitoring the flow of information in the Remote File Sharing system. The reports they produce can help you determine whether your resources are located in the most appropriate places on your network and whether it may be possible to improve performance by adjusting system parameters. (System parameters are discussed in the next section.) The monitoring tools are summarized in Table 41.1, then described in the paragraphs that follow.

TABLE 41.1. Tools for Monitoring RFS

| Command | Description |
|---|---|
| **sar -Dc** | Monitor incoming and outgoing requests for resources. |
| **sar -Du** | Display the percent of CPU time spent on various activities. |
| **sar -S** | Monitor server processes and remote requests. |
| **fusage** | Find out how much other hosts are using your resources. |
| **df** [*resource*] | Display remaining disk space on a remote resource. |

### Monitoring requests for resources **sar -Dc**

Your host keeps a record of incoming requests for local resources and outgoing requests for remote resources. The **sar -Dc** command displays averages for system calls per second (scall/s), read and write system calls per second (sread/s and swrit/s), forks and exec's per second (fork/s and exec/s), and characters read and written per second (rchar/s and wchar/s). See Figure 41.1 for a sample display.

### Displaying CPU time by activity **sar -Du**

You can display the percentage of time your host's CPU spends processing local and remote system calls (%sys local and %sys remote), along with

FIGURE 41.1. The display for **sar -Dc**.

```
$ sar -Dc
compl ace 3.0v1 0572 VAX    04/16/87

00:05:30     scall/s sread/s swrit/s fork/s exec/s  rchar/s wchar/s
01:05:30
       in       5      2       3                 0.00      379     236
       out      4      3       2                 0.00      254     309
       local  148     35      18      0.81       1.46    14236    5921
02:05:30
       in       4      3       4                 0.00      346     265
       out      3      2       2                 0.00      261     297
       local  152     29      21      0.74       1.39    13602    6053
03:05:30
       in       6      4       5                 0.00      293     286
       out      4      3       4                 0.00      278     312
       local  159     41      26      0.83       1.51    15433    7115

Average       153     35      22      0.79       1.45    14423    6363
$ _
```

other information with the **sar -Du** command. See Figure 41.2 for a sample display.

FIGURE 41.2. The display for **sar -Du**.

```
$ sar -Du
compl ace 3.0v1 0572 VAX    04/16/87

00:05:30     $usr    %sys    %sys    %wio    %idle
                     local   remote
01:05:30      23      21      12      11      33
02:05:30      17      28      13      12      30
03:05:30      15      20      19      15      31
04:05:30      12      18      15       9      46
05:05:30      10      15      12       7      56
06:05:30      11      16      14       8      51

Average       15      20      14      10      41
$ _
```

## MONITORING SERVERS AND REQUESTS **sar -S**

Incoming requests for resources are handled by *server processes.* An excess number of requests may have to be queued until a server becomes available. You can display information about requests and servers with the **sar -Du** command. See Figure 41.3 for a sample display.

FIGURE 41.3. The display for **sar -S**.

```
$ sar -S
compl ace 3.0v1 0572 VAX    04/16/87

00:05:30    serv/lo-hi  request    request    server   server    %
            5-25        %busy      avg lgth   %avail   avg       avail
01:05:30     13          32        26          62        3
02:05:30     12          21        18          70        5
03:05:30      8          40        37          83        7
04:05:30     10          36        33          88        6
05:05:30     11          34        21          93        6
06:05:30     14          52        16          74        4

Average      12          36        25          78        5
$ _
```

If too many servers are available or if there aren't enough servers to handle all the requests, you can make adjustments to the MINSERVE and MAXSERVE parameters (discussed in the next section).

## Displaying usage by other hosts **fusage**

The **fusage** command can tell you how much other hosts are using your resources. The display shows how much data has been read from and written to your resources by each remote host. See Figure 41.4 for a sample display.

FIGURE 41.4. The display for **fusage**.

```
# fusage

FILE USAGE REPORT FOR rt200


                    /dev/dsk/c1d0s0     /
                                        /
                              rt200       507 Kb
                            Clients       432 Kb
                              TOTAL       939 Kb

                    /dev/dsk/c1d1s2     /usr
                                        /usr
                              rt200       498 Kb
                            Clients       375 Kb
                              TOTAL       873 Kb


# _
```

You can also confine the output of this command to a single advertised directory by naming the directory after the command, like this:

```
# fusage /usr/admin/corr
```

## DISPLAYING DISK SPACE ON A REMOTE HOST **df**

The **df** command can tell you how much space is left on a disk on which a remote resource is located. If *resource* is a resource mounted on your host, then you can name *resource* after the **df** command. See Figure 41.5 for a sample display.

FIGURE 41.5. The display for **df**.

```
# df CUSTOMERS ENTRIES H_DISK

/usr/admin/cust      (CUSTOMERS ):    4029 blocks     1936 i-nodes
/usr/admin/entries   (ENTRIES   ):    6754 blocks     2543 i-nodes
/dev/rdsk/1          (H_DISK    ):    3978 blocks     1624 i-nodes

# _
```

You can also confine the output of this command to a single advertised directory by naming the directory after the command, like this:

```
# fusage /usr/admin/corr
```

# 41.6 Adjusting performance

Eight parameters located in the **/etc/master.d/du** file allow you to adjust the performance of your Remote File Sharing system. The parameters are summarized in Table 41.2, then described in the paragraphs that follow.

TABLE 41.2. Performance Parameters

| Command | Description |
|---|---|
| **NRDUSER** | Number of user entries for receive descriptors |
| **NRCVD** | Maximum number of receive descriptors |
| **NSNDD** | Maximum number of send descriptors |
| **NSRMOUNT** | Maximum number of entries in the mount table |
| **NADVERTISE** | Maximum number of entries in the advertise table |
| **MAXGDP** | Maximum number of virtual circuits |
| **MINSERVE** | Minimum number of server processes |
| **MAXSERVE** | Maximum number of server processes |

### Receive descriptors — NRDUSER NRCVD

Every time a user from another system names a file or directory in your system, your system creates one receive descriptor. Use **NRDUSER** to indicate how many user entries to allocate to receive descriptors. By lowering the value of **NRCVD**, you can reduce the number of local files and directories that remote users can access at any given moment.

### Send descriptors — NSNDD

Every time a user from your system names a file or directory in anothe system, your system creates one receive descriptor. By lowering the value of **NSNDD**, you can reduce the number of remote files and directories that your own users can access at any given moment.

### Advertise table — NADVERTISE

Every time you advertise a resource, your system places an entry in your advertise table. With **NADVERTISE**; you can set a limit on the number of resources that you can advertise.

### Virtual circuits — MAXGDP

Every time any host mounts a resource on another host, one connection, or virtual circuit, is set up on the network. Between any two hosts A and B there can be either no virtual circuits (neither host has any resources from the other mounted), one virtual circuit (one host has resources from the other mounted), or two two virtual circuits (each host has resources from the other mounted). You can set the maximum number of virtual circuits allowed on the network with **MAXGDP**.

### Server processes — MINSERVE MAXSERVE

Server processes on your system handle requests for your resources from users of other systems (monitored by the **sar -S** command). You can set both the minimum (**MINSERVE**) and the maximum (**MAXSERVE**) number of server processes allowed.

## 41.7 Summary

In this chapter, you learned how to start and stop run level 3 (Remote File Sharing mode), maintain domains and hosts, monitor RFS performance, and tune various performance parameters.

### Remote file sharing mode

System V, Release 3 introduces a new run level 3 (Remote File Sharing mode), initiated by the **init 3** command. When you start run level 3, the system executes all shell scripts in directory /etc/rc3.d whose names begin with S. One of these, S21rfs, is a standard file; the rest are custom files that you place in the file. When you stop run level 3, the system executes all shell scripts in directory /etc/rc3.d whose names begin with K.

### Maintaining domains

The files for each domain are stored in directory /usr/nserve. These contain the name of the domain (*domain*), the name of each host's RFS network (*netspec*), the names and addresses of the name servers (*rfmaster*), and each host's authentication password (*loc.passwd*). You have commands to display the name of your host's domain and network type, the name of the current name server, the names and addresses of name servers, along with methods for adding a new host, removing a host, changing domain name servers, adding a new domain, and taking care of error recovery.

### Maintaining hosts

You have commands to display the name of your host's domain and network type and the name of the current name server. You also have methods for starting up a host, shutting down a host, and changing a host's password.

### Monitoring and tuning performance

You have five tools for monitoring the RFS performance: **sar -Dc**, **sar -Du**, **sar -S**, **fusage**, and **df**. You can adjust performance with any of eight parameters: **NRDUSER**, **NRCVD**, **NSNDD**, **NSRMOUNT**, **NADVERTISE**, **MAXGDP**, **MINSERVE**, and **MAXSERVE**.

# 42

# RFS Security

## 42.1 Introduction

Some security measures have already been mentioned in previous chapters on RFS. In this chapter we'll review these, and then describe additional measures that allow you to protect your resources through an extension of the UNIX system of permissions known as *user mapping.* Table 42.1 gives a quick summary of the security measures available to each system administrator.

TABLE 42.1. Summary of RFS Security Measures

| Security Issue | Measure Available |
|---|---|
| *Connection*—Which hosts are allowed to connect to yours? | *Verification*—**rfstart -v** requires verification via password before the remote host is allowed to connect to your host. |
| *Mounting*—Which hosts are allowed to mount your resources? | *Client list*—**adv** ... *client(s)* restricts permission to mount to the clients named. |
| *Permissions*—What permissions are granted to users who use your resources? | *Mapping*—**idload** allows you to control permissions for remote users and groups, using features within your own system. |

### THE CONCEPT OF MAPPING

To control the permissions of remote users who are accessing your resources, you can assign to each remote user the permissions already granted to one of your own local users. The technique for making these assignments is called *mapping,* and each remote user (or group) id is said to be *mapped* to a local user (or group).

As you may recall from Chapter 35, "System Security," each user on a system has a unique *name* and *number* (id) for identification. For example, in these three entries from the password file on system **sales**, you can see that each user has a login name and also an identifying user number (plus a group number):

**sales**

```
robin:MWxG24o.118fM:10:50:Robin Russell:/usr/robin:/bin/sh:
pat:Hj3*gFW,n0x5:20:50:Pat Wyman:/usr/pat:/bin/sh
bill:Uo.68mG3h"d6s:30:50:Bill Freeman:/usr/bill:/bin/csh
```

Robin's user id is 10, Pat's is 20, and Bill's is 30; the group id for all three is 50. In the following three entries from the password file on system mrktg, you will find similar information:

**mrktg**

```
mary:Bk8Lio45Cxf:10:500:Mary Spaulding:/usr/mary:/bin/sh
walt:BvCX78z,9ofD:20:500:Walter Field:/usr/walt:/bin/sh
ted:Vu5.WqOxc8eA:30:500:Ted Patterson:/usr/ted:/bin/sh
```

Mary's user id is 10, Walter's is 20, and Ted's is 30; the group id for all three is 500. We can summarize the identifying information in these password files as follows:

| sales | | mrktg | |
|---|---|---|---|
| **Name** | **Number** | **Name** | **Number** |
| robin | 10 | mary | 10 |
| pat | 20 | walt | 20 |
| bill | 30 | ted | 30 |

Then we could set up a mapping scheme either by name or by number:

| By Name | | | By Number | | |
|---|---|---|---|---|---|
| robin | ⇒ | mary | 10 | ⇒ | 10 |
| pat | ⇒ | walt | 20 | ⇒ | 20 |
| bill | ⇒ | ted | 30 | ⇒ | 30 |

With either mapping, Bill will have the permissions of Ted any time he accesses the resources of Ted's host (mrktg). Similarly, Robin will have Mary's permissions and Pat will have Walt's. When a name or number on one host is mapped to the same name or number on another, this is called a *transparent mapping.* The mapping by number shown above is an example.

## Mapping options available

The Remote File Sharing system allows you to make mapping as simple or as complex as you choose. If you take no action at all, each remote user will be mapped to a *special guest* id on your host (the highest id allowed plus one). A second approach is to map each remote user either to a single name or number or to the name or number that matches the user's own. A third approach is to map each remote user individually to a specific local user. Here is a brief summary of your options, with a pictorial example of each:

1. *No mapping*—one choice, whether by name or by number:

| By Name | | | By Number | | |
|---|---|---|---|---|---|
| robin | ⇒ | guest | 10 | ⇒ | 101 |
| pat | ⇒ | guest | 20 | ⇒ | 101 |
| bill | ⇒ | guest | 30 | ⇒ | 101 |

2. *Default mapping*—four different choices:

| Map to One Name | | | Map to One Number | | |
|---|---|---|---|---|---|
| robin | ⇒ | ted | 10 | ⇒ | 30 |
| pat | ⇒ | ted | 20 | ⇒ | 30 |
| bill | ⇒ | ted | 30 | ⇒ | 30 |

| Map to Same Name | | | Map to Same Number | | |
|---|---|---|---|---|---|
| robin | ⇒ | robin | 10 | ⇒ | 10 |
| pat | ⇒ | pat | 20 | ⇒ | 20 |
| bill | ⇒ | bill | 30 | ⇒ | 30 |

3. *Specific mapping*—by name or by number:

| By Name | | | By Number | | |
|---|---|---|---|---|---|
| robin | ⇒ | mick | 10 | ⇒ | 50 |
| pat | ⇒ | phil | 20 | ⇒ | 40 |
| bill | ⇒ | paula | 30 | ⇒ | 60 |

## 42.2 Specifying mapping

If you choose to set up mapping, you will have to enter a pair of *rules files* (one for user ids and one for group ids). A command called **idload** will read these files, along with the appropriate password files, to perform the actual mapping requested. Note that, if you choose to map *by name*, you and the other hosts will have to share password files with each other. All files used by **idload** are stored in the /usr/nserve/auth.info directory: /usr/nserve/auth.info

| | |
|---|---|
| **domain/passwd** | Password files for individual domains |
| **domain/host/passwd** | Password files for individual hosts |
| **domain/host/group** | Group files for individual hosts |
| **uid.rules** | Rules for mapping remote user ids |
| **gid.rules** | Rules for mapping remote group ids |

## Rules files

A rules file (user or group) is composed of blocks, one *global* block that applies to all hosts and possibly host blocks for individual hosts. User rules files and group rules files employ the same format, in which all statements are optional. The following simple example of a user rules file (which contains a global block only) maps each user from any host to the user's own id by number and maps **root** (id 0) to the special guest login:

```
global
default transparent
exclude 0
```

Here is an explanation of each line allowed in a global block:

`global` These rules apply to all remote hosts.

`default` Map each remote user to either of the following:

| | |
|---|---|
| *local_id* | a single local user id number |
| `transparent` | the same id number |

`exclude` Exclude the remote users with the id number(s) indicated from the default rule and map the number(s) instead to the special guest id:

| | |
|---|---|
| *remote_id* | a single remote id number |
| *remote_id-remote_id* | a range of remote id numbers |

`map` Perform a special mapping from the remote id number indicated to the local id name or number indicated:

| | |
|---|---|
| *remote_id:local* | from *remote_id* to *local* |

Note that in the global block, you must map *by number*, not *by name*.

You are also allowed to include additional blocks for individual hosts (or sets of hosts). Here is an explanation of each line in a host block:

`host` These rules apply only to the remote host(s) named:

*domain.host* [ *domain.host* ... ]

`default` Map each remote user to either of the following:

| | |
|---|---|
| *local* | a single local user id name or number |
| `transparent` | the same id number |

`exclude` Exclude the remote users with the id number(s) or the id name indicated from the default rule and map the number(s) instead to the special guest id:

| | |
|---|---|
| *remote_name* | a single remote id name |

| | |
|---|---|
| *remote_id* | a single remote id number |
| *remote_id-remote_id* | a range of remote id numbers |

`map` Perform a special mapping from the remote id number or name indicated as indicated:

| | |
|---|---|
| *remote:local* | from *remote* to *local* |
| *remote* | from *remote* to the same name or number |
| `all` | from each remote name to the same name |

In a host block, you are allowed to map either *by name* or *by number.* Once again, you may have as many host blocks as you wish—either for single hosts or for sets of hosts.

To summarize for both types of blocks (*global* and host), the `default` line maps remote users either to a single local user or to the same user id number (`transparent`), while the `exclude` line exempts remote users from the rules of the `default` line, mapping them instead to the special guest id. The `map` line then allows you to set up additional mappings—from one id to another, from one id to itself, or from all id names to themselves (`all`).

## Examples of mapping by number

Here are some examples of mapping remote users by id number, along with explanations following each:

```
global
default 100
exclude 0
```

Map all users except **root** (user id 0) to local user id number 100; map **root** to the special guest id. In the interest of security, it's usually best to restrict **root** to special guest permissions.

```
global
default transparent
exclude 0-50
map 60:100 70:100
```

Map all users except **root**, all users with id numbers from 1 to 50 and 60 and 70 to their own user id numbers; map **root** and users 1-50 to the special guest id; map users 60 and 70 to local user 100.

```
global
default transparent
exclude 0-50
map 60:100 70:100
```

```
host ace.eng
default 200
exclude 0-199
```

In this example, the `global` block is the same as in the previous example. However, a `host` block has been added to indicate a different set of rules for users on remote system **ace.eng**: map all users except **root** and users 1-199 to `200`; map **root** and users 1-199 to the special guest id.

## EXAMPLES OF MAPPING BY NAME

Before you can map users by name, you and the other hosts will first have to share your password and group files with each other (described in a later section). Once you've done this, here are some examples of mapping remote users by id name, along with explanations following each:

```
global
default transparent
exclude 0

host ace.acctg
default tanya
exclude 0 paul lp
```

In this example, a `host` block has been added to indicate a different set of rules for **ace.acctg**: map all users except **root**, **paul**, and **lp** to **tanya**; map **root**, **paul**, and **lp** to the special guest id.

```
global
default transparent
exclude 0

host ace.acctg
exclude 0 paul lp
map robin pat:will
```

In this example, the `exclude` block exempts the users named from the `map` rules, rather than from the `default` rules. According to the `map` line, map **robin** to **robin** and **pat** to **will**; then map **root**, **paul**, and **lp** to the special guest id.

```
global
default 100
exclude 0

host ace.acctg
exclude 0 paul lp
map all
```

In this example, the `exclude` block again exempts the users named from the `map` rules. According to the `map` line, map each user except **root**, **paul**, and **lp** to the same name; then map **root**, **paul**, and **lp** to the special guest id.

## THE MAPPING COMMAND **idload**

Once you've set up your user and group rules files, the **idload** command, which is executed automatically whenever a remote mount takes place, reads these files and performs the mapping requested. You can execute this command manually either to display your current mappings or to change the directory and filenames for your rules files. To display your current mappings, use this form of the command:

```
# idload -n

TYPE  MACHINE        REM_ID       REM_NAME      LOC_ID      LOC_NAME

USR   GLOBAL         DEFAULT      n/a           100         remote_d
USR   GLOBAL         0            n/a           60001       guest_id
USR   ace.acctg      0            n/a           60001       guest_id
USR   ace.acctg      67           paul          60001       guest_id
USR   ace.acctg      243          lp            60001       guest_id
USR   ace.acctg      10           robin         120         robin
USR   ace.acctg      20           pat           136         will

GRP   GLOBAL         DEFAULT      n/a           100         remote_d

# _
```

The display shown here describes the mappings requested in the following default rules files:

```
                             /usr/nserve/auth.info/uid.rules

global                       global
default 100                  default 100
exclude 0

host ace.acctg
exclude 0 paul lp
map robin pat:will
```

To use other files for your rules files and the directory that contains them, you can use a form like this for the **idload** command:

```
# idload -u /usr/perm/u.rules -g /usr/perm/g.rules /usr/perm
```

The general format for the **idload** command is as follows:

```
# idload [-n] [-u  u'rules] [-g  g'rules] [ directory]
```

where

**-n** requests a display only with no updating of mapping.

**-u** *u_rules* indicates a rules file other than /usr/nserve/auth.info/uid.rules.
**-g** *g_rules* indicates a group file other than /usr/nserve/auth.info/gid.rules.
*directory* is a directory name other than /usr/nserve/auth.info.

