UNIX™ & XENIX® DEMYSTIFIED

LEE PAUL CLUKEY

87221
49

UNIX™ & XENIX® DEMYSTIFIED

LEE PAUL CLUKEY

TAB BOOKS Inc.
Blue Ridge Summit, PA 17214

FIRST EDITION

FIRST PRINTING

Copyright © 1985 by TAB BOOKS Inc
Printed in the United States of America

Reproduction or publication of the content in any manner, without express permission of the publisher, is prohibited No liability is assumed with respect to the use of the information herein

Library of Congress Cataloging in Publication Data

Clukey, Lee Paul.
UNIX and XENIX demystified.

On t p. the registered trademark symbol "TM" is superscript following "UNIX" in title and registered trademark symbol "R" is superscript following "XENIX" in title.

Includes index.
1. UNIX (Computer operating system) 2. XENIX (Computer operating system). I. Title.
QA76.6 C57 1985 001.64'2 85-2658
ISBN 0-8306-0874-5
ISBN 0-8306-1874-0 (pbk.)

Should You Read This Book?

Unix and Xenix Demystified is applicable to Unix and Unix-based systems like IBM's PC/IX, as well as to the popular Xenix, which is currently in use on a great many microcomputers and which will be used to operate the newly announced IBM Personal Computer AT. If you are apprehensive about Unix, you will find that *Unix and Xenix Demystified* provides easy-to-read and understandable explanations about Unix and what you will need to know about operating a computer with a Unix system.

This book has been written to help managers, computer salesmen, and individuals considering purchasing or who have recently purchased a computer operating with Unix, to gain a basic understanding of Unix without getting involved with the more stringent technical details found in more advanced Unix textbooks.

It is written for the person who uses a computer for word processing, accounting, and other office application programs, but who would like to gain a better understanding of what Unix is doing in their computer or who would like to learn how to begin taking greater advantage of the capabilities available in their Unix system.

Unix and Xenix Demystified is based on an actual