## 42.3 Procedures for mapping by name

As mentioned earlier, you have some additional tasks to perform when you map remote users by name. These tasks include all the steps required for sharing password and group files with other hosts.

### Procedures for the domain

The domain administrator has to advertise the domain and collect user and group information. Here are the steps:

1. Advertise the domain:

   □ Execute a command like the following:

   ```
   # adv -d "ace data" ACE /usr/nserve/auth.info/ace
   ```

   □ Now other hosts have access to your host password file and to the names of your users

2. Collect user and group information:

   □ Ask each host administrator to access the directory you just advertised and store their password and group files

   □ Now advertise the directory again (read-only this time):

   ```
   # adv -r -d "Sun data" ACE /usr/nserve/auth.info/ace
   ```

### Procedures for the host

Each host administrator has to mount and copy the domain directory, set verification, set remote user permissions, and create the password and group files. Here are the steps:

1. Mount and copy the domain directory:

   □ Create a new directory if necessary:

   ```
   # mkdir /d_info
   ```

   □ Mount the domain directory:

```
# mount -r -d ACE /d_info
```

□ Change to the new directory:

```
# cd /d_info
```

□ Copy all files into /usr/nserve/auth.info/ace:

```
# find . -depth -print | cpio -pd /usr/nserve/auth.info/ace
```

□ Change back to the root directory:

```
# cd /
```

□ Unmount the domain directory:

```
# umount ACE
```

2. Set up verifications:

□ Open the password file:

```
# vi /usr/nserve/auth.info/ace/passwd
```

□ Delete passwords for any hosts that you don't need to verify after issuing **rfstart -v**

3. Set up remote user permissions:

□ Create a user rules file (/usr/nserve/auth.info/uid.rules).

□ Create a group rules file (/usr/nserve/auth.info/gid.rules).

4. Deliver password and group files to the domain name server:

□ Make copies of your /etc/passwd and /etc/group files.

□ Place them in the directory /usr/nserve/auth.info/ace/sales (assuming domain ace and host sales).

This last step matches step 2 "Collect user and group information" under "Procedures for the Domain" above. Now you are prepared to map remote users by name.

## 42.4 Summary

In this chapter you learned how to maintain the security of your host when sharing resources. Mapping allows you to assign to a remote user all the permissions granted to a local user on your own system, using either user names or user numbers (ids).

Using user and group rules files, you can request either default mapping (either to a single id or to matching ids) or specific mapping. Each rules file must contain one `global` block and may contain additional blocks for individual hosts, using the `default`, `exclude`, and `map` statements. You can display the mappings you have selected with the **idload -n** command.

If you choose to map *by name*, several additional steps are necessary. The domain administrator must advertise the domain and collect user and group permission information. Each host administrator must mount and copy the domain directory, set authorizations, set remote user permissions, and create the **passwd** and **group** files.

# Appendices

# Appendix A

# Summary of Basic Commands and Symbols

Here is a list of basic UNIX commands and symbols.

## A.1 Basic commands for starting out

| | |
|---|---|
| **date** | Display date and time |
| **who** | List logged on users |
| **@** | Erase an entire command line |
| **# or ^H** | Erase a character on the command line |
| CTRL-D | Log off the system; sometimes end-of-file (EOF) |
| **man** | Call up on-line UNIX documentation |
| **learn** | Call up on-line UNIX tutorials |

## A.2 Working with directories and files

| | |
|---|---|
| / | The root directory |
| /bin | The directory that contains most system shell procedures and executable command programs |
| /usr | The directory that usually contains user accounts |
| . | The current directory |
| .. | The parent directory |

| | |
|---|---|
| >*file* | Redirect output *to file* |
| >>*file* | Redirect output and append to *file* |
| <*file* | Redirect input *from file* |
| *cmd1* \| *cmd2* | Pipe: use the output of *cmd1* as the input for *cmd2* |

| | |
|---|---|
| **cat** *file* | Display the contents of *file*<br>CTRL-S : Halt scrolling of screen output<br>CTRL-Q : Resume scrolling of screen output |
| **pg** *file* | Display the contents of *file* by screen (UNIX) |
| **more** *file* | Display the contents of *file* by screen (XENIX) |

| | |
|---|---|
| **head** *file* | Display the opening lines of *file* (XENIX) |
| **tail** *file* | Display the last lines of *file* |
| | |
| **cat** *f1 f2 >file* | Concatenate files *f1* and *f2* into file *file* |
| **mv** *f1 f2* | Change the name of file *f1* to *f2* |
| **mv** *f1 f2 dir* | Move files *f1* and *f2* to directory *dir* |
| **cp** *f1 f2* | Make a copy of file *f1* and call the copy *f2* |
| **cp** *f1 f2 dir* | Copy files *f1* and *f2* to directory *dir* |
| **rm** *file* | Remove *file* from the current directory |
| **ln** *file* | Link *file* to the current directory |
| **ls** | List the names of files and subdirectories |
| **ls -l** | Display a *long* directory list |
| **pwd** | Display the name of your working directory |
| **cd** *dir* | Change your working directory to *dir* |
| **mkdir** *dir* | Create directory *dir* |
| **rmdir** *dir* | Remove directory *dir* (empty—no files) |
| **rmdir -r** *dir* | Remove directory *dir* and every file in it |
| **mv** *d1 d2* | Change the name of directory *d1* to *d2* |
| | |
| **chmod** ... *file* | Change the access permissions to *file*, using the following abbreviations: |

| | | | | | |
|---|---|---|---|---|---|
| **u** | user | **+** | add | **r** | read |
| **g** | group | **-** | remove | **w** | write |
| **o** | others | **=** | absolute | **x** | execute |

## A.3 Searching: forming regular expressions

| | |
|---|---|
| ^ | Beginning of line |
| **$** | End of line |
| **?** | Match a single character in a search |
| ***** | Match any character in a search |
| **[ ]** | Match any of the characters enclosed in a search |
| \ | Interpret the following special character literally |

## A.4 Setting basic features

| | |
|---|---|
| **passwd** | Change your password |
| **newgrp** *name* | Switch to working group *name* |
| **newgrp** | Switch back to your default group |
| **stty** | Set your terminal options |
| **tabs** *term* | Specify tab settings for terminal *term* |

## A.5 Working with processes

| | |
|---|---|
| **&** | Run a process in the background (placed at the end of a command line) |
| **DEL** | Abort a foreground process |
| **kill** *nnnn* | Abort background process with PID number *nnnn* |
| **ps** | Display the status of active processes associated with your terminal |
| **time** *process* | Display the amount of time required to run *process* |

## A.6 Processing information

| | |
|---|---|
| **dc** | Call up the desk calculator |
| **bc** | Call up the high-precision calculator |
| **cal** | Display a calendar for a given month or year |
| **echo** | Display a string on the screen |
| **sort** *file* | Sort *file* by the sequence specified |
| **grep 'p'** *file* | Search for pattern *p* in *file* |
| **awk** '*p* { *a* }' *file* | Select lines in *file* with pattern *p* and process them by taking action *a* |
| **wc** *file* | Count the lines, words, and characters in *file* |
| **spell** *file* | Check spelling of words in *file* |
| **diff** *f1 f2* | Display the differences between files *f1* and *f2* |
| **cmp** *f1 f2* | Compare files *f1* and *f2* |
| **comm** *f1 f2* | Display lines common to files *f1* and *f2* |
| **tr** *s1 s2 file* | Translate from string *s1* to *s2* in *file* |
| **lp** *file* | Send contents of *file* to the lineprinter |
| **nroff** *file* | Format and print the contents of *file* |

## A.7 Communicating

| | |
|---|---|
| **mail** *user* | Send electronic mail to *user* |
| **mail** | Read your electronic mail |
| **calendar** | Display events stored by date as a reminder |
| **write** *user* | Write directly to *user*'s terminal |
| **cu** | Call up another UNIX system |
| **uucp** | Communicate with another UNIX system |

# Appendix B

# Summary of **ed**

## B.1 Commands

The general form of an **ed** command is a letter, preceded by one or two line numbers (generally optional). Two **ed** commands, **e** (edit) and **r** (read), must be followed by a filename, and **w** (write) may be. Only one command is allowed per line, but **p** (print) may follow any command except **e** (edit), **r** (read), **w** (write), or **q** (quit). A backslash (\) may be used at the end of each line to continue a long command string on the next line.

**a** *Append* lines to the editing buffer, starting at the current line, unless another line is specified. Appending continues until a period appears by itself as the first character on a separate line. Once appending is completed, the last line appended is considered the current line.

**c** *Change* the lines specified to the text that follows. The text must be terminated with a period on a separate line. If no lines are specified, replace only the current line. The last line changed becomes the new current line.

**d** *Delete* the lines of text specified. If no lines are specified, the current line is deleted. Generally, the first line following the deleted text becomes the new current line. However, if the last line is deleted, then the new last line becomes the new current line.

**e** *Edit* a file. If the editing buffer contains text when **e** is issued, it will be overwritten by the contents of the new file. To avoid loss of text, use the **w** (write) command for the buffer before using **e**.

**f** *File* is to recall the name of the active file. If you type a filename after **f**, that is the name that will become the new active file. It will also be recalled the next time you issue **f**.

**g** *Global search* is to execute the command specified for all lines in the editing buffer.

**i** *Insert* the text that follows in front of the current line (or in front of the line specified). The text to be inserted must be terminated by a period on a separate line. The last line inserted becomes the current line.

**m** *Move* the lines specified to the location immediately following the line specified after **m**. The last line moved becomes the current line.

**p** *Print* (that is, display) the lines specified. If no lines are specified, display the current line. Press RETURN to display the line following the current line.

**q** *Quit* **ed** is to exit the editor and return to the shell. Unless you first issue **w** (write), **q** will discard the editing buffer.

**r** *Read* a file into the editing buffer, beginning at the end of the buffer, unless another line is specified. The last line read becomes the new current line.

**s** *Substitute* the first occurrence of one string for another in the lines specified. If no lines are specified, the substitution takes place only in the current line. The last line in which a substitution takes place becomes the new first line. If no substitution takes place, the current line remains unchanged.

**v** *Reverse global search* is to execute commands on those lines that do *not* match the string indicated.

**w** *Write* text from the editing buffer to a file. The file may be named either here or in an earlier **ed** or **e** (edit) command. The current line remains unchanged.

**/** *Context search* is to search for the next line that matches the string indicated by */string/*. The search begins at the line following the current line, and continues all the way through the editing buffer back to the current line if necessary. The line where the string is matched becomes the new current line.

**?** *Reverse context search* is to search backwards for the line that matches the string indicated by *?string?*. The search begins at the line preceding the current line, and continues all the way through the file—wrapping back around to the current line if necessary. The line where the string is matched becomes the new current line.

**.=** *Current line* is to display the number of the current line.

**$=** *Last line* is to display the number of the last line of the editing buffer.

**!** *Escape* is to execute a UNIX command without leaving **ed**.

## B.2 Special characters for searching

In search and replace commands, the following special characters have the meanings given here, except when preceded by a backslash (\). In those instances they are interpreted literally.

| | | | |
|---|---|---|---|
| ^ | Beginning of line | **$** | End of line |
| **.** | Match a character | * | Repeat a character |
| **[ ]** | Match one of the characters enclosed | | |
| **&** | Use the search string as the replacement string | | |
| \ | Turn off special meaning | | |

The following special characters are fixed, and do not change in meaning:

/ Delimiter for search strings and replacement strings; in a replacement string, you can also use any other character not found in the replacement string itself.

\( \) Tag a substring (in a search string).

\*n* Insert substring *n* (in a replacement string).

# Appendix C

# Summary of **vi** and **ex** Commands

You enter **vi** commands from *vi command mode.* These commands are never displayed on the screen. To enter an **ex** command, type a colon ( *:* ) to enter *ex command mode*, type the command, then type ESC or RETURN to return to *vi command mode.* The search commands begin with either / or ? and end with a RETURN. Commands that begin with :, /, and ? are displayed at the bottom of the screen.

## C.1 Moving the cursor

Move the cursor as indicated:

| | |
|---|---|
| **h** | One space to the left |
| **j** | One space down |
| **k** | One space up |
| **l** | One space to the right |

| | |
|---|---|
| **b** | Beginning of the previous word |
| **B** | Beginning of the previous word (ignore punctuation) |

| | |
|---|---|
| **w** | Beginning of the next word |
| **W** | Beginning of the next word (ignore punctuation) |

| | |
|---|---|
| **e** | End of the next word |
| **E** | End of the next word (ignore punctuation) |

| | |
|---|---|
| **(** | Beginning of sentence |
| **)** | End of sentence |
| { | Beginning of paragraph |
| } | End of paragraph |
| **[** | Beginning of section |
| **]** | End of section |
| **^** | First visible character of line |
| **0** | Beginning of line |
| **$** | End of line |

| | |
|---|---|
| **H** | Top of screen |
| **M** | Middle of screen |
| **L** | Bottom of screen |
| *n***G** | Line *n* (last line if *n* is omitted) |
| **%** | Matching symbol: ( ), { }, [ ] |
| **^U** | Scroll up (CTRL-U) |
| **^B** | Page back (CTRL-B) |
| **^D** | Scroll down (CTRL-D) |
| **^F** | Page forward (CTRL-F) |

## C.2 Adding new text

| | |
|---|---|
| **a** | Append text after the cursor |
| **A** | Append text at the end of the line |
| **I** | Insert text at the beginning of the line |
| **i** | Insert text before the cursor |
| **O** | Open a new line of text above the current line |
| **o** | Open a new line of text below the current line |

## C.3 Changing text

In each instance, change text from the cursor to the place indicated. All of the commands shown here may be used with multipliers (for example, **c5w**, **5cc**, **c5)**, **c5}**, and so on).

| | | | |
|---|---|---|---|
| **cw** | End of word | | |
| **cW** | End of word (ignore punctuation) | | |
| **c^** | First visible character of line | | |
| **c0** | Beginning of line | **c$** | End of line |
| **cc** | The entire line | | |
| **c(** | Beginning of sentence | **c)** | End of sentence |
| **c{** | Beginning of paragraph | **c}** | End of paragraph |

## C.4 Shifting text

In each instance, shift the text from the cursor to the place indicated a distance of one *shiftwidth* either left (<) or right (>). All of the commands may be used with multipliers (for example, **5**>>, >**5)**, <**5**}, and so on).

| | | | |
|---|---|---|---|
| << | Entire line | >> | Entire line |
| <**(** | Beginning of sentence | >**(** | Beginning of sentence |
| <**)** | End of sentence | >**)** | End of sentence |
| <{ | Beginning of paragraph | >{ | Beginning of paragraph |
| <} | End of paragraph | >} | End of paragraph |

## C.5 Deleting text

In each instance, delete text from the cursor to the place indicated. All of the commands shown here may be used with multipliers (for example, **d5w**, **5dd**, **d5)**, **d5**}, and so on).

| | | | |
|---|---|---|---|
| **dw** | End of word | | |
| **dW** | End of word (ignore punctuation) | | |
| **d^** | First visible character of line | | |
| **d0** | Beginning of line | **d$** | End of line (same as **D**) |
| **dd** | Entire line | | |
| **d(** | Beginning of sentence | **d)** | End of sentence |
| **d**{ | Beginning of paragraph | **d**} | End of paragraph |

## C.6 Searching and Replacing

| | |
|---|---|
| **tx or Tx** | Move the cursor next to $x$ on the current line |
| **ctx or cTx** | Change from the cursor to the character next to $x$ |
| **dtx or dTx** | Delete from the cursor to the character next to $x$ |
| **fx or Fx** | Find character $x$ on the current line |
| **cfx or cFx** | Change from the cursor to character $x$ on the current line |
| **dfx or dFx** | Delete from the cursor to character $x$ on the current line |
| /*word* | Search forward for *word* in the work area |
| **?***word* | Search backwards for *word* in the work area |
| **r** | Replace the character under the cursor with another |
| **R** | Replace several characters one after another as you type |

| | |
|---|---|
| **s** | Replace one character with several |
| **S** | Replace an entire line (same as **cc**) |
| **:n,Ns/old/new/g** | Substitute *new* for *old* in every occurrence from line *n* to line *N*, searching forward in the work area. |
| **:n,Ns?old?new?g** | Substitute *new* for *old* in every occurrence from line *n* to line *N*, searching backwards in the work area. |

## C.7 Invoking the editor

| | |
|---|---|
| $ **vi** *file* | Begin an editing session with *file*, starting on the first line. |
| $ **vi** **+***line* *file* | Begin an editing session with *file*, starting on line *line.* |
| $ **vi** **+/***word***/** *file* | Begin an editing session with *file*, starting on the first line that contains *word.* |
| $ **vi** **+** *file* | Begin an editing session with *file*, starting on the last line. |
| $ **vi** **-r** *file* | Begin an editing session to recover *file* after a system failure. |
| $ **vi** **-R** *file* | Begin an editing session to make *file* read-only. |
| $ **vi** **-t** ***tag*** | Edit the file that contains *tag.* |
| $ **vi** **-x** *file* | Edit an encrypted file (or edit a cleartext file and have it encrypted when writing it to disk). |

## C.8 Exiting the editor

| | |
|---|---|
| **:wq** (RETURN) | *Write and Quit* —Write the text in the work area to a file and end the editing session. |
| **:x** (RETURN) | *Conditional Write and Quit*—Write the text in the work area only if there have been changes, then end the editing session (same as **ZZ**). |
| **:q!** (RETURN) | *Abandon Text* Abandon the text in the work area (along with any changes made to it) and end the editing session. |

## C.9 Moving or copying text

The general procedure for moving (or copying) text within the buffer is to move the cursor to the beginning of the text, delete (or yank) the text, move the cursor to the new location, and then put the text either in front of the cursor (**P**) or after the cursor (**p**). If you are moving text to another file, you must delete the text to a buffer (a-z) and then put it from the same buffer. You must begin editing the other file (**:e**) with no other operators intervening between the delete and the put. Some of the many variations of the basic procedure involve the following combinations:

| Delete | yank | text | delete | yank | text |
|---|---|---|---|---|---|
| **x** | | a character | *n***x** | | *n* characters |
| **dw** | **yw** | a word | **d***n***w** | **y***n***w** | *n* words |
| **d)** | y) | a sentence | **d***n***)** | **y***n***)** | *n* sentences |
| **d**} | **y**} | a paragraph | **d***n*} | **y***n*} | *n* paragraphs |
| **dd** | **yy** | a line | *n***dd** | *n***yy** | *n* lines |
| **P** | Before or above | | "*x***P** | Before or above from buffer *x* | |
| **p** | After or below | | "*x***p** | After or below from buffer *x* | |

## Customizing vi

You can customize **vi** by setting options, assigning abbreviations, and assigning command sequences to keys on the keyboard.

## Setting options **set**

| | |
|---|---|
| **:set aw** | Set auto-write mode |
| **:set report=3** | Set **report** to 3 |
| **:set dir=files** | Set your directory to **files** |

## Assigning abbreviations **ab**

| | |
|---|---|
| **:ab unx UNIX operating system** | Assign `UNIX operating system` to the abbreviation `unx`. |

## Defining keys **map**

| | |
|---|---|
| **:map** * { >} | Assign the sequence { >} to the * key. |

## Making assignments permanent

To make assignments permanent, place the commands (without colons) in the **.exrc** file, which will be read each time you call up **vi**. Here is a sampling of line entries you could place in **.exrc**:

```
set aw report=3 dir=files
ab xen XENIX operating system
map * { >}
```

# Appendix D

# Summary of vi Options

This appendix lists all **vi** (and **ex**) options that can be set with the **:set** command. The options are grouped as toggled, number-valued, and string-valued. For each option, you will find its name, abbreviation (if any), default setting, and description.

## D.1 Toggled options

These options may be set only as on or off. To turn an option off, type **no** in front of its name (or abbreviation).

### AUTOMATIC INDENTATION **ai**

The **autoindent** (**ai**) option allows you to achieve the indentation desired in structured programming. For each line entered during text entry mode, **vi** matches the indentation with the previous line, using the spacing between tab stops specified by the **shiftwidth** option. The only way to back the cursor up to the previous tab stop is to type CTRL-D.

When this mode is set, certain lines are processed in a different way:

1. A line with no characters in it is turned into a blank line.
2. The input for a line that begins with a caret (^), followed immediately by CTRL-D, is placed at the beginning of the line, but the indentation for the previous line is retained for the line that follows.
3. The input for a line that begins with a zero (0), followed immediately by CTRL-D, is placed at the beginning of the line, along with input for subsequent lines.

The **autoindent** option is not in effect for global commands. The default for **autoindent** is **noai**.

### AUTOMATIC PRINTING **ap**

The **autoprint** (**ap**) option causes the current line to be printed after any **ex** command that changes the buffer. This is the same as appending **p** to each command. The **ap** option applies only to the last of a sequence of commands on a single line, and is not in effect for global commands. The Default is **ap**.

## Automatic writing **aw**

The **autowrite** (**aw**) option causes the contents of the editing buffer to be written automatically to the current file if you have made any changes to it before giving one of the following commands:

| | | | |
|---|---|---|---|
| **:n** | Next | **:!** | Escape |
| **:rew** | Rewind | (CTRL-^) | Switch files |
| **:tag** | Tag | (CTRL-]) | Go to tag |

Default for **autowrite**: **noaw**.

## Beautifying text **bf**

The **beautify** (**bf**) option prevents all control characters except tab, newline, and formfeed from being entered into the editing buffer during text entry. The first time a backspace is discarded, there is a message. The **bf** option does not apply to command input. The Default is **nobf**.

## Compatibility with **ed** **edcompatible**

The **edcompatible** option retains the global (**g**) and confirm (**c**) suffixes during multiple substitutions, and causes the read (**r**) suffix to work like the **read** command, instead of like the repeat string (**&**) suffix. The Default is **noedcompatible**.

## Ignore case **ic**

The **ignorecase** (**ic**) option maps upper case characters in text to lower case for the purpose of matching regular expressions—except for characters enclosed within brackets. The Default is **noic**.

## Lisp operation **lisp**

The **lisp** option allows indentation appropriate for writing programs in the LISP language, and modifies the meanings of the following symbols to conform to standard LISP usage:

```
( )        { }        [[ ]]
```

The Default is **nolisp**.

### List all **list**

The **list** option displays text the way **ed**'s list command (**l**) does, with tab stops and newline characters shown on the screen as "^I" and "$"; **list** also folds any line longer than 72 characters. The Default is **nolist**.

### Special meanings for characters **magic**

The **magic** option allows metacharacters to retain their special meanings. When this option is turned off, only the caret (^) and dollar sign ($) retain their special meanings. However, you can still evoke the special meanings of other special characters by preceding each with a backslash (\). The Default is **magic**.

### Allowing messages **mesg**

The **mesg** option allows other UNIX users to write to your screen with the UNIX **write** command while you are using **vi**. The usual setting **nomesg** prevents others from interfering with your screen display during editing. The Default is **nomesg**.

### Line-numbering **nu**

The **number** option displays line numbers with text. The Default is **nonu**.

### Optimizing output **optimize**

The **optimize** option is used to speed up the operation of terminals that lack cursor-addressing by eliminating automatic returns when displaying more than one line. This is especially helpful if the text contains quite a bit of indentation at the start of each line. The Default is **optimize**.

### Redrawing the screen **redraw**

The **redraw** option allows **vi** to simulate the operation of an intelligent terminal on a dumb terminal by redrawing the screen each time a deletion takes place and refreshing text continuously during insertion. Since this option requires considerable processing, it is useful only at very high transmission speeds. The Default is **noredraw**.

### Displaying matching symbol **sm**

The **showmatch** option allows you to display the matching symbol of a pair of parentheses or braces. When you type **)** or }, for example, the cursor moves to the matching **(** or { and waits for one second before returning—if the matching symbol is on the screen. The Default is **nosm**.

### Allowing brief messages **terse**

The **terse** option allows an experienced user to revert to briefer messages on the screen (for example, `15/297` instead of `15 lines 297 characters`). The Default is **noterse**.

### Warning to save **warn**

The **warn** option makes **ex** give a warning before allowing you to execute a shell command via the escape command (**!**), for instance, [No write since last change]. The Default is **warn**.

### Allowing wrapping during searches **ws**

The **wrapscan** (**ws**) option allows a search for a string to wrap from the end of the editing buffer to the beginning, rather than stop at the end. The Default is **ws**.

### Allowing any write **wa**

The **writeany** (**wa**) option allows you to write to any file allowed by UNIX file access permissions, inhibiting the checks usually made before executing a **write** command. The Default is **nowa**.

## D.2 Numbered options

These options are set to numerical values, using an equal sign to indicate the assignment.

### Hardware tab stops **ht**

The **hardtabs** (**ht**) option indicate the spacing between terminal hardware tab stops. The Default is **ht=8**.

### Report on commands **report**

The **report** option indicates the number of repetitions of a command needed to cause a message to be displayed on the screen. For commands that may have extensive consequences (global commands and the undo command), the message is saved until completion of execution. The Default is **report=5**.

### Lines to be scrolled **scroll**

The **scroll** option indicates the number of logical lines to be scrolled during execution of scroll down (CTRL-D) and scroll up (CTRL-U). The Default is **scroll=1/2 window**.

### Software tab stops **sw**

The **shiftwidth** (**sw**) option indicates the number of spaces between software tab stops, which is used by the shift commands, **autoindent**, and CTRL-D for indenting text and moving back to previous indentations. The Default is **sw=8**.

### Editor tab stops **ts**

The **tabstops** (**ts**) option indicates the amount of space between tab stops when a file is displayed on the screen. The Default is **ts=8**.

### Tag length **tl**

The **taglength** (**tl**) option indicates how many characters in a tag name are significant. A value of zero (0) means that all characters in the name are significant. The Default is **tl=0**.

### Window height **window**

The **window** option indicates the number of lines to be displayed in one window of text, with the default value based on the transmission speed to the terminal. The defaults are **window=8** (up to 600 bit/s), **window=16** (1200 bit/s), **window=23** (9600 bit/s). The variables **w300, w1200, w9600** may be used in place of the numbers **8**, **16**, and **23**.

### Wrap margin **wm**

The **wrapmargin** (**wm**) option indicates the margin for automatic word wrapping to the following line. A value of zero (0) means no margin (and therefore no wrapping). The Default is **wm=0**.

## D.3 String-valued options

These options are set to string values using an equal sign to indicate the assignment.

## Directory **dir**

The **directory** (**dir**) option names the directory that is to contain the editing buffer. If this directory is write-protected, **vi** will exit immediately, since it won't be able to maintain an editing buffer. The Default is **dir=/tmp**.

## Paragraphs **para**

The **paragraphs** (**para**) option gives the delimiters for the "beginning of paragraph" (/pg {) and "end of paragraph" (}) commands, using the names of **nroff** macros for starting paragraphs. The Default is **para=IPLPPPQPP LIbp** .

Note: The default string represents an abbreviation for seven macros—four **ms** macros (**.IP**, **.LP**, **.PP**, **.QP**, and **.PP**), two **mm** macros (**.P** and **.LI**), and one **nroff** macro (**.bp**).

## Sections **sections**

The **sections** option gives the delimiters for the "beginning of section" (**[[**) and "end of section" (**]]**) commands, using the names of **nroff** macros for starting paragraphs. The Default is **sections=SHNHH HU**.

Note: The default string represents an abbreviation for four macros—two **ms** macros (**.SH**, and **.NH**) and two **mm** macros (**.H** and **.HU**).

## Shell **sh**

The **shell** (**sh**) option gives the pathname of the shell to be used whenever you execute a shell escape command (**!**) or a **shell** command. If **SHELL** has been assigned a value, then this becomes the default. The Default is **sh=/bin/sh**.

## Tags **tags**

The **tags** option gives the pathnames of files to be used as tag files by the **tag** command. Any time a tag is requested, this tag is searched for sequentially in the files designated. The Default is **tags=/usr/lib** and the current directory.

## Terminal **term**

The **term** option gives the type of the terminal. The Default is **term=$TERM** (the value of the shell variable **TERM**).

# Appendix E

# Summary of Processing Commands

## E.1 Searching with **grep**

Use **grep** to search for text in a file (or a series of files), using a command line like this:

```
$ grep [- options]  pattern  file(s)
```

The pattern can be literal strings, regular expressions, or both.

### Options for **grep**

**-b** Precede each matched line with a block number  
**-c** Show how many times the string was found, but not the text  
**-i** Ignore case during comparisons  
**-l** Show the name(s) of the file(s) in which the string was found, but not the text  
**-n** Show line numbers along with the text  
**-s** Suppress any file error messages that may arise during processing  
**-v** Invert (match those lines in which the string was *not* found)

### Additional options for **fgrep**

**-e** *string* Match a string that begins with a hyphen  
**-f** *file* Read strings from *file*  
**-x** Match only entire lines

### Additional options for **egrep**

**-e** *expr* Match an expression that begins with a hyphen  
**-f** *file* Read expressions from *file*

Each program can match

**fgrep** Literal strings only  
**grep** Patterns (strings and regular expressions)

**egrep** Compound expressions

## E.2 Sorting with **sort**

Use **sort** to sort lines of text in a file (or series of files), using a command line like this:

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

### Options for **sort**

**-b** Ignore leading blanks (spaces and tabs) when comparing fields—useful if items in a field vary in length
**-n** Sort a numeric field (allow for optional blanks, minus signs, zeroes, or decimal points)—this option includes **-b** automatically
**-M** Sort a field that contains months (`JAN`, `FEB`, ..., `DEC`)
**-r** Sort in reverse order (`Z-A`, `z-a`, `9-0`)

**-f** Fold upper case letters onto lower case (for example, treat `CASE`, `Case`, and `case` as identical when sorting)
**-d** Sort in dictionary order
**-i** Ignore non-printing characters when sorting
**-u** Sort uniquely (if lines are identical, discard all but the first)
**-m** Merge several files: the files are already sorted; just merge them
**-c** Make sure the input file has already been sorted
**-t** *x* Make *x* (any character) the field separator
**-y** *k* Set aside *k* kilobytes of memory for this sort
**-z** *n* Allow no more than *n* characters per line of input
**-o** *file* Place the output in *file*—the same as >*file*

## E.3 Programming with **awk**

Use **awk** to program output from an input file (or series of files), using a command line like this:

```
$ awk pattern { action } file(s)
```

The *pattern* selects lines to be processed; the *action* specifies the processing to be performed. Fields in the input file(s) are designated $1, $2, $3, and so on; while an entire record is designated $0.

## BUILT-IN VARIABLES USED BY **awk**

| Variable | Description | Default |
|---|---|---|
| **$1-$n** | Individual fields | None |
| **NF** | Number of fields in a record | None |
| **FS** | Input field separator | Space or TAB |
| **OFS** | Output field separator | Space |
| **$0** | The current record | None |
| **NR** | Number of the current record | None |
| **RS** | Input record separator | Newline |
| **ORS** | Output record separator | Newline |
| **FILENAME** | Name of the current input file | None |

## PRINTING

You can display or print the output using either the plain **print** command or the more sophisticated **printf** command, which allows custom formatting.

## SEARCH PATTERNS

To match lines for processing, you have available pre-processing (**BEGIN**), post-processing (**END**), the standard regular expressions (Appendix A, "Summary of Basic Commands and Symbols"), arithmetic and relational operators, ranges, and compound statements. To match an expression with a particular field, **awk** uses tilde for match (˜) and exclamation-tilde for no-match (**!**˜).

## OPERATORS

| | | | |
|---|---|---|---|
| **+** | Add | ***** | Multiply |
| **-** | Subtract | **/** | Divide |
| | | **%** | Take remainder |
| < | Less than | > | Greater than |
| <**=** | Less than or equal to | >**=** | Greater than or equal to |
| **==** | Equal to | **!=** | Not equal to |
| **++** | Increment the variable | **–** | Decrement the variable |
| **+=** | Add and assign | **-=** | Subtract and assign |
| ***=** | Multipy and assign | **/=** | Divide and assign |
| | | **%=** | Take remainder and assign |
| **\|\|** | OR | **&&** | AND |
| **!** | NOT | | |

## Action statements

To construct action statements to perform processing, you can use built-in functions, variables (built-in and user-defined), arrays, and program control statements.

### Built-in functions used by **awk**

**length[$i]** Without an argument, returns the length of the current record; with an argument, returns the length of the field named as an argument

**split(string,array[,fs])** Split string *string* into *n* fields, place them in separate array elements *array[1]*, *array[2]*, ..., *array[n]*, and return the number of elements *n*; if the optional field separator *fs* is omitted, then the value of **FS** becomes the default.

**substr(string,p[,max])** Returns the number of elements in the substring of *string* that begins at position *p* and contains no more than *max* characters

**index(***STRING*,*string***)** Returns the position in *STRING* where *string* begins (or 0 if *string* doesn't occur)

**sprintf(***format*,$ex_1$,$ex_2$,...**)** Returns the value of the expressions named in **printf** format statement *format*

**sqrt(n)** Returns the square root of *n*

**log(n)** Returns the natural logarithm of *n*

**exp(n)** Returns the value of **e** to the power *n*

**int(n)** Returns the truncated integral value of *n*

### Directives used by **awk**

**next** Proceed to the next record and begin looking for a match

**continue** Proceed to the next iteration of the current loop

**break** Exit from the current **while** or **for** loop immediately

**exit** Proceed as if the last record had just been read

# Appendix F

# Summary of Formatting Requests

## MM REQUESTS

### FORMING PARAGRAPHS

| | | | |
|---|---|---|---|
| **.P 0** | Block paragraph | **.P 1** | Indented paragraph |

### FORMING LISTS

| | | | | |
|---|---|---|---|---|
| **.BL** | Begin bullet list | | | |
| **.DL** | Begin dash list | | | |
| **.ML** *mark* | Begin marked list | **.AL** | **1**<br>**A**<br>**a**<br>**I**<br>**i** | Begin auto-numbered list |
| **.RL** | Begin reference list | | | |
| **.VL n** | Begin variable-item list | | | |
| **.LI** | Item in the list | **.LE** | | End of list |

### DISPLAYING TEXT

| | | | |
|---|---|---|---|
| **.DS** | Standard display | **.DS I** | Indented display |
| **.DS C** | Centered display | **.DS CB** | Blocked (centered) display |
| | | **.DE** | End of display |

### EMPHASIZING TEXT

| | | | |
|---|---|---|---|
| **.I** | Underlining (italic) | **.R** | End of underlining (roman) |
| **.B** | Bold | **.R** | End of bold (roman) |

### JUSTIFYING THE RIGHT MARGIN

| | | | |
|---|---|---|---|
| **.SA 1** | Justify | **.SA 0** | Unjustify |

### Skipping lines

**.SP** *n* Skip *n* blank lines

### Keeping lines together

| | | | |
|---|---|---|---|
| **.DS** | Standard display | **.DF** | Floating display |

### Using footnotes

| | | | |
|---|---|---|---|
| **.FS** | Start of footnote | **.FE** | End of footnote |
| ***F** | Use auto-numbering | | |

### Using headings

| | | | |
|---|---|---|---|
| **.HU** | Unnumbered heading | **.H** *n* | Numbered heading, level *n* |

### Table of contents

**.TC** Produce a table of contents (placed after all text)

### Page layout

| | | |
|---|---|---|
| **.ND** Change the date | ***(DT (date)** | |
| | \\\\**nP (page number)** | |
| **.PH ”’left’center’right’”** | | for all pages |
| **.EH ”’left’center’right’”** | Change the header | for even pages |
| **.OH ”’left’center’right’”** | | for odd pages |
| **.PF ”’left’center’right’”** | | for all pages |
| **.EF ”’left’center’right’”** | Change the footer | for even pages |
| **.OF ”’left’center’right’”** | | for odd pages |
| **.SK** Skip a page | | |
| **.OP** Skip to an odd page | | |

### The **mm** command line

```
$ mm [- options]  file(s) [ | pg ] [ | lp ]
```

| | | | |
|---|---|---|---|
| **-E** | Equal spacing | **-12** | Pica output |
| **-t** | Include tables | **-y** | No compact files |
| **-e** | Include equations | **-T***name* | Name the printer |
| **-c** | Filter columns | | |

## Nroff and troff requests

### Page Adjustment

| | | | |
|---|---|---|---|
| **.pl** | Page length | **.po** | Page offset |
| **.ll** | Line length | **.in** *n* | Indent *n* characters |
| **.pn** | Page number | **.ti** *n* | Temporary indentation |
| **.bp** | Page break | **.ne** | Need vertical space |

### Displaying lines

| | | | |
|---|---|---|---|
| **.fi** | Turn on filling | **.nf** | Turn off filling |
| **.ad** | Adjust text | **.na** | No adjustment |
| **.hy** *n* | Hyphenation | **.nh** | No hyphenation |
| **.br** | Break to a new page | **.ls** *n* | Line spacing *n* |
| **.ce** | Center a line | **.ce** *n* | Center next *n* lines |

### Displaying characters

| | | | |
|---|---|---|---|
| **.ul** | Underline | **.cu** | Continuous underline |
| | | **u** | Superscript |
| **%** | Page number | **d** | Subscript |

### Column format

| Setup | | Execution |
|---|---|---|
| `.1C` | Single column | $ ***mm -c -Tlp file / lp*** |
| `.2C` | Double column | $ ***nroff -cm -Tlp file / col / lp*** |

### Formatting tables

| Setup | | Execution |
|---|---|---|
| `.TS` | [Start of table] | |

| | | |
|---|---|---|
| `over-all;` | | |
| `heading format` | | $ ***mm -t -c -Tlp \| lp*** |
| `body format.` | | |
| `headings` | | $ ***tbl file \| nroff -cm \| col \| lp*** |
| `Body of table` | | |
| `.TE` | [End of table] | |

## Formatting equations

| | Setup | Execution |
|---|---|---|
| `.EQ` | [Start of equation] | $ ***mm -e -c -Tlp \| lp*** |
| *statements* | | $ ***neqn /usr/pub/eqnchar file \| lp*** |
| `.EN` | [End of equation] | $ **nroff -cm \| col \| lp** |

## Writing macros

| | |
|---|---|
| `.de` *name* | Start of macro definition for *name* (one or two characters) |
| *statements* | |
| `..` | End of macro |
| `.` *name* | Execute macro *name* |
| `.wh` *n mm* | Activate macro *mm* at page location *n* |

## Troff requests

| | |
|---|---|
| `.ps` *n* | Change the point size |
| `\f` *n* | |
| `.vs` *n* | Change the vertical spacing |
| `.ss` *n* | Change the word spacing |
| `.cs` *f n* | Change to constant character spacing |
| `.ft` *n* | Select a font |
| `\f` *n* | |
| `.fp` *n f* | Change font positions |
| `.tl 'L'C'R'` | Print title line using left, center, and right positions |

# Appendix G

# Summary of Formatting Options

## G.1 Modifying **mm**

### PAGE LAYOUT AND GENERAL SETTINGS

Number Registers

| | | |
|---|---|---|
| **L** | Length of page | `66` (66 lines) |
| **W** | Width of page | `6i` (Six inches) |
| **O** | Offset of page | `.75i` (3/4 inch) |

| | | |
|---|---|---|
| **N** | Numbering style | 0 |
| **P** | Page number manager | 0 |
| **T** | Type of **nroff** device | 0 |

Strings

| | | |
|---|---|---|
| **DT** | Today's date | *mm dd*, *yy* format |
| **EM** | Em dash | – |
| **Tm** | Trademark symbol | TM (superscript) |

| | | | |
|---|---|---|---|
| ‘ | Grave accent | , | Cedilla |
| ’ | Acute accent | ; | Umlaut over upper case |
| ˆ | Circumflex accent | : | Umlaut over lower case |
| ˜ | Tilde | | |

## Paragraphs

Number registers

| | | |
|---|---|---|
| **Pt** | Paragraph type | 0 (Left-justified), 1 (Indented) |
| **Ps** | Paragraph spacing | 1 (Single-spacing) |
| **Pi** | Paragraph indentation | 5 (Five spaces) |
| | | |
| **Np** | Numbering of paragraphs | 0 (No numbering) |
| **Hy** | Hyphenation | 0 (Turn off hyphenation) |

## Lists

Number registers

| | | |
|---|---|---|
| **Ls** | List spacing | 6 (Spacing at all levels) |
| **Li** | List indentation | 6 (Six spaces from the margin) |
| | | |
| | Strings | |
| **BU** | Bullet | ⊕ `(Plus sign over lower case o)` |

## Displays

Number registers

| | | |
|---|---|---|
| **Ds** | Display spacing | 1 (One line before and after) |
| **Si** | Standard indentation | 5 (Five spaces from the margin) |

## Floating Displays

Number registers

0 (Move to end of section or document)

| | | |
|---|---|---|
| | | 1 (Current page if possible, or 0) |
| | | 2 (Only one display to top of page) |
| **Df** | Display format | 3 (One display on current page if possible; otherwise, move to the top of the next page) |
| | | 4 (As many displays as space permits at the top of the next page) |

| | | |
|---|---|---|
| | | 5 (Current page if possible; otherwise, as many displays as space permits at top of next page) |
| **De** | Display eject | 0 (No action)<br>1 (Eject a page automatically) |

## Footnotes

Number register

| | | |
|---|---|---|
| **Fs** | Footnote spacing | 1 (One line before and after) |

String

| | |
|---|---|
| **F** | Footnote numberer (Superscript) |

## Headings

Number registers

| | | |
|---|---|---|
| **Hu** | Heading level (.HU only) | 2 (Level 2 by default) |
| **Ht** | Heading type (.H only) | 0 (Numbers concatenated) |
| **U** | Underlining style | 0 (Continuous underlining) |
| **H1-7** | Heading counters | 0 (Start at zero) |
| **Hs** | Heading spacing level | 2 (Double after level 1 or 2) |
| **Hi** | Heading temporary indentation | 1 (Indent as a paragraph) |
| **Hb** | Heading break level | |
| **Hc** | Heading centering level | 0 (No centered headings) |
| **Ej** | Page eject for headings | 0 (None), 1 (Before level 1) |

Strings

| | | |
|---|---|---|
| **HF** | Heading font list | 3 3 2 2 2 2 2 (Levels 1 and 2 bold, levels 3–7 underlined) |

## Table of Contents

Options for request

**.TC** [*sl*] [*sp*] [*tl*] [*tb*] 1 1 2 0 (See lines following)

Precede each heading of level *sl* or higher with *sp* blank lines; right-justify the page number of each heading of level *tl* or higher, preceding the page number with either dots (0) or spaces (1)

Number Registers

| | | |
|---|---|---|
| **Cl** | Contents level | 2 (Include all level 1 and 2) |
| **Oc** | Page-numbering style | 0 (Lower case Roman numerals) |
| **Lf** | Include a list of figures | 1 (Yes), 0 (No) |
| **Lt** | Include a list of tables | 1 (Yes), 0 (No) |
| **Lx** | Include a list of exhibits | 1 (Yes), 0 (No) |
| **Le** | Include a list of equations | 0 (No), 1 (Yes) |

Strings

| | | |
|---|---|---|
| **Ci** | Indent list | `0m 0m 0m 0m 0m 0m 0m`<br>(no indentation) |
| **Lf** | Title for list of figures | `LIST OF FIGURES` |
| **Lt** | Title for list of tables | `LIST OF TABLES` |
| **Lx** | Title for list of exhibits | `LIST OF EXHIBITS` |
| **Le** | Title for list of equations | `LIST OF EQUATIONS` |

### Figures, Tables, and Equations

Number registers

| | | |
|---|---|---|
| **Of** | Figure caption style | 0 (Period separates description) |
| **Fg** | Figure counter (.FG) | 0 (Start at zero) |
| **Cp** | Placement of list of figures | 1 (Separate page) |
| **Tb** | Table counter (.TB) | 0 (Start at zero) |
| **Ec** | Equation counter (.EC) | 0 (Start at zero) |
| **Eq** | Equation label placement | 0 (Right adjusted) |

## G.2 Modifying **nroff**

### General Number Registers

| | |
|---|---|
| **%** | Current page number |
| **c.** | Current line number (same as **.c**) |
| **hp** | Current horizontal position on current line |
| **ln** | Output line number |

| | |
|---|---|
| **nl** | Vertical position on most recent output line |
| **dw** | Current day of the week (1–7) |
| **dy** | Current day of the month (1–31) |
| **mo** | Current month (1–12) |
| **yr** | Last two digits of current year (00–99) |

## Read-Only Number Registers

| | |
|---|---|
| **.F** | Name of current input file |
| **.c** | Current line number in the input file |
| **.H** | Horizontal resolution in basic units |
| **.V** | Vertical resolution in basic units |
| **.P** | 0 if the current page has not been selected to be printed (**-o**)<br>1 if the current page has been selected to be printed (**-o** option) |
| **.T** | 0 if the **-T** option is not in effect<br>1 if the **-T** option is in effect |
| **.p** | Current page length |
| **.o** | Current page offset |
| **.l** | Current line length |
| **.i** | Current indentation |
| **.h** | High-water mark on the current page |
| **.u** | 0 in no-fill mode (**.nf**)<br>1 in fill mode (**.fi**) |
| **.j** | Current adjustment mode and type (**.ad**) |
| **.n** | Length of text in the previous output line |
| **.k** | Length of text in the current output line |
| **.v** | Current vertical line spacing |
| **.L** | Current line spacing (**.ls**) |

# Appendix H

# Summary of the Bourne Shell

## H.1 Shell variables

These variables may be set in the login file (**.profile**):

| Bourne and C shells | | Bourne shell only | |
|---|---|---|---|
| **HOME** | Login directory | **PS1** | Primary prompt |
| **MAIL** | Mail file | **PS2** | Secondary prompt |
| **PATH** | Command file search path | **IFS** | Internal file separator |
| **TERM** | Terminal type | **TZ** | Time zone |
| | | **LOGNAME** | Login name |

## H.2 Standard input, output, and diagnostics

| | | |
|---|---|---|
| Standard input | File descriptor 0 | From terminal |
| Standard output | File descriptor 1 | To terminal |
| Standard diagnostic output | File descriptor 2 | To terminal |

| | |
|---|---|
| < | Redirect command input |
| > | Redirect command output |
| >> | Append to an output file |

## H.3 Background commands

| | |
|---|---|
| **&** | Run a process in the background |
| **ps** | Monitor the status of background processes |
| **nice** | Change the priority of a process |
| **kill** *nnnn* | Terminate background process number *nnnn* |
| **shl** | Begin a session with the shell layer manager |

## H.4 Connecting processes

*p1* | *p2*    Pipeline: Connect *p1* to *p2* with a pipeline

| | |
|---|---|
| *p1* \| tee *file* | Tee: View the output of *p1* as it is being written to *file* |

## H.5 Giving directives to the shell

### Command grouping

| | |
|---|---|
| (*p1 p2*) | Create a subshell to read commands *p1* and *p2* |
| {*p1 p2*} | Cause the shell to read commands *p1* and *p2* |
| *p1*;*p2*;*p3* | Command list with three commands |

### Conditional execution

| | |
|---|---|
| *p1* \|\| *p2* | OR-IF: Invoke command *p2* only if *p1* fails |
| *p1* && *p2* | AND-IF: Invoke *p2* only if *p1* succeeds |

### Pattern matching

| | |
|---|---|
| * | Match any string of any length |
| ? | Match any single character |
| [ ] | Match any of the characters enclosed |

## H.6 Shell procedures

### Preliminary commands

| | |
|---|---|
| **echo** | Display text at the standard output (terminal screen) |
| **read** | Receive text from the standard input (terminal keyboard) |

### Shell variables

| | |
|---|---|
| $ *var=string* | Assign *string* to the variable *var* |
| **$ *echo* *$var*** | Display the contents of the variable |
| ' ' | Use single quotes for a string that contains spaces |
| " " | Use double quotes for a string that also contains special characters whose meanings need to be retained |
| **eval** | Evaluate a command string |
| {**$***var*}**s** | Distinguish variable *var* from the surrounding text |

### Command substitution

| | |
|---|---|
| **$ *files='ls'*** | Assign the result of the **ls** command to **files** |

| | |
|---|---|
| **$ *y='expr $x + 5*** | Evaluate the arithmetic expression and assign the result to **y** |

CONDITIONAL SUBSTITUTION

| | |
|---|---|
| **$**{*var*-*sub*} | If *var* has a value, use it; if not, use *sub* and leave *var* unassigned |
| **$**{*var*=*sub*} | If *var* has a value, use it; if not, assign the value of *sub* to *var* |
| **$**{*var*?*sub*} | If *var* has a value, use it; if not, display the message *var*:*sub* and abort the process |
| **$**{*var*+*sub*} | If *var* has a value, use the value of *sub* instead; if not, use a null string and leave *var* unassigned |

POSITIONAL PARAMETERS

| | | | |
|---|---|---|---|
| **$0** | Command | **$2** | Second argument |
| **$1** | First argument | **$3** | Third argument |

RESERVED SHELL VARIABLES

| | |
|---|---|
| **#** | Number of arguments in a command line |
| **?** | Return code for the command |
| **$** | PID of the current process |
| **!** | PID of the most recent background process |
| **-** | Status of shell flags |

## H.7 Constructing loops

**Looping While True**

```
while command list
do [Execute commands
     between do and done]
done
```

**Nesting Loops**

```
while command list
do [Execute commands
     between do and done]
     until command list
     do [execute commands
          between do and done]
     done
```

**Looping While False**

```
until command list
do [Execute commands
     between do and done]
done
```

**Looping on a Variable**

```
for variable in list
do [Execute commands
          between do and done
     done
```

**done**

### Looping statements

| | |
|---|---|
| **break** | Exit from a loop |
| **continue** | Resume execution at the start of the nearest loop |
| **exit** | Exit from the shell procedure |

## H.8 The conditional statement

```
if  expression                          if  expression
    then  command list                      then  command list
fi                                      else  command list
                                        fi
elif = else if
```

**test**
**[ ]**

### Testing files, strings, and quantities

### Testing Files

| | |
|---|---|
| **[ -r** *file]* | [File exists and can be read] |
| **[ -w** *file]* | [File exists and can be written to] |
| **[ -f** *file]* | [File exists and is **not** a directory] |
| **[ -d** *file]* | [File exists and **is** a directory] |
| **[ -s** *file]* | [File is not empty—its size is non-zero] |

### Comparing quantities

| | |
|---|---|
| **[***A* **-eq** *B***]** | [Is A equal to B?] [$A = B$?] |
| **[***A* **-ne** *B***]** | [Is A not equal to B?] [$A \neq B$?] |
| **[***A* **-ge** *B***]** | [Is A greater than or equal to B?] [$A \geq B$?] |
| **[***A* **-le** *B***]** | [Is A less than or equal to B?] [$A \leq B$?] |
| **[***A* **-gt** *B***]** | [Is A greater than B?] [$A > B$?] |
| **[***A* **-lt** *B***]** | [Is A less than B?] [$A < B$?] |

### Comparing strings

| | |
|---|---|
| **[ -n** *string* **]** | [The string exists] |
| **[ -z** *string* **]** | [The string doesn't exist] |
| **[** *str1* **=** *str2* **]** | [The two strings are the same] |
| **[** *str1* **!=** *str2* **]** | [The two strings are not the same] |

### Compound testing

**!** NOT **-o** OR **-a** AND

## H.9 Other shell programming techniques

### Multiple branching

```
case string in
     S1) command list ;;
     S2) command list ;;
          :
     Sn) command list ;;
  esac
```

### Interrupt-Handling

```
Signal 1: Hangup
Signal 2: Quit signal    trap ' command list'   signals
Signal 3: Interrupt
Signal 15: Software termination
```

### Debugging a shell procedure

```
$ sh -x file [Execution option: display commands]
$ sh -v file [Verbose option: display input lines]
```

# Appendix I

# Summary of the C Shell

## I.1 Initialization files

A SAMPLE LOGIN FILE *.login*

| | |
|---|---|
| **stty** | Set up your terminal (tty) |
| **set path** | Set up your command search path |
| **setenv TERM** | Select a terminal name from those already described |
| **echo** | Select an opening display |

THE SHELL READ COMMAND FILE *.cshrc*

| | |
|---|---|
| **set prompt** | Set your prompt |
| **set history** | Set the number of events to be saved |
| **alias** | Set an alias |

## I.2 Reinvoking previous commands

| | |
|---|---|
| **!!** | Reinvoke the most recent event |
| **!17** | Reinvoke event number 17 |
| **!-4** | Reinvoke the fourth event back from the current event |
| **!ls** | Reinvoke the event that begins with **ls** |
| **!?s** | Reinvoke the event that contains s (anywhere in the line) |

## I.3 Selecting individual arguments

| | |
|---|---|
| **!18:4** | Select the fourth argument of event 18 |
| **!18:^** | Select the first argument of event 18 (also **!18^**) |
| **!18:$** | Select the last argument of event 18 (also **!18$**) |
| **!18:2-5** | Select arguments 2, 3, 4, and 5 of event 18 |
| **!18:*** | Select all arguments of event 18 (also **!18***) |

## I.4 Modifying a command line

| | |
|---|---|
| **!!:s/w/W** | Replace w with W in the previous event |
| **!!:&** | Repeat the previous substitution in the previous event |
| **!!:h** | Remove the last name from a pathname in the previous event |
| **!!:t** | Remove the prefix from a pathname in the previous event |
| **!!:r** | Remove the suffix from a pathname in the previous event |
| **!!:p** | Preview the previous event |
| **!!:q** | Protect the previous event from further modification |

## I.5 Assigning an alias to a command string

| | |
|---|---|
| **alias d pwd** | Use d as an abbreviation (alias) for **pwd** |
| **d** | Invoke **d** |
| **unalias d** | Remove **d** as an abbreviation (alias) |
| **ch chapter10** | Invoke **ch** with a real-time selection |
| **cdl /etc/termcap** | Invoke **cdl** as a multiple-command alias |

## I.6 The logout file

### Standard logout

| | |
|---|---|
| % **^D** | [Press CTRL-D] |

### Explicit logout

| | |
|---|---|
| set ignoreeof | (placed in **.login**) |
| % ***logout*** | |

## I.7 Assigning string values

| | |
|---|---|
| % ***set VAR = value*** | Assign value to variable **VAR** |
| % ***echo $VAR*** | Retrieve the value of **VAR** |
| % ***set*** | Display all declared variables and their values |
| % ***echo $path*** | |
| /usr/peter/bin /bin /usr/clan/bin | |
| % ***echo $path[2]*** | Display the second argument of **path** |
| /bin | |
| % ***set array = (" " )*** | Declare **array** with two elements |
| % ***set array[2] = two*** | Assign the second element |
| % ***echo $?array*** | Test whether **array** has been declared |

```
1                                 [True: array declared]
% echo $#array                    Display the number of elements of array
2
```

## I.8 Manipulating variables that contain numeric values

```
% @ val = 5          Assign 5 to the variable val
% echo $val          Display the value of val
5
% @                  Display all numeric variables and their assigned values
```

| | | | | | |
|---|---|---|---|---|---|
| + | Add | * | Multiply | | |
| - | Subtract | / | Divide | % | Remainder |

```
% @ Y++                           Increment Y by one
% @ X+=5                          Increment X by five

% @ Y--                           Decrement Y by one
% @ X-=3                          Decrement X by three

% @ X =~ $X                       Assign the one's complement of X to X

% @ M = 9; @ N = 10               Assign 9 to M and 10 to N
% @ T =($M < $N)                  Test whether M is less than N
```

| | | | | | |
|---|---|---|---|---|---|
| < | Less than | <= | Less than or equal to | == | Equal to |
| > | Greater than | >= | Greater than or equal to | != | Not equal to |

## I.9 Variables reserved by the C shell

| Description | Name | Type |
|---|---|---|
| Argument Number *n* | **$argv[*n*]** | String or numeric |
| Number of Arguments | **$#argv** | Numeric |
| Search Path | **$cdpath** | String |
| Child Process Number | **$child** | Numeric |
| Echo a Command | **$echo** | True or false |
| Length of History List | **$history** | Numeric |
| Home Directory | **$home** | String |
| Ignore End-of-File | **$ignoreeof** | True or false |
| Prevent Overwriting | **$noclobber** | True or false |
| Remove Special Meaning Automatically | **$noglob** | True or false |
| Remove Special Meaning If There Is No Match | **$nonomatch** | True or false |
| Search Path | **$path** | String |
| Prompt Symbol | **$prompt** | String |
| Pathname of the Shell | **$shell** | String |
| Exit Status of Last Command | **$status** | True or false |
| Time to Execute a Command | **$time** | String |
| Terminal Type | **$TERM** | String |
| Process Identification Number | **$$** | Numeric |
| Complete Messages | **$verbose** | True or false |

You can set some of these variables in your .cshrc file, which is executed each time you log on and each time a new process is created.

## I.10 File-checking

The C shell uses the following expressions in **if**, **foreach**, and **while** statements the way the Bourne shell uses **test** to performs similar tests:

| | |
|---|---|
| **-e** *file* | True if the file exists |
| **-z** *file* | True if the size of the file is zero |
| **-f** *file* | True if the file is an ordinary file |
| **-d** *file* | True if the file is a directory |
| **-o** *file* | True if the user owns the file |
| **-r** *file* | True if the file is readable |
| **-w** *file* | True if the file is writable |
| **-x** *file* | True if the file is executable |

## I.11 Forming conditional statements

**if** . . . **then**
**if** . . . **then** . . . **else**
**if** . . . **then** . . . **else** . . . **else if**

## I.12 Forming loops

| | |
|---|---|
| **goto** *label* | Branch to the line labeled *label* |

```
foreach variable ( wordlist)
  command(s)
end

while ( expression)
  command(s)
end
```

| | |
|---|---|
| **break** | Exit from a loop |
| **continue** | Continue execution at the start of the nearest loop |
| **exit** | Exit from the shell procedure |

## I.13 Other programming techniques

```
switch ( string)
case pattern1:
    command(s)
breaksw
case pattern2:
    command(s)
breaksw
default:                [optional]
    command(s)
breaksw
endsw
```

| | |
|---|---|
| **shift** | Shift all arguments to the left one position |
| **onintr** *label* | Branch to the line labeled *label* if an interrupt occurs |

## I.14 Built-in commands

Built-in commands are commands whose code is part of the C shell. When a built-in command is invoked, the C shell doesn't need to use au command search path to locate this command, since the command isn't in a separate file. The C shell gives control to an invoked built-in command without creating a new process. The following is a list of built-in commands. Most of them are explained in greater detail in Chapters 25–27.

**@**<br>
**alias**<br>
**break**<br>
**breaksw**<br>
**case**<br>
**cd (chdir)**<br>
**continue**<br>
**default**<br>
**echo**<br>
**else**<br>
**end**<br>
**endsw**<br>
**exec**<br>
**exit**<br>
**foreach**<br>
**goto**<br>
**history**<br>
**if**<br>
**login**<br>
**logout**<br>
**set**<br>
**setenv**<br>
**shift**<br>
**source**<br>
**switch**<br>
**time**<br>
**unalias**<br>
**unset**<br>
**while**

# Appendix J

# Summary of System Administration

## J.1 Basic commands and files

| | |
|---|---|
| /etc | Directory for administrative commands and files |
| root | Login name from outside of UNIX |
| su | Login name from inside UNIX |
| /etc/motd | File for the message of the day |
| /etc/news | File for news items |
| **wall** | Command to write directly to all users |

## J.2 File systems

Contents of the *super block*:

1. Size of the file system in blocks
2. The name of the file system
3. The name of the volume
4. The time of the last update
5. The time of the last back-up
6. The number of blocks used for i-nodes
7. A list of free i-nodes
8. A list of free data blocks

Contents of each *i-node*:

1. The size of the file in blocks
2. The owner ID (UID)
3. The group ID (GID)
4. Permissions allowed
5. The time of creation
6. The time of last access
7. The time of last modification
8. The number of links to the file
9. Direct pointers to the file's actual data blocks
10. Indirect pointers (if required) to indirect blocks

**/etc/fsck** Program to find and correct errors

## J.3 Devices

/dev The directory that contains special (device) files

**ls** Command to list the contents of a directory

| | |
|---|---|
| **-a** | Show all entries |
| **-i** | Display i-numbers |
| **-s** | Display sizes in blocks |

File types:

| | | | | | |
|---|---|---|---|---|---|
| - | Ordinary | b | Block special | p | fifo (named pipe) |
| d | Directory | c | Character special | | |

**mknod** Command to create a device file in /dev

```
# mknod file name  file type  major device number  minor device number
```

**mkfs** Command to create a file system

```
# mkfs file name  number of blocks[:number of i-nodes]
```

**labelit** Command to create a label for a file system

```
# labelit file system name  file system identifier  volume identifier
```

**mount** Command to mount a file system

```
# mount device name  file system name
```

**umount** Command to unmount a file system

```
# umount device name
```

**tar** Command to copy files to and from tape (or disk)

```
$ tar  crutx[fbvlmw0-7] [file(s)]
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **c** | Create | **t** | Contents | **b** | Block | **m** | Modification |
| **r** | Write | **x** | Extract | **v** | Verbose | **w** | Confirm |
| **u** | Update | **f** | File | **l** | Links | **0-7** | Override drive number |

**cpio** Command to copy files to and from tape (or disk)

| | |
|---|---|
| **cpio -o[crtvuamBdf]** | Copy out |
| **cpio -i[crtvuamB]** *patterns* | Copy in |
| **cpio -p[crtvuamdl]** *directory* | Pass |

| | | | |
|---|---|---|---|
| **c** | Compatible | **v** | Verbose |
| **r** | Rename | **u** | Unconditional |
| **t** | Contents | **a** | Access time |
| **m** | Modification time | | |
| **B** | Block | **l** | Link |
| **d** | Directory | **f** | Reverse |

## J.4 Disk maintenance

**df** Command to display the number of free blocks available

```
     ⎧    ⎫               Display the number of free i-nodes
$ df ⎨ -t ⎬  name(s)      Display also the number of allocated blocks
     ⎩ -f ⎭               Dislay also the number of free blocks
```

**du** Command to display the number of blocks used by files

```
     ⎧ -s ⎫            Summary report: total number of blocks used only
$ du ⎨ -a ⎬  name(s)All report: blocks used by ordinary files also
     ⎩ -r ⎭            Error report: blocks used and troublesome files
```

**find** Command for finding files and taking action

```
      +---  the name of the command
      |                 +---  which files to select
      |                 |
  $ find  path(s) criteria action(s)
            |                |
            |                +---  what to do with the file(s)
            +---  the directory (or directories) to be searched
```

Criteria:

| | |
|---|---|
| **-name** *file* | Find file(s) named *file* |

$$\textbf{-type}\left\{\begin{matrix}\textbf{f}\\\textbf{d}\\\textbf{b}\\\textbf{c}\\\textbf{p}\end{matrix}\right\}\quad\text{Find file(s) of type}\left\{\begin{matrix}\text{ordinary file}\\\text{directory}\\\text{block device}\\\text{character device}\\\text{named pipe}\end{matrix}\right\}$$

| | |
|---|---|
| **-user** *name* | Find by user *name* |
| **-group** *name* | Find by group *name* |
| **-perm** *pppp* | Find by permission code *pppp* |
| **-newer** *file* | Find file(s) modified more recently than file *file* |

$$\textbf{-size}\left\{\begin{matrix}+b\\b\\-b\end{matrix}\right\}\quad\text{Find file(s) with}\left\{\begin{matrix}\text{more than } b \text{ blocks}\\\text{exactly } b \text{ blocks}\\\text{fewer than } b \text{ blocks}\end{matrix}\right\}$$

$$\textbf{-links}\left\{\begin{matrix}+l\\l\\-l\end{matrix}\right\}\quad\text{Find file(s) with}\left\{\begin{matrix}\text{more than } l \text{ links}\\\text{exactly } l \text{ links}\\\text{fewer than } l \text{ links}\end{matrix}\right\}$$

$$\left\{\begin{matrix}\textbf{-ctime}\\\textbf{-mtime}\\\textbf{-atime}\end{matrix}\right\}\left\{\begin{matrix}+d\\d\\-d\end{matrix}\right\}\text{Find file(s)}\left\{\begin{matrix}\text{created}\\\text{modified}\\\text{accessed}\end{matrix}\right\}\left\{\begin{matrix}\text{more than } d \text{ days ago}\\\text{exactly } d \text{ days ago}\\\text{fewer than } d \text{ days ago}\end{matrix}\right\}$$

Logical constructions:

| | |
|---|---|
| **Grouping:** | **(***option-1 option-2***)** *option-3* (1 and 2 before 3) |
| **AND:** | *option-1 option-2 option-3* (as in the examples above) |
| **OR:** | *option-1* **-o** *option-2 option-3* (1 or 2) |
| **NOT:** | *option-1 option-2* **!** *option-3* (not 3) |
| | **! (***option-1 option-2***)** *option-3* (not 1 and 2) |

Action statements:

| | |
|---|---|
| **-print** | Display names |
| **-exec** *command* | Execute unconditionally |

| | |
|---|---|
| **-ok** *command* | Execute with confirmation |
| **-depth** | Used before **-cpio**, include directory entries |
| **-cpio** *device* | Copy to *device* using **cpio** format. |
| **update** | Daemon to update the disk every thirty seconds |
| **cron** | Daemon to perform tasks automatically on a timer<br>Fields: minute, hour, day, month, day of week, command line. |

## J.5 Startup and shutdown

| | |
|---|---|
| /etc/rc | System initialization script carries out three steps:<br>1. Mounts devices (**mount**)<br>2. Performs disk maintenance (**fsck**)<br>3. Starts deamons (**update**, **cron**) |
| /etc/inittab | Processes to be run at each run level |
| /etc/gettydefs | Information about each terminal |
| /etc/passwd | Information about each user |
| /etc/profile | Initialization script for all users (Bourne shell) |
| .profile | Initialization script for one user (Bourne shell) |
| .cshrc | Initialization script for one user (C shell) |
| .login | Initialization script for one user (C shell) |
| /etc/shutdown | System shutdown script carries out five steps:<br>1. Warns users (**wall**)<br>2. Kills all processes (**killall**)<br>3. Concludes disk activity (**sync**)<br>4. Unmounts devices (**umount**)<br>5. Returns the system to single-user mode (**init s**) |

## J.6 Terminals and printers

| | |
|---|---|
| **stty** | Command to set terminal operation (too many options) |
| /etc/termcap | File that contains terminal descriptions |
| /usr/lib/terminfo/*/* | Terminal descriptions (System V, Release 2) |

**tabs** Command to set tab stops:

| | |
|---|---|
| **-a** | IBM S/370 assembler |
| **-a2** | IBM S/370 assembler (second format) |
| **-c** | COBOL normal |
| **-c2** | COBOL compact |
| **-c3** | COBOL expanded compact |
| **-f** | FORTRAN |
| **-p** | PL/I |
| **-s** | SNOBOL |
| **-u** | UNIVAC |
| **-*n*** | Every *n*th column, starting at 1 |

/usr/lib

| | |
|---|---|
| **lpsched** | Start the printing scheduler (usually placed in **/etc/rc**) |
| **lpshut** | Stop the scheduler (usually placed in **/etc/shutdown**) |
| **accept** *ppp* | Allow **lp** to accept printing requests to printer *ppp* |
| **reject** *ppp* | Prevent **lp** from accepting printing requests to printer *ppp* |
| **lpmove** *ppp qqq* | Reroute printing requests from printer *ppp* to printer *qqq* |
| **enable** *ppp* | Allow the scheduler to rout requests to printer *ppp* |
| **disable** *ppp* | Prevent the scheduler from routing requests to printer *ppp* |
| **lpadmin** | Configure the **lp** spooling system; the scheduler must be idle; details in the following: |

```
                                     -m model
# lpadmin -p printer -v device -e printer
                                     -i program
```

| Variable | Function | Default Value |
|---|---|---|
| **ADMIN** | Login name of the **lp** administrator | **lp** |
| **GROUP** | Group that owns the **lp** commands and data | **bin** |
| **ADMDIR** | Directory for the administrator's commands | /usr/lib |
| **USRDIR** | Directory for user commands | /usr/bin |
| **SPOOL** | Directory for printer spooling | /usr/spool/lp |

## J.7 System security

**/etc/passwd** File that contains the following information about users:

1. Identifier
2. Encrypted password
3. Numerical user identifier
4. Numerical group identifier
5. Comments
6. Home directory
7. Login program

**/etc/group** File that contains the following information about groups:

1. Name of the group
2. Encrypted password
3. Numerical group identifier
4. Members of the group

**chmod** Command for granting permissions for files, with permissions in either numeric or symbolic form:

| | | |
|---|---|---|
| 4000 | Set user ID (UID) on execution | **u+s** |
| 2000 | Set group ID (GID) on execution | **g+s** |
| 1000 | Set the sticky bit (super-user) | **u+t** |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0400 | Allow read by owner | u+r | 0040 by group | g+r | 0004 by others | o+r |
| 0200 | Allow write by owner | u+w | 0020 by group | g+w | 0002 by others | o+w |
| 0100 | Allow execute* by owner | u+x | 0010 by group | g+x | 0001 by others | o+x |

* Execute a file or search a directory

$$\$\ \textbf{\textit{chmod}} \begin{Bmatrix} u \\ g \\ o \\ a \end{Bmatrix} \begin{Bmatrix} + \\ - \\ = \end{Bmatrix} \begin{Bmatrix} r \\ w \\ x \\ s \\ t \end{Bmatrix}$$

Symbolic command line

**umask** Command for setting default permissions:

| | | | | | |
|---|---|---|---|---|---|
| 0400 | Deny read by owner | 0040 | by group | 0004 | by others |
| 0200 | Deny write by owner | 0020 | by group | 0002 | by others |
| 0100 | Deny execute by owner | 0010 | by group | 0001 | by others |

## J.8 System accounting

| | |
|---|---|
| **acctcom** | Command to check system activity with eight fields |
| **startup** | Command to start process accounting |
| **runacct** | Command to produce command and usage summaries |
| **dodisk** | Command to produce usage files for **runacct** |
| **ckpacct** | Command to keep the accounting file from becoming overgrown |
| **monacct** | Command to generate a monthly summary |
| **shutacct** | Command to shut down process accounting |
| **timex** | Command to time a command and report on the system |
| **sar** | Command to report on the system in tabular format |
| **sag** | Command to report on the system in graphical format |
| **sadp** | Command to check disk activity every second |

```
        ┌-p x┐             [Number of characters and blocks transferred]
$ timex │-o  │  command    [Show process activity—see suboptions below]
        └-s  ┘             [Show all system activity during execution]
```

Process activity can be displayed using the following parameters:

| | |
|---|---|
| **-pr** | Display user time divided by (system time + user time) |
| **-pt** | Separate user time from system CPU time |
| **-ph** | Display CPU time divided by elapsed time |
| **-pm** | Display mean core size |
| **-pk** | Display Kcore-minutes |
| **-pf** | Display the fork/exec flag and exit status of the command |

```
      ┌-A┐                    [All system activity]
      │-a│                    [File access]
      │-b│                    [Buffers]
      │-c│                    [System calls]
      │-d│                    [Block devices]
$ sar │-m│ [-o file] t [ n]   [Messages and semaphores]
      │-q│                    [Queues]
      │-u│                    [CPU usage—the default]
      │-v│                    [Text, processes, i-nodes, and files]
      │-w│                    [Swapping and switching]
      └-y┘                    [Terminal devices]
```

| | |
|---|---|
| *t* | the length of each sample in seconds |
| *n* | the number of samples to take (default: 1) |

```
       ┌-t┐
$ sadp │  │ [-d device[- drive]] s [ n]
       └-h┘
```

The options are as follows:

**-t** Use tabular format—the default  
**-h** Use histogram format  
**-d** Device (and optional drive number)  
*s* Length of each sample in seconds  
*n* Number of times to repeat sampling

# Appendix K

# Network Administration

Here is a list of administrative commands and files that relate to communication and resource sharing.

## K.1 Communication before Release 3

| | |
|---|---|
| /usr/bin | Directory for **uucp**'s operational commands |
| /usr/lib/uucp | Directory for maintenance commands and daemons |
| /usr/lib/uucp/L-devices | Directory for identifying other systems:<br>`type line call-unit speed [protocol]` |
| /usr/bin/uucp | Command for requesting a file transfer |
| /usr/bin/uux | Command for requesting remote command execution |
| /usr/bin/uustat | Command for checking network status |
| /usr/bin/uuname | Command for listing the names of other systems |
| /usr/bin/uulog | Command for printing a history log |
| /usr/lib/uucp/uucico | Daemon that calls other systems |
| /usr/lib/uucp/uuxqt | Daemon that executes remote UNIX commands |
| /usr/lib/uucp/uuclean | Command for cleaning the spool directory |
| /usr/lib/uucp/uusub | Command for creating and monitoring networks |
| /etc/passwd | File for identifying your system to others |
| /usr/lib/uucp/L.sys | File for identifying other systems to yours:<br>`name time device speed phone login` |
| /usr/lib/uucp/L-dialcodes | File for storing phone abbreviations for L.sys |
| /usr/spool/uucp/LCK* | Directory of lock files (requests that failed) |
| /usr/spool/uucp/LOGFILE | Directory of log files (all transactions) |
| /usr/spool/uucp/SYSLOG | Directory of system log files |

| | |
|---|---|
| /usr/spool/uucp/ERRLOG | Directory of error log files |
| /usr/lib/uucp/USERFILE | File for controlling user access:<br>`login,system [c] path [path] ...` |
| /usr/lib/uucp/ORIGFILE | File for restricting forwarding of files:<br>`system[user1[,user2[,...]]]` |
| /usr/lib/uucp/FWDFILE | File for restricting forwarding of files:<br>`system[user1[,user2[,...]]]` |

```
     ┌-s s┐                       [Transmission speed s; default: 300]
     │-l l│                       [Line l; default in /usr/lib/uucp/L-devices]
     │-h  │                       [Emulate local echo for half-duplex]
     │-t  │                       [Auto-answer ASCII terminal; CR → CR LF]
$ cu │-d  │ phone | name | dir    [Print diagnostic traces]
     │-e  │                       [Generate even parity]
     │-o  │                       [Generate odd parity]
     │-m  │                       [Direct line with modem control]
     └-n  ┘                       [Manual dialing]
```

```
       ┌-d┐                [Create any directories required (default)]
       │-f│                [Do not create intermediate directories]
       │-c│                [Use the source file directly (default)]
       │-C│                [First copy the file to the spool directory]
$ uucp │-m│ source dest.   [Mail to requester upon completion]
       │-n│                [Notify the user named on the other system]
       │-e│                [Execute uucp on the system named]
       │-r│                [Queue the job, but don't transfer files]
       └-j┘                [Control writing of the uucp job number]
```

Notes: **-m** may be followed by a filename (**-m***file*); **-n** must be followed by a user identifier (**-n***user*); **-e** must be followed by the name of a system (**-e***system*).

```
      ┌  ┐
      │- │                 [Use standard input to uux for command(s)]
$ uux │-m│ | command(s)    [Mail to requester upon completion]
      │-n│                 [Do not notify the requester]
      └-j┘                 [Control writing of the uucp job number]
```

Note: **-m** may be followed by a filename (**-m***file*).

```
           -u u   [Report the status on requests from user u]
           -s s   [Report the status on requests from system s]
           -o h   [Report the status on requests older than h hours]
           -y h   [Report the status on requests younger than h hours]
           -c h   [Remove status entries older than h hours—uucp or root]
           -m m   [Report on accessibility of machine m (all for all)]
$ uustat   -M m   [Same as m, but give time of last status and transfer]
           -j j   [Report the status of request j (all for all)]
           -k j   [Kill request j, which must be owned by requester]
           -r j   [Rejuvenate request j to avoid deletion by uuclean]
           -q     [List number of jobs and files queued for each machine, time
                   of youngest and oldest file, date of any lock file         ]
           -O     [Report status using codes, as listed here]
```

| **Code** | **Meaning** |
|---|---|
| 000001 | Copying failed, reason unknown |
| 000002 | Permission to access local file denied |
| 000004 | Permission to access remote file denied |
| 000010 | The **uucp** command is incorrect |
| 000020 | Remote system is unable to create a temporary file |
| 000040 | Unable to copy to remote directory |
| 000100 | Unable to copy to local directory |
| 000200 | Local system is unable to create a temporary file |
| 000400 | Cannot execute **uucp** |
| 001000 | Copy succeeded partially |
| 002000 | Copy completed, job deleted |
| 004000 | Job is queued |
| 010000 | Job has been killed (incomplete) |
| 020000 | Job has been killed (complete) |

```
          -a s   [Add system s to the network]
          -d s   [Delete system s from the network]
          -c s   [Exercise connection to system s (all for all)]
$ uusub   -l     [Report statistics on connections]
          -f     [Flush statistics on connections]
          -r     [Report statistics on amount of traffic]
          -u h   [Collect statistics on traffic for the past h hours]
```

## K.2 Communication after Release 3

| | |
|---|---|
| /usr/bin | Directory for **uucp**'s operational commands |

| | |
|---|---|
| /usr/lib/uucp | Directory for maintenance commands and daemons |
| /usr/lib/uucp/Devices | Directory for identifying other systems:<br>`Type   Line   Line2   Class   Dialer,Token...` |
| /usr/bin/cu | Command for calling another UNIX system |
| /usr/bin/ctcp | Command for calling a terminal |
| /usr/bin/uucp | Command for requesting a file transfer |
| /usr/bin/uuto | Command for sending files to a spool file |
| /usr/bin/uupick | Command for retrieving files from a spool file |
| /usr/bin/uux | Command for requesting remote command execution |
| /usr/bin/uustat | Command for checking network status |
| /usr/bin/uuname | Command for listing the names of other systems |
| /usr/bin/uulog | Command for printing a history log |
| /usr/lib/uucp/uucico | Daemon that calls other systems |
| /usr/lib/uucp/uuxqt | Daemon that executes remote UNIX commands |
| /usr/lib/uucp/uusched | Daemon that schedules work queued |
| /usr/lib/uucp/uucleanup | Command for cleaning the spool directory |
| /usr/lib/uucp/Uutry | Command for testing and debugging |
| /usr/lib/uucp/uucheck | Command for checking directories, programs |
| /usr/spool/C.*sysnxxxx* | Work files |
| /usr/spool/X.*sysnxxxx* | Execute files |
| /usr/spool/D.*sysnxxxxyyy* | Data files |
| /usr/spool/LCK.*name* | Lock files |
| /usr/spool/TM.*pid.ddd* | Temporary data files |
| /etc/passwd | File for identifying your system to others |
| /usr/lib/uucp/Dialers | File for character strings for connections:<br>`dialer   substitutions   expect-send` |
| /usr/lib/uucp/Systems | File for identifying other systems to yours:<br>`Name   Time   Type   Class   Phone   Login` |
| /usr/lib/uucp/Dialcodes | File for storing phone abbreviations for L.sys |
| /usr/lib/uucp/uudemon.poll | Shell script for setting up work scheduling |
| /usr/lib/uucp/uudemon.hour | Shell script for scheduling work files |

| | |
|---|---|
| /usr/lib/uucp/uudemon.admin | Shell script for starting **uustat -p -q** |
| /usr/lib/uucp/uudemon.cleanup | Shell script for storing log files |
| /usr/lib/uucp/Permissions | File for controlling access to your system |
| /usr/lib/uucp/Sysfiles | File for identifying alternate files |
| /usr/lib/uucp/Devconfig | File for specifying STREAMS modules |

## Remote file sharing

# ***adv** [-r] [-d " description"] RESOURCE pathname [ client(s)]*

| | |
|---|---|
| /etc/rstab | File for advertizing automatically: |
| # **adv** | Display resources advertised by your host |
| # **nsquery** | Display resources advertised in your domain |
| # **unadv** | Unadvertise a resource |

**mount** Command for mounting a resource

# **mount [-r] -d** *RESOURCE mount'point*

| | |
|---|---|
| /etc/fstab | File for mounting automatically:<br>*RESOURCE mount'point* -d[r] |
| # **mount** | Display resources mounted on your host |
| # **rmntstat** | Display resources mounted in your domain |
| # **fuser [-ku]** *RESOURCE* | Report processes that use a resource |
| # **umount -d** *RESOURCE* | Unmount a remote resource |
| # **fumount [-w** *sec*] *RESOURCE* | Forcibly unmount a remote resource |

| | |
|---|---|
| /etc/rc1.d<br>/etc/rc2.d<br>/etc/rc3.d | Files for start and stop files for run level $\left\{\begin{matrix}1\\2\\3\end{matrix}\right\}$ |
| /etc/init.d | File for initialization shell scripts |
| /usr/nserve/domain | File with the name of the host's domain |
| /usr/nserve/netspec | File with the name of each host's network |
| /usr/nserve/rfmaster | File with names and addresses of name servers |
| /usr/nserve/loc.passwd | File with each host's authentication password |
| **# rfadmin -a** *domain.host* | Add a host |
| **# rfadmin -r** *domain.host* | Remove a host |
| **# rfstart [-v] [-p** *primary* | Start up a host |
| **# rfstop** | Shut down a host |
| **# rfpasswd** | Change a host's password |

| | |
|---|---|
| **sar -Dc** | Monitor incoming and outgoing requests for resources |
| **sar -Du** | Display the percent of CPU time spent on various activities |
| **sar -S** | Monitor server processes and remote requests |
| **fusage** | Find out how much other hosts are using your resources |
| **df [***resource***]** | Display remaining disk space on a remote resource |
| **NRDUSER** | Number of user entries for receive descriptor |
| **NRCVD** | Maximum number of receive descriptors |
| **NSNDD** | Maximum number of send descriptors |
| **NSRMOUNT** | Maximum number of entries in the mount table |
| **NADVERTISE** | Maximum number of entries in the advertise table |
| **MAXGDP** | Maximum number of virtual circuits |
| **MINSERVE** | Minimum number of server processes |
| **MAXSERVE** | Maximum number of server processes |

Files for security (/usr/nserve/auth.info directory):

| | |
|---|---|
| **domain/passwd** | Password files for individual domains |
| **domain/host/passwd** | Password files for individual hosts |
| **domain/host/group** | Group files for individual hosts |
| **uid.rules** | Rules for mapping remote users |
| **gid.rules** | Rules for mapping remote groups |

Global block

| | |
|---|---|
| **global** | These rules apply to all remote hosts |
| **default** | *local_id* A single local user id number |
| | **transparent** The same id number |
| **exclude** | |
| | *remote_id* A single remote id number |
| | *remote_id-remote_id* A range of remote id numbers |
| **map** | *remote_id:local* From *remote_id* to *local* |

Host block

| | |
|---|---|
| **host** | These rules apply only to the remote host(s) named: |
| **default** | *local* A single local user id name or number |
| | **transparent** The same id number |
| **exclude** | |
| | *remote_name* A single remote id name |
| | *remote_id* A single remote id number |
| | *remote_id-remote_id* A range of remote id numbers |
| **map** | *remote:local* From *remote* to *local* |
| | *remote* From *remote* to the same name or number |
| | **all** From each remote name to the same name |

## idload

Command for mapping users

```
# idload [-n] [-u u˙rules] [-g g˙rules] [ directory]
```

# Appendix L

# *termcap* and *terminfo*

This appendix gives you a summary of the most common features that can be specified in an entry in **termcap** or **terminfo**. For each feature, you will find its **termcap** name, its **terminfo** name, and a brief description. There are three types of variables used: numeric (like **lines#24**), string-valued (like **clear=^Z**), and boolean (like **am**).

## L.1 Terminal features

| termcap | terminfo | Description | Type |
|---|---|---|---|
| **li** | **lines** | Number of lines per screen | Numeric |
| **co** | **cols** | Number of columns per line | Numeric |
| **bs** | **bs** | Terminal can backspace | Boolean |
| **os** | **os** | Terminal can overstrike | Boolean |
| **am** | **am** | Terminal has automatic (right) margins | Boolean |
| **bw** | **bw** | Terminal has automatic left margins | Boolean |
| **hs** | **hs** | Terminal has an extra status line | Boolean |
| **xo** | **xon** | Terminal uses XON/XOFF handshaking | Boolean |
| **hc** | **hc** | Hardcopy terminal | Boolean |

## L.2 Cursor movement and scrolling

| termcap | terminfo | Description | Type |
|---|---|---|---|
| **ho** | **home** | Home the cursor | String |
| **cl** | **clear** | Clear screen and home the cursor | String |
| **cm** | **cup** | Cursor motion (cursor addressing) | String |
| **cv** | **vpa** | Vertical position absolute (set row) | String |
| **ch** | **hpa** | Horizontal position absolute (set column) | String |
| **up** | **cuu1** | Cursor up one line | String |
| **do** | **cud1** | Cursor down one line | String |
| **nd** | **cuf1** | Cursor right one column | String |
| **le** | **cub1** | Cursor left one column | String |
| **UP** | **cuu** | Cursor up *r* lines | String |
| **DO** | **cud** | Cursor down *r* lines | String |
| **RI** | **cuf** | Cursor right *c* columns | String |
| **LE** | **cub** | Cursor left *c* columns | String |
| **ch** | **hpa** | Set cursor column | String |
| **ta** | **ht** | Tab right to the next tab stop | String |
| **bt** | **cbt** | Tab left to the previous tab stop | String |
| **cr** | **cr** | Carriage return | String |
| **sf** | **ind** | Scroll text up one line | String |
| **sr** | **ri** | Scroll text down one line | String |
| **SF** | **indn** | Scroll text up *n* lines | String |
| **SR** | **rin** | Scroll text down *n* lines | String |

## L.3 Screen editing

| termcap | terminfo | Description | Type |
|---|---|---|---|
| **ce** | **el** | Clear to end of line | String |
| **cd** | **ed** | Clear to end of display | String |
| **cl** | **clear** | Clear entire screen and home cursor | String |
| **ic** | **ich1** | Insert one character | String |
| **dc** | **dch1** | Delete one character | String |
| **al** | **il1** | Insert one line | String |
| **dl** | **dl1** | Delete one line | String |
| **IC** | **ich** | Insert *n* characters | String |
| **DC** | **dch** | Delete *n* characters | String |
| **AL** | **il** | Insert *r* lines | String |
| **DL** | **dl** | Delete *r* lines | String |
| **ec** | **ech** | Erase *n* characters | String |
| **im** | **smir** | Enter insert mode | String |
| **ei** | **rmir** | Exit insert mode | String |
| **dm** | **smdc** | Enter delete mode | String |
| **ed** | **rmdc** | Exit delete mode | String |

## L.4 Functions activated by special keys

| termcap | terminfo | Description | Type |
|---|---|---|---|
| **kh** | **khome** | Home key | String |
| **ku** | **kcuu1** | Up arrow key | String |
| **kd** | **kcud1** | Down arrow key | String |
| **kr** | **kcuf1** | Right arrow key | String |
| **kl** | **kcub1** | Left arrow key | String |
| **kb** | **kbs** | Backspace key | String |
| **ke** | **kel** | Clear to end of line key | String |
| **kS** | **ked** | Clear to end of display key | String |
| **kC** | **kcl** | Clear screen key | String |
| **kl** | **kich1** | Insert one character key | String |
| **kD** | **kdch1** | Delete one character key | String |
| **kA** | **kil1** | Insert one line key | String |
| **dK** | **kdl1** | Delete one line key | String |
| **ks** | **smkx** | Enter keypad transmit mode | String |
| **ke** | **rmkx** | Exit keypad transmit mode | String |
| **k***n* | **kf***n* | Function key *n* | String |

## L.5 Video attributes

| termcap | terminfo | Description | Type |
|---|---|---|---|
| **md** | **bold** | Begin bold mode | String |
| **mh** | **dim** | Begin dim mode | String |
| **mb** | **blink** | Begin blink mode | String |
| **so** | **smso** | Begin standout mode | String |
| **us** | **smul** | Begin underscore mode | String |
| **mr** | **rev** | Begin reverse video mode | String |
| **mp** | **prot** | Begin protected mode | String |
| **se** | **rmso** | End standout mode | String |
| **ue** | **rmul** | End underscore mode | String |
| **me** | **sgr0** | End all attributes | String |

## L.6 Control directives

| termcap | terminfo | Description | Type |
|---|---|---|---|
| **is** | **is** | Initialization string (for setting options) | String |
| **if** | **if** | Initialization file (for setting tab stops) | String |
| **ti** | **ti** | Terminal initialization (cursor motion) | String |
| **te** | **te** | Terminal end (cursor motion) | String |
| **vs** | **vs** | Visual start (**vi**) | String |
| **ve** | **ve** | Visual end (**vi**) | String |
| **as** | **as** | Start alternate character set | String |
| **ae** | **ae** | End alternate character set | String |
| **po** | **mc5** | Turn on the printer | String |
| **pf** | **mc4** | Turn off the printer | String |
| **tc** | **tc** | Use another termcap or terminfo definition | String |

# Appendix M

# UNIX versus XENIX

## M.1 Description of XENIX

### IMPROVEMENTS

XENIX was developed by Microsoft in the late 1970s and early 1980s to fill the need for a commercial version of UNIX. (The Santa Cruz Operation became a major distributor of XENIX in 1983.) Evolving as it did in the world of academic research, UNIX originally lacked some of the features required for use in a business environment. XENIX and System V, Release 3.0 have added some of these features, along with some other general improvements.

First, let's consider the improvements. XENIX takes advantage of the unique features of each of the major microprocessors used in the more recent microcomputers (the Intel 8086, 8088, and 80386; the Motorola 68000; and the Zilog Z8000). For example, in the 8086 and 8088, XENIX avails itself of memory segmentation, dynamic relocatability of code, separation of data and instructions, input/output via memory-mapping, and multiprocessing.

On the minicomputers on which UNIX was developed, central processing and memory resources were scarce, while disk input/output was fast. On today's personal computers, however, the reverse is true, with central processing and memory abundant and disk input/output relatively slow. XENIX accommodates these differences by, for example, increasing the data transfer rate and reducing the amount of swapping of programs to and from disk.

Other improvements include avoidance of bad disk sectors, interactive system configuration, built-in floating-point arithmetic, logging of device errors, automatic file system recovery, optimization of the C compiler, and a number of miscellaneous bug fixes.

### ENHANCEMENTS

Next, we'll consider the enhancements, largely to facilitate the implementation of network and database applications. XENIX offers record- and file-locking. With this feature, a program can gain exclusive access to a single record, a group of records, or an entire file, thereby avoiding the errors that can result when more than one program attempts to access and update the same record at the same time.

To coordinate the activity of different tasks in progress, XENIX provides semaphores and a message buffer. Semaphores allow two different tasks to achieve synchronization with simple signals to each other. The message buffer allows unrelated tasks to communicate with each other.

Other enhancements include *scatter-loading* (placing segments of a program in noncontiguous areas of memory), synchronous writing of records to disk, and nonblocked reading of records to avoid unnecessary waiting.

## M.2 Differences between UNIX and XENIX

To the ordinary user, each version of XENIX is very similar to the corresponding version of UNIX from which it has been derived. For example, XENIX System V is very much like UNIX System V. This is because XENIX V is in fact UNIX System V—with the internal modifications described above and a number of extra utilities. You will see differences mainly in the area of system administration, including additional utilities like the **mkuser** and **rmuser** commands of XENIX, which aren't included in AT&T's System V.

Other features of XENIX worth mentioning are its menu-driven visual shell (**vsh**); its custom installation program (**custom**); its automatic tuning for the current system configuration; its Micnet subsystem for direct networking (**netutil**, **remote**, and **rcp**); and its support of programmable key functions (**setkey**), multiple and color screens (**multiscreen** and **setcolor**), various diskette formats, add-on boards for serial ports, laser printers, and cross-development between MS-DOS and XENIX.

### Commands unique to UNIX

The following System V commands are available in UNIX, but not in XENIX. The right-hand column indicates the part of this book to which the command most closely relates. A dash (—) means the command relates to software development, which is not covered in this book.

| Command | Description | Topic |
|---|---|---|
| **300/300s** | DASI 300/300s terminal handler | VI |
| **450** | DASI 450 terminal handler | VI |
| **arcv** | Convert archive file format | VI |
| **convert** | Format archive and object files | — |
| **crypt** | Encrypt and decrypt standard input to standard output | I |
| **ct** | Call a terminal and log in | VII |
| **dis** | Disassembler | — |
| **efl** | Extended FORTRAN language | — |

| | | |
|---|---|---|
| **fsplit** | Split **f77**, **efl**, or **ratfor** files into segments | — |
| **gath** | Gather files for Remote Job Entry (RJE) | VII |
| **greek** | Set up filtering for an extended character set | IV |
| **help** | Explain a message or a command | I |
| **hp** | HP 2640 and HP 2621 terminal handler | VI |
| **hpio** | Tape file archiver for the HP 2645A terminal | VI |
| **login** | Log into a UNIX system | I |
| **makekey** | Create an encryption key | I |
| **manprog** | Return a date file modified for **nroff** or **troff** | IV |
| **mvt** | Typeset a viewgraph | IV |
| **nscstat** | Report the status of an NSC network | VII |
| **nsctorje** | Reroute jobs from the NSC network to RJE | VII |
| **nusend** | Send files to UNIX over an NSC network | VII |
| **osdd** | Print **mm** documents in OSDD format | IV |
| **pcc** | Portable C compiler | — |
| **pdp11** | Return exit status of true for a PDP-11 | VI |
| **pg** | Display a file by pages | I |
| **prof** | Display profile data | VI |
| **rjestat** | Report RJE status and simulate an IBM remote console | VII |
| **sadp** | Report on disk access | VI |
| **sar** | Report on system activity | VI |
| **scat** | Concatenate and route files to a synchronous printer | I |
| **scc** | C compiler to generate stand-alone programs | — |
| **sdb** | Symbolic debugger for C and **f77** | — |
| **se** | Screen editor | II |
| **send** | Send files to RJE | VII |
| **sno** | SNOBOL interpreter | — |
| **stlogin** | Log into a synchronous terminal | I |
| **ststat** | Display the status of synchronous terminals | VI |
| **tabs** | Set terminal tabs | VI |
| **tc** | Simulate a phototypesetter | IV |
| **timex** | Display time and system activity for a command | VI |
| **trouble** | Report trouble | VI |
| **u3b** | Return an exit status of true for a 3B2 processor | VI |
| **vax** | Return an exit status of true for a VAX 11/750 or 11/780 | VI |
| **vc** | Handle UNIX version control | VI |
| **vedit** | Screen editor for beginners | II |
| **vpr** | Route files to a Versatec printer | I |

## Commands unique to XENIX

The following System V commands are available in XENIX, but not in UNIX. The right-hand column indicates the part of this book to which the command most closely relates. A dash (—) means the command relates to software development, which is not covered in this book.

| Command | Description | Topic |
|---|---|---|
| **assign** | Assign a device to a user | VI |

| | | |
|---|---|---|
| **asx** | C compiler assembler for **cmerge** | — |
| **batch** | Run a command when system scheduling permits | VI |
| **cmchk** | Show the block size of a hard disk in bytes | VI |
| **cref** | Create a cross reference listing | — |
| **crontab** | Copy a file to the **crontab** directory | VI |
| **csh** | C shell | V |
| **ctags** | Create a tags file of functions for a source program | — |
| **custom** | Select XENIX system features | VI |
| **cwcheck** | Check constant-width text | III |
| **deassign** | Deassign a device from a user | VI |
| **devnm** | Display device names for a mounted file system | VI |
| **dial** | Dial a modem | I |
| **diction** | Locate awkward phrases in a file | III |
| **diskcmp** | Compare the contents of two diskettes | VI |
| **diskcp** | Copy the contents of one diskette to another | VI |
| **dmesg** | Display all system messages since the last boot | VI |
| **doscat** | Copy DOS files to the standard output | VI |
| **doscp** | Copy files from a DOS disk to a XENIX file system | VI |
| **dosdir** | List DOS files using the format of the DOS **dir** command | VI |
| **dosld** | Cross linker from XENIX to MS-DOS | — |
| **dosls** | List DOS files using the format of the XENIX **ls** command | VI |
| **dosmkdir** | Create a directory on a DOS disk | VI |
| **dosrm** | Remove a DOS file | VI |
| **dosrmdir** | Remove a directory from a DOS disk | VI |
| **dtype** | Report on disk type | VI |
| **eqncheck** | Check an **eqn** input file for syntax | IV |
| **explain** | Suggest better phrases for **diction** | III |
| **finger** | Find information about system users | VI |
| **fixhdr** | Change headers of binary files | — |
| **fixperm** | Set or correct file permissions | VI |
| **format** | Format a diskette | VI |
| **gets** | Get a string from the standard input | V |
| **grpcheck** | Check entries in a group file | VI |
| **hd** | Produce a hexadecimal file dump | — |
| **hdr** | Display parts of object files | — |
| **head** | Display the opening lines of a file | I |
| **l** | Display the contents of a directory in long format (**ls -l**) | I |
| **lc** | Display the contents of a directory in columns | I |
| **look** | Locate lines of text that begin with a string in a list | III |
| **lpr** | Send a file to a line printer | I |
| **mapscrn** | Display the output mapping of a screen | — |
| **mapstr** | Display the mapping of function keys on a keyboard | — |
| **masm** | XENIX assembler for the 8086 and the 80286 | — |
| **mkdev** | Create device files | VI |
| **mknod** | Create special files | VI |
| **mkstr** | Create an error message file from a set of C programs | — |
| **mkuser** | Set up a new user account on the XENIX system | VI |
| **more** | Display the contents of a file by screens | I |
| **multiscreen** | Access up to ten screens on the console | VI |

| | | |
|---|---|---|
| **mvdir** | Move a directory and its contents | VI |
| **netutil** | Create and maintain a Micnet network | VII |
| **prep** | Break a file into one-word lines | I |
| **pstat** | Display system statistics | VI |
| **pwcheck** | Check the password file for inconsistencies | VI |
| **random** | Generate a random number | I |
| **ranlib** | Create a random library from archive | VI |
| **rcp** | Copy files from one system to another over Micnet | VI |
| **red** | Restricted version of the **ed** text editor | II |
| **remote** | Execute a command on a machine in XENIX network | VII |
| **restor** | Restore files from archive | VI |
| **restore** | Restore files from archive | VI |
| **rmuser** | Remove a user from a XENIS system | VI |
| **sddate** | Check the date of the last system backup | VI |
| **setcolor** | Select the screen color for a color monitor | VI |
| **setkey** | Assign function keys | VI |
| **settime** | Change the dates of last file access and modification | VI |
| **soelimn** | Perform **.so** requests during **nroff** input | IV |
| **stackuse** | Determine stack requirements for a C program | — |
| **strings** | Search binary files for ASCII strings | — |
| **style** | Report on writing style | IV |
| **tset** | Set terminal-dependent parameters | VI |
| **uuclean** | Delete old **uucp** files | VII |
| **uusub** | Define a **uucp** subnetwork and monitor traffic | VII |
| **vsh** | Menu-driven visual shell | V |
| **whodo** | Determine what users are doing | VI |
| **xref** | Create a cross reference for C programs | — |
| **xstr** | Facilitate shared strings by extracting strings | — |
| **yes** | Output repeated strings | III |

## M.3 Features of System V, Release 3

UNIX System V, Release 3 offers features that provide resource-sharing, standardization in communication, shared libraries, demand paging, and file- and record-locking. The document that describes System V software interface standards in detail for this version and future versions of UNIX is AT&T's two-volume *System V Interface Definition.*[1] Here is a summary of the features of Release 3:

1. *Remote File Sharing (RFS)* allows users to share resources (files and devices) over a network. This feature is transparent to the user and relies on the use of standard UNIX commands. It allows system administrators to save disk space and reduce the number of expensive devices required (tape drives, plotters, laser printers, and so on).

[1] *System V Interface Definition*, [no location given], AT&T, 1986.

2. *STREAMS* provides a consistent, uniform character I/O interface based on the ISO/OSI reference model, allowing programmers to develop network interfaces that are independent of media and low-level protocols.

3. *Transport Level Interface and Transport Provider Interface (TLI/TPI)* combine with STREAMS to implement the reference model of the International Standards Organization (ISO).

   The TLI is accessible from either the kernel or a user process. Kernel applications and higher level protocols access the TLI via STREAMS messages; user processes access the TLI via a new Transport Interface Library. The RFS feature operates over TLI directly through STREAMS; **cu** and **uucp** now operate over the TLI using the new library. The capabilities that must be furnished by a protocol (transport provider) are specified by the TPI. The TPI also specifies how to maintain consistency with the new library.

4. *Modification of* **cu** *and* **uucp** to incorporate STREAMS and TLI/TPI make these programs independent of media and low-level protocols.

5. *Shared libraries* for active processes reduce disk and memory space requirements and simplify system administration.

6. *Demand-paging* (actually from Release 2.1) makes it easier to run programs that are too large to fit in memory and allows all programs to run more efficiently.

7. *Mandatory and advisory file- and record-locking* allow many users to share and update files freely.

8. *ASSIST* allows beginners to construct command lines from a series of questions on a screen menu.

9. *Internationalization* paves the way for the introduction of supplementary character sets in future release of UNIX. This will allow users in different parts of the world to add national supplements (with messages, databases, documentation, and device drivers for other languages and hardware). Eight-bit words will allow all European character sets; 16-bit words will allow Oriental character sets (Chinese, Japanese, Korean, and so on).

# Appendix N

# Character codes

TABLE N.1. U.S. ASCII Character Set

| Char | dec | oct | hex | char | dec | oct | hex | char | dec | oct | hex | char | dec | oct | hex |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NUL | 0 | 000 | 0 | [sp] | 32 | 040 | 20 | @ | 64 | 100 | 40 | ‘ | 96 | 140 | 60 |
| SOH | 1 | 001 | 1 | ! | 33 | 041 | 21 | A | 65 | 101 | 41 | a | 97 | 141 | 61 |
| STX | 2 | 002 | 2 | ” | 34 | 042 | 22 | B | 66 | 102 | 42 | b | 98 | 142 | 62 |
| ETX | 3 | 003 | 3 | # | 35 | 043 | 23 | C | 67 | 103 | 43 | c | 99 | 143 | 63 |
| EOT | 4 | 004 | 4 | $ | 36 | 044 | 24 | D | 68 | 104 | 44 | d | 100 | 144 | 64 |
| ENQ | 5 | 005 | 5 | % | 37 | 045 | 25 | E | 69 | 105 | 45 | e | 101 | 145 | 65 |
| ACK | 6 | 006 | 6 | & | 38 | 046 | 26 | F | 70 | 106 | 46 | f | 102 | 146 | 66 |
| BEL | 7 | 007 | 7 | ’ | 39 | 047 | 27 | G | 71 | 107 | 47 | g | 103 | 147 | 67 |
| BS | 8 | 010 | 8 | ( | 40 | 050 | 28 | H | 72 | 110 | 48 | h | 104 | 150 | 68 |
| HT | 9 | 011 | 9 | ) | 41 | 051 | 29 | I | 73 | 111 | 49 | i | 105 | 151 | 69 |
| NL | 10 | 012 | A | * | 42 | 052 | 2A | J | 74 | 112 | 4A | j | 106 | 152 | 6A |
| VT | 11 | 013 | B | + | 43 | 053 | 2B | K | 75 | 113 | 4B | k | 107 | 153 | 6B |
| FF | 12 | 014 | C | , | 44 | 054 | 2C | L | 76 | 114 | 4C | l | 108 | 154 | 6C |
| CR | 13 | 015 | D | - | 45 | 055 | 2D | M | 77 | 115 | 4D | m | 109 | 155 | 6D |
| SO | 14 | 016 | E | . | 46 | 056 | 2E | N | 78 | 116 | 4E | n | 110 | 156 | 6E |
| SI | 15 | 017 | F | / | 47 | 057 | 2F | O | 79 | 117 | 4F | o | 111 | 157 | 6F |
| Dle | 16 | 020 | 10 | 0 | 48 | 060 | 30 | P | 80 | 120 | 50 | p | 112 | 160 | 70 |
| DC1 | 17 | 021 | 11 | 1 | 49 | 061 | 31 | Q | 81 | 121 | 51 | q | 113 | 161 | 71 |
| DC2 | 18 | 022 | 12 | 2 | 50 | 062 | 32 | R | 82 | 122 | 52 | r | 114 | 162 | 72 |
| DC3 | 19 | 023 | 13 | 3 | 51 | 063 | 33 | S | 83 | 123 | 53 | s | 115 | 163 | 73 |
| DC4 | 20 | 024 | 14 | 4 | 52 | 064 | 34 | T | 84 | 124 | 54 | t | 116 | 164 | 74 |
| NAK | 21 | 025 | 15 | 5 | 53 | 065 | 35 | U | 85 | 125 | 55 | u | 117 | 165 | 75 |
| SYN | 22 | 026 | 16 | 6 | 54 | 066 | 36 | V | 86 | 126 | 56 | v | 118 | 166 | 76 |
| ETB | 23 | 027 | 17 | 7 | 55 | 067 | 37 | W | 87 | 127 | 57 | w | 119 | 167 | 77 |
| CAN | 24 | 030 | 18 | 8 | 56 | 070 | 38 | X | 88 | 130 | 58 | x | 120 | 170 | 78 |
| EM | 25 | 031 | 19 | 9 | 57 | 071 | 39 | Y | 89 | 131 | 59 | y | 121 | 171 | 79 |
| SUB | 26 | 032 | 1A | : | 58 | 072 | 3A | Z | 90 | 132 | 5A | z | 122 | 172 | 7A |
| ESC | 27 | 033 | 1B | ; | 59 | 073 | 3B | [ | 91 | 133 | 5B | { | 123 | 173 | 7B |
| FS | 28 | 034 | 1C | < | 60 | 074 | 3C | & | 92 | 134 | 5C | — | 124 | 174 | 7C |
| GS | 29 | 035 | 1D | = | 61 | 075 | 3D | ] | 93 | 135 | 5D | } | 125 | 175 | 7D |
| RS | 30 | 036 | 1E | > | 62 | 076 | 3E | ˆ | 94 | 136 | 5E |  | 126 | 176 | 7E |
| US | 31 | 037 | 1F | ? | 63 | 077 | 3F | _ | 95 | 137 | 5F | DEL | 127 | 177 | 7F |

# N.1 The control characters

The control characters shown in Table N.1 are described in Table N.2. Each entry gives one character (abbreviation and full name), the keys used to generate it, and some typical uses for the character in the operation of a terminal. The names were selected long before computer users adopted them.

TABLE N.2. The Control Characters

| | Character | keys | typical use |
|---|---|---|---|
| NUL | Null | CTRL-@ (^@) | Occupy space in a string |
| SOH | Start of Header | CTRL-A (^A) | Begin a heading |
| STX | Start of Text | CTRL-B (^B) | Begin text |
| ETX | End of Text | CTRL-C (^C) | End text |
| EOT | End of Transmission | CTRL-D (^D) | End transmission (or UNIX process) |
| ENQ | Enquiry | CTRL-E (^E) | Request information |
| ACK | Acknowledgment | CTRL-F (^F) | Affirmative reply |
| BEL | Bell | CTRL-G (^G) | Sound the beeper (bell) |
| BS | Backspace | CTRL-H (^H) | Move the cursor (or printhead) back |
| HT | Horizontal Tab | CTRL-I (^I) | Advance to a tab stop (TAB key) |
| NL | Newline* | CTRL-J (^J) | Begin a new line of text (RETURN key) |
| VT | Vertical Tab | CTRL-K (^K) | Advance to a designated line |
| FF | Formfeed | CTRL-L (^L) | Advance to top of page |
| CR | Carriage Return | CTRL-M (^M) | Return to start of line |
| SO | Shift Out | CTRL-N (^N) | Shift out of ASCII codes |
| SI | Shift In | CTRL-O (^O) | Shift back into ASCII codes |
| DLE | Data Link Escape | CTRL-P (^P) | Begin transmission control sequence |
| DC1 | Device Control 1 | CTRL-Q (^Q) | Transmit-ON (XON); resume scrolling |
| DC2 | Device Control 2 | CTRL-R (^R) | Turn on an unspecified device |
| DC3 | Device Control 3 | CTRL-S (^S) | Transmit-OFF (XOFF); stop scrolling |
| DC4 | Device Control 4 | CTRL-T (^T) | Turn of an unspecified device |
| NAK | Negative Ack. | CTRL-U (^U) | Negative reply |
| SYN | Synchronous Idle | CTRL-V (^V) | Wait for further transmission |
| ETB | End of Trans. Block | CTRL-W (^W) | End a block of transmission data |
| CAN | Cancel | CTRL-X (^X) | Ignore data just transmitted |
| EM | End of Medium | CTRL-Y (^Y) | End of tape, paper tape, etc. |
| SUB | Substitute | CTRL-Z (^Z) | Replace one character with another |
| ESC | Escape | CTRL-[ (^[) | Begin an escape sequence |
| FS | File Separator | CTRL-\ (^\) | Mark the end of a file |
| GS | Group Separator | CTRL-] (^]) | Mark the end of a group (block, etc.) |
| RS | Record Separator | CTRL-^ (^^) | Mark the end of a record |
| US | Unit Separator | CTRL-_ (^_) | Mark the end of a unit (field, etc.) |

* This character is known as LF (linefeed) in most ASCII tables; but in UNIX ASCII tables, it's called NL (newline), which combines the function of CR and LF.

Note that UNIX makes no distinction between CTRL-D and CTRL-d, which means that you don't have to press the SHIFT key when you type CTRL-D.

The 8-bit extended character set, with its international symbols, is shown in Table N.3. You can currently access these characters via the XENIX **mapchan** command; by the time this book is published, it should also be possible to access these characters in UNIX.

TABLE N.3. Extended Character Set

| Char | dec | oct | hex | char | dec | oct | hex | char | dec | oct | hex | char | dec | oct | hex |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| [#] | 128 | 200 | 80 | NBSP | 160 | 240 | A0 | A | 192 | 300 | C0 | a | 224 | 340 | E0 |
| [#] | 129 | 201 | 81 | | 161 | 241 | A1 | A | 193 | 301 | C1 | a | 225 | 341 | E1 |
| [#] | 130 | 202 | 82 | | 162 | 242 | A2 | A | 194 | 302 | C2 | a | 226 | 342 | E2 |
| [#] | 131 | 203 | 83 | | 163 | 243 | A3 | A | 195 | 303 | C3 | a | 227 | 343 | E3 |
| IND | 132 | 204 | 84 | | 164 | 244 | A4 | A | 196 | 304 | C4 | a | 228 | 344 | E4 |
| NEL | 133 | 205 | 85 | | 165 | 245 | A5 | A | 197 | 305 | C5 | a | 229 | 345 | E5 |
| SSA | 134 | 206 | 86 | | 166 | 246 | A6 | | 198 | 306 | C6 | | 230 | 346 | E6 |
| ESA | 135 | 207 | 87 | | 167 | 247 | A7 | C | 199 | 307 | C7 | c | 231 | 347 | E7 |
| HTS | 136 | 210 | 88 | | 168 | 250 | A8 | E | 200 | 310 | C8 | e | 232 | 350 | E8 |
| HTJ | 137 | 211 | 89 | | 169 | 251 | A9 | E | 201 | 311 | C9 | e | 233 | 351 | E9 |
| VTS | 138 | 212 | 8A | | 170 | 252 | AA | E | 202 | 312 | CA | e | 234 | 352 | EA |
| PLD | 139 | 213 | 8B | | 171 | 253 | AB | E | 203 | 313 | CB | e | 235 | 353 | EB |
| PLU | 140 | 214 | 8C | | 172 | 254 | AC | I | 204 | 314 | CC | i | 236 | 354 | EC |
| RI | 141 | 215 | 8D | | 173 | 255 | AD | I | 205 | 315 | CD | i | 237 | 355 | ED |
| SS2 | 142 | 216 | 8E | | 174 | 256 | AE | I | 206 | 316 | CE | i | 238 | 356 | EE |
| SS3 | 143 | 217 | 8F | | 175 | 257 | AF | I | 207 | 317 | CF | i | 239 | 357 | EF |
| Dcs | 144 | 220 | 90 | | 176 | 260 | B0 | D | 208 | 320 | D0 | d | 240 | 360 | F0 |
| PU1 | 145 | 221 | 91 | | 177 | 261 | B1 | N | 209 | 321 | D1 | n | 241 | 361 | F1 |
| PU2 | 146 | 222 | 92 | | 178 | 262 | B2 | O | 210 | 322 | D2 | o | 242 | 362 | F2 |
| STS | 147 | 223 | 93 | | 179 | 263 | B3 | O | 211 | 323 | D3 | o | 243 | 363 | F3 |
| CCH | 148 | 224 | 94 | | 180 | 264 | B4 | O | 212 | 324 | D4 | o | 244 | 364 | F4 |
| MW | 149 | 225 | 95 | | 181 | 265 | B5 | O | 213 | 325 | D5 | o | 245 | 365 | F5 |
| SPA | 150 | 226 | 96 | | 182 | 266 | B6 | O | 214 | 326 | D6 | o | 246 | 366 | F6 |
| EPA | 151 | 227 | 97 | | 183 | 267 | B7 | [#] | 215 | 327 | D7 | [#] | 247 | 367 | F7 |
| [#] | 152 | 230 | 98 | | 184 | 270 | B8 | O | 216 | 330 | D8 | o | 248 | 370 | F8 |
| [#] | 153 | 231 | 99 | | 185 | 271 | B9 | U | 217 | 331 | D9 | u | 249 | 371 | F9 |
| [#] | 154 | 232 | 9A | | 186 | 272 | BA | U | 218 | 332 | DA | u | 250 | 372 | FA |
| CSI | 155 | 233 | 9B | | 187 | 273 | BB | U | 219 | 333 | DB | u | 251 | 373 | FB |
| ST | 156 | 234 | 9C | | 188 | 274 | BC | U | 220 | 334 | DC | u | 252 | 374 | FC |
| OSC | 157 | 235 | 9D | | 189 | 275 | BD | Y | 221 | 335 | DD | y | 253 | 375 | FD |
| PM | 158 | 236 | 9E | | 190 | 276 | BE | | 222 | 336 | DE | | 254 | 376 | FE |
| APC | 159 | 237 | 9F | | 191 | 277 | BF | | 223 | 337 | DF | y | 255 | 377 | FF |

## N.2 The extended control characters

The control characters shown in Table N.3 are described in Table N.4. Each entry gives one character (abbreviation and full name), the keys used to generate it, and some typical uses for the character in the operation of a terminal.

TABLE N.4. The Extended Control Characters

| | Character | keys | typical use |
|---|---|---|---|
| IND | Index | ESC D | Move the cursor down one line |
| NEL | Next line | ESC E | Move the cursor to the first column of the nex |
| SSA | Start of selected area | ESC F | Start a designated block |
| ESA | End of selected area | ESC G | End a designated block |
| HTS | Horizontal Tab Set | ESC H | Set horizontal tab at cursor column |
| HTJ | Horizontal Tab with Justification | ESC I | Set horizontal tab with justification |
| VTS | Vertical Tab Set | ESC J | Set vertical tab at cursor line |
| PLD | Partial Line Down | ESC K | Move down less than a full line |
| PLU | Partial Line Up | ESC L | Move up less than a full line |
| RI | Reverse Index | ESC M | Move the cursor up one line |
| SS2 | Single Shift 2 | ESC N | Shift to a second character set |
| SS3 | Single Shift 3 | ESC O | Shift to a third character set |
| DCS | Device Control String | ESC P | Introduce a device control string |
| PU1 | Private Use 1 | ESC Q | First user-designated |
| PU2 | Private Use 2 | ESC R | Second user-designated |
| STS | Set Transmit State | ESC S | Prepare for transmission of text |
| CCH | Cancel Character | ESC T | Cancel one character |
| MW | Message Waiting | ESC U | Notification of a message |
| SPA | Start of Protected Area | ESC V | Start a protected block of text |
| EPA | End of Protected Area | ESC W | End a protected block of text |
| CSI | Control Sequence Introducer | ESC [ | Introduce a control sequence |
| ST | String Terminator | ESC \ | End a string introduced by DCS |
| OSC | Operating System Command | ESC ] | Command for an operating system |
| PM | Privacy Message | ESC ^ | Non-public message to follow |
| APC | Application Program Command | ESC _ | Command for an application program |

# Glossary

**advertise** In the Remote File Sharing (FRS) system, to make known to other UNIX systems in a network the availability of a resource on your system with the **adv** (advertise) command. See Chapter 40, "Basic Resource Sharing," for details.

**alias** In the C shell, a user-defined command constructed from standard UNIX commands.

**analog** A method of indicating quantities by measuring something (voltage, current, a column of mercury, the position of a pointer) on a continuous scale. Until recently, audio and video devices (telephones, radios, stereos, and televisions) have been analog; while computers have been *digital*. However, as the twentieth century draws to a close, designers of all electronic devices are abandoning analog logic in favor of digital. (See also *digital*.)

**argument** In computer usage, something added to a basic command to modify the way the command is used. In the command line **cat wall**, **wall** is an argument for the **cat** command. The UNIX shell uses **$0** to denote a command and **$1**, **$2**, **$3**, ... to denote its arguments.

**array** In computer usage, a table of either string or numeric values, each referred to by an ordinal number called its *index*. By convention, each *element* of an array is denoted by giving the name of the array, followed by the index in brackets. For example, if the individual letters of the word `UNIX` have been stored in an array called **name**, then **name[1]** refers to `U`, **name[2]** refers to `N`, **name[3]** refers to `I`, and **name[4]** refers to `X`. In mathematical usage, an *array* is the same as a *matrix*.

**ASCII** In computer usage, "American Standard Code for Information Interchange," a set of 128 codes that are used by UNIX and many other hardware and software systems for control and display. The ASCII character set is known in Europe as the `ISO` (International Standards Organization) character set. (See Chapter 5, "Communication in UNIX," for details.)

**assembly language** In computer usage, a computer language whose commands parallel the primitive machine language codes used by the main processor, or CPU (central processing unit). The main advantage of programming in assembly language is that the programs produced run very fast; the main disadvantages are that the coding is very difficult to read and that the programs produced are not *portable* to machines that use a different CPU. Machine language and assembly language are also referred to as *low-level* languages.

**asynchronous** In electronic communication, a method of communication that allows individual characters to be transmitted one at a time; as opposed to *synchronous* communication, which allows two devices to exchange large amounts of data by synchronizing the timing of the exchange. In general, *asynchronous* co mmunication is used for devices like keyboards, where human entry of data is relatively slow and at random intervals; while *synchronous* communication is used for high-speed devices that communicate directly with each other.

**background** In computer usage, processing that a system carries out without human interaction; as opposed to *foreground* processing. In UNIX, the ampersand (&), appended to the end of a command line, indicates a request for background processing.

**backup** In computer usage, an extra copy of a file that can be retrieved in the event that some system failure damages or destroys your working copy.

**baud rate** See *data rate*.

**binary** Base two. (See *number*.)

**bit**

**blank** Open space that appears either on the video screen or on a printed page, produced by either space characters or tab characters. The limitations of some terminals and printers sometimes make it necessary to convert tabs to spaces, or vice versa.

**block** In computer usage, a predetermined segment of data, usually measured in *bytes*, that is used to set aside larger amounts of storage space. In System V, a block is 1,024 bytes; in earlier versions of UNIX, a block was 512 bytes. Mass storage devices like disk and tape drives, which transfer data in blocks, are sometimes referred to as *block devices* in UNIX terminology.

**bold** In printing, a style of typeface that is characterized by thicker, darker character strokes.

**Bourne shell** The standard shell, originally designed by Stephen R. Bourne at Bell Telephone Laboratories; as opposed to the *C shell* and the

*Korn shell*, newer shells derived from the standard shell. The Bourne shell is the fastest of the three, the C shell is richer in programming features, and the Korn shell combines the advantages of the two earlier shells. (See also *shell.*)

**buffer** In computer usage, a temporary storage area in *memory* where data may be either held or processed during an intermediate phase of processing.

**bug** In computer usage, an error in *hardware* or *software* design. According to legend, the term came into the vernacular when an investigation into an early computer failure in the 1940s revealed a dead insect lying amid the machine's electronic components. Getting rid of bugs is called *debugging.*

**byte** In computer usage, a smaller unit of measure, usually eight *bits*—In text-processing, a byte is equivalent to one character.

**character** The smallest unit that is used in text-processing, which may either represent a visible symbol (*alphanumeric* or *special character*) or carry out a control function (*control character*). You can enter a control character at the keyboard either by pressing a designated key (RETURN, TAB, ESC) or by holding down the CTRL key and pressing another key. The *asynchronous* devices that process characters (terminals, printers, modems) are referred to as *character devices* in UNIX terminology.

**client** In networking in general and in the Remote File Sharing (RFS) system in particular, a UNIX system that uses the file systems made available by another UNIX system. This feature is unique to System V, Release 3.0.

**compiler** A program that converts the coding used to enter a high-level language to machine language. Any program written in C, FORTRAN, Pascal, Ada, or Modula-2 must be compiled before the computer can execute it. The C compiler in UNIX is called **cc** (**cc** also performs assembly).

**concatenate** To join two units, such as words or files. The UNIX command that concatenates files is called **cat**.

**condition code** See *exit status.*

**console** In computer usage, the primary terminal in a multi-terminal system, usually located next to, or on top of, the main computer. In UNIX, the console is named **tty00** (terminal 00); in System V, any terminal can be used as the console because of the *virtual console* feature.

**control character** See *character*.

**C shell** The shell that was developed by William N. Joy and others at the University of California in the late 1970s to enhance the original shell. The C shell allows *arrays* and reselection of previous command lines (referred to as *events*). (See also *shell.*)

**cursor** The symbol that holds your place on a video screen and points to where the character you type next will be entered.

**daemon** In DEC and UNIX usage, a process that the system initiates automatically without human intervention.

**data rate** In communication, the rate at which data is being transferred, as measured in bits per second (bit/s). Also called the *bit rate*; also called (incorrectly) the *baud rate*. Engineers love to say "baud" and "baud rate" because they sound so esoteric; however, *baud rate* is controlled by the telephone company, not by a computer that happens to be using the telephone lines.

**debug** To get rid of bugs. (See also *bug*.)

**decimal** Base ten. (See *number*.)

**demand paging** A memory multiplexing scheme whereby programs are executed in segments called pages. Whenever an instruction in a program refers to data or an instruction not currently in memory, the page that contains that data or instruction can be swapped into memory upon demand.

**device** Any piece of equipment that is attached to a computer, also called an *input/output device* or a *peripheral device*. Terminals, printer, modems, disk drives, and tape drives are examples of physical devices. In UNIX, a file system is regarded as a logical device.

**device driver** A program that processes the flow of data between a computer and one of its devices.

**device file** A file that indicates the presence of a device that is included in a UNIX system. Device files are found in the device directory /dev. Device files are also called *special files*.

**digital** A method of measuring that reduces quantities to numeric values, expressed in digits. Except for a handful of *analog* computers, all computers are digital, and all of them use the binary system to express numbers. There are several advantages of *digital* processing over *analog*: 1) it can be performed with simpler electronic circuits; 2) the opportunity for error is reduced; 3) all computers use exactly the same number system. (See also *analog*.)

**directory** In UNIX and other operating systems, a file that contains the names of other files. In UNIX, the primary directory is called the **root** directory (or just the *root*), and is denoted by a single slash (/).

**domain** In networking in general and in the Remote File Sharing (RFS) system, a set of UNIX systems on a network that are grouped together for administrative purposes.

**drive** A machine that rotates a storage medium (disk or tape), reads information from it, and writes information to it.

**driver** See *device driver*.

**echo** To display on a video screen a character that has been typed at the keyboard. The shell's **echo** command performs this function. Because printing Teletype machines, with paper output in place of a screen, were used for the earliest UNIX terminals, displaying of characters on the screen (or echoing) is frequently referred to as *printing* in UNIX manuals.

**element** An individual item in an *array*. (See also *array*.)

**environment** A set of variables and their assigned values that is made available to a process when the process is called; also called the *calling environment*. The environment in effect when you log in is stored in a file named either **.profile** (Bourne shell) or **.login** (C shell). You can alter the environment during a UNIX session by re-assigning variables; refer to either Chapter 23, "Bourne Shell Variables" or Chapter 26, "C Shell Variables").

**EOF** In computer usage, end-of-file character; e.g., in UNIX, CTRL-D.

**escape** To leave one program and enter another (as in escaping to **ex** from **vi**); to take a special character (such as ^ or $) literally, rather than use its special meaning to the shell. In general, you can use a backslash (\) o escape a special character. However, the shell also allows you to use pairs of quotation marks ("... ", '... ', `... `) to produce different results (see Chapter 23, "Bourne Shell Variables," for details). In UNIX terminology, *escaping* is also called *quoting.*

**event** In the C shell, a previous command line that you have executed, which is stored in a *history list* for later retrieval, modification, and re-execution.

**exit-status** In UNIX and other operating systems, a numeric code that is returned to the operating system at the conclusion of a process to indicate whether or not the process was executed successfully. In UNIX, zero (0) means that the process failed, while a non-zero code means that it succeeded.

**expression** A numeric or string representation of something that may assume one of a number of different values. For example, `2*(x+3)` represents "8" if $x = 1$, "10" if $x = 2$, and so on, while `a.e` represents "ace" if `c` is read for `.`, "axe" if `x` is read for `.`, and so on. See also *regular expression.*

**field** In text-processing, an area within a line that may be named and processed by a program. The **sort** and **awk** commands both use fields extensively, but, unfortunately, use different conventions for naming fields.

**field separator** A character that separates fields, usually a blank (a space or a tab). The **awk** program and the Bourne shell allow you to use any character instead.

**fifo file** First-in first-out file; also called a *named pipe.*

**file** A collection of characters with a name that may contain a program, text, records, the names of other files, or the names of devices. In UNIX, a file that contains the names of other files is called a *directory file*, or a *directory*; a file that contains the names of devices is called a *device file* or a *special file* and a file that begins with a period (or dot) is called an *invisible file* (because the ordinary **ls** command doesn't display it).

**file server** In networking in general and in the Remote File Sharing (RFS) system in particular, a node that offers files to other nodes. (See also *server*)

**fill** In text-formatting, to draw upon text in subsequent lines to make the current line extend to its margins (or nearly to its margins).

**font** In typesetting, a set of characters that share a common size, style, and typeface.

**footer** In text-formatting, a segment of text that is placed at the bottom, or foot, of every page of a multi-page document.

**foreground** In computer usage, processing that invloves user interaction; as opposed to *background* processing.

**fork** In UNIX shell usage, to produce a copy of a program before executing it; *forking* is also called *spawning.*

**gateway server** Also called a *gateway.* (See *server.*)

**global** In computer usage, something that occurs or is defined throughout an entire file or program; as opposed to *local.*

**hardware** The tangible parts of a computer system, the actual electronic circuits and connections; as opposed to the programs executed on the system (*software*).

**header** In text-formatting, a segment of text that is placed at the top, or head, of every page of a multi-page document.

**hexadecimal** Base sixteen. (See number.)

**history** In the C shell, a collection of previous command lines (or *events*) that can be displayed on the screen for review.

**home directory** The directory assigned to you by the system administrator; the directory that you enter whenever you log in; also called your *login directory.*

**host** In networking in general and in the Remote File Sharing (FRS) system in particular, a single computer in a network; also called a *node* or a *station.*

**i-node** In UNIX system administration, a data structure that provides information about one file (length; location; number of links; IDs of owner and group; access permissions; and time stamps for creation, last access, and last modification).

**interrupt** In computer usage, a request to the main processor from one of its devices for processing time.

**I/O** Input/output; the transfer of data to and from a computer and its devices.

**italic** In printing, a style of typeface that is characterized by slanting of vertical character strokes to the right and rounding of corners.

**justify** In text-formatting, to align the characters on the margin. Since most text is justified on the left margin automatically, *justify* often becomes synonymous with *right-justify.*

**kernel** The program that provides the basic interface between UNIX and the machine's hardware. When UNIX is transported to a new computer system, the kernel is all that needs to be modified. The kernel resides in a file that is named either /unix or /xenix.

**Korn shell** The third, and latest, major shell program in use on UNIX systems. The Korn shell, named after David G. Korn, was designed to combine the major advantages of the standard shell (the Bourne shell) and the C shell.

**language** A set of commands, names, and rules of syntax that allow you to write a computer program. Languages are classified as either *low-level* (computer-oriented) or *high-level* (task-oriented). Machine language and *assembly language* are low-level languages; BASIC, FORTRAN, COBOL, and Pascal are high-level languages. Other languages, like C and Modula-2, fall somewhere in between, and are sometimes called *middle-level* languages. A language may also be called a *computer language* or a *programming language*.

**laser printer** a printer, based on the technology used in photocopiers, makes it possible to print graphics and text (in different fonts) on the same page.

**link** Attachment of a file to a directory. UNIX allows a user (or several users) to attach one file to more than one directory. No matter how many links there are for a file, there is still only one copy of the file in the system.

**listener** In networking in general and in the Remote File Sharing (RFS) system in particular, a program that notifies stations of the presence of a message on the network.

**local** In computer usage, applicable only to one segment of a file or program; as opposed to *global*.

**login name** The name you use to log into your UNIX system; the name used to indicate ownership of your files.

**machine language** The set of primitive codes that operate a computer's main processor, or *CPU*. (See also *assembly language* and *language*.)

**macro** A set of commands combined to form a single, more comprehensive command. In **nroff** and **troff** formatting, a user can define a macro from existing requests with the **.de** request; in the C shell, a user can construct a macro from existing UNIX commands with the **alias** command.

**map** In computer usage, to re-route program execution from one memory address to another; to assign a group of characters or functions to a key on the keyboard. In **vi**, you can design your own key functions with the **:map** command.

**memory** The internal area of a computer where programs, text, and data are stored for immediate processing; as opposed to *mass storage media*, where files are stored for long-term preservation. The UNIX operating system frequently writes program files from disk to memory, a proceess called *swapping*.

**modem** Modulator/demodulator, a device that modulates data (converts it from digital to analog) before sending it over a telephone line and demodulates data (converts it from analog back to digital) after receiving it from a telephone line. (See also *analog* and *digital*.)

**mount** In system administration, to make a file system known to the UNIX operating system. UNIX provides two commands, **mount** and **umount**. In System V, Release 3.0, a system administrator can use the **-d** option of the **mount** command to obtain access to file systems on other UNIX systems within a network. (See Part VII for further information.)

**multiplexing** In computer applications in general, the process whereby a series of data segments is processed in increments using a rotation scheme. For example, many frequencies of data can be multiplexed over a single channel, rather than transmitted sequentially. Swapping and demand paging are examples of memory multiplexing.

**name server** In the Remote File Sharing (RFS) system, the host that maintains records on the use of files by hosts inside and outside its domain; also called the *domain name server*. (See also *server*. )

**node** In networking in general and in the Remote File Sharing (RFS) system in particular, another name for a *host* or a *station*. (See *host*.)

**number** In a computer system, a number may be used to represent not only a quantity, but also a character or a machine code. The computer itself recognizes only two digits (low and high voltage in hardware design; 0 and 1 in software design), which form a *binary* (base two) number system. However, because binary numbers are so difficult to read, people who work with computers usually use either *octal* (base eight) or *hexadecimal* (base sixteen) representation for numbers. For example, the binary number 1011010 becomes 132 (octal), 5A (hexadecimal), or 90 (decimal), which is also the ASCII code for the letter Z.

**octal** Base eight. (See *number*.)

**owner** The user who created a file.

**page** In text-editing, to move to either the previous or the following screenful of text; in memory management in general and in UNIX System V, Release 3.0 in particular, a segment of a program (usually 2,048 bytes) that is transferred to memory to be executed.

**paging** See *demand paging*.

**path** In the UNIX file system, the sequence of directories from *root* to a file that indicate the file's location in the file system. Depending on your *current directory*, you may be able to use either a *full pathname* (an *absolute pathname* or a *relative pathname*) to identify a file.

**peripheral device** Same as a *device* or an *I/O device.*

**pipe** A connection that allows UNIX users to use the output of one process as the input of another. A pipe can also be called a *pipeline.*

**process** In UNIX, a command being executed at this moment. Each process has its own PID (process identifier) for reference.

**program** A set of instructions to a computer to carry out a procedure, written in a *language.* (See also *language.*)

**prompt** In computer usage, something that appears on the screen to solicit your next input. In UNIX, there are at least three possible shell prompts: `$` (Bourne shell), `%` (standard C shell), or `#` (super-user). In addition, the C shell allows you to design your own prompt, with an option for numbering your command lines for later reference (see Chapter 25, "Introduction to the C Shell," for details).

**quote** Same as *escape.* (See *escape.*)

**record** In text-processing, one entity (usually a line) presented to a program for processing. Each record is composed of *fields*, and the usual record separator is newline. The **awk** program allows you to specify your own record separator.

**redirect** In UNIX, to accept input from a source other than the standard input or send output to a destination other than the standard output. (See Chapter 4, "Using UNIX Commands," for details.)

**regular expression** In UNIX, a symbolic expression that allows you to search for a particular string of characters. It can specify characters via sets, ranges, and wild-card selection.

**return code** Same as *exit status.* (See *exit status.*)

**roman** In typesetting, a "plain" style of typeface that is neither bold nor italic.

**root** The primary directory of UNIX. (See *directory.*)

**scroll** In text-editing, to move the screen display either up or down one line at a time.

**server** In a communications network in general and in the Remote File Sharing (RFS) system in particular, a node that provides some service to the network. A *routing server* connects two different networks of the same architecture; a *gateway server* connects networks of different architectures by translating between their protocols; a *file server* make files available to other nodes; and in RFS a *name server* keeps track of the names of resources available for sharing.

**shell** The command interpreter for UNIX; a programming language that allows you to modify processing, handle input and output, and design your own UNIX interface. When you use the shell for programming, you produce a *shell script*, or *shell program*, which can be executed immediately. The shell is discussed in detail in Part V.

**software** A collective name for computer programs; as opposed to *hardware*, a collective name for the tangible components of a computer system.

**sort** In computer usage, to take a list of items as input and place them in a given order as output. The UNIX **sort** command is described in detail in Chapter 13, "Searching and Sorting."

**spawn** Same as *fork*. (See *fork*.)

**special character** A keyboard character that has a special meaning to the UNIX shell; also called a *metacharacter*. For example, caret (^) and dollar sign ($) mean beginning and end of line, respectively.

**special file** A file that stands for a device; also called a *device file*.

**standard input** The expected source of input to the UNIX shell (the default is the user's keyboard). You can change the standard input with *redirection*. (See *redirect*.)

**standard output** The expected destination for output from the UNIX shell (the default is the user's screen). You can change the standard output with *redirection*. (See *redirect*.)

**station** In networking in general and in the Remote File Sharing (RFS) system in particular, another name for a *host* or a *node*. (See *host*.)

**sticky bit** In system administration, a bit that can be set to guarantee that a program is kept in the swap area; a useful feature for a program that is in constant demand.

**string** In computer usage, a sequence of characters. In UNIX, you can assign strings to *variables*, search for strings, and replace one string with another.

**super user** In system administration, a user who has unlimited access to all files in a UNIX system, usually to facilitate administrative functions.

**swap** To rotate programs through memory and disk while they are being executed, using a *swap area* (or *swap disk*) that is always set aside on disk for this purpose.

**termcap** A file that contains terminal names and descriptions used for Berkeley versions of UNIX ("terminal capability"). System V uses **terminfo**, but also supports **termcap**.

**terminfo** A collection of files that contain compiled terminal names and descriptions used to identify the features of various terminals for screen-oriented programs like **vi** ("terminal information").

**tty** An abbreviation for Teletype; used in UNIX to mean a terminal.

**typeface** In printing, a specific design for a set of characters that typically includes roman, italic, and bold versions of the characters. Typefaces are grouped into two categories, depending on whether or not the characters have decorative accents called *serifs*: *serif* and *sans-serif*. Times Roman, Century Schoolbook, and Bodoni are examples of serif typefaces; Helvetica, Futura, and Optima are examples of sans-serif typefaces.

**variable** In computer programming, a storage location with a name to which you can assign a string or numeric value for subsequent use. In UNIX, the C language, the shell, and the **awk** program allow variables.

**wild card** A *special character* (or *metacharacter*) used in a *regular expression* that can be used to match any of a set of characters (., *, or .*). The name comes from the Joker, which can be used in some card games to represent any other card in the deck. (See Chapter 13, "Searching and Sorting," for details.)

**work area** An area in main memory for text-editing; also called a *work buffer*.

**working directory** The directory in which you are working at a given moment; also called the *current directory*.

# Index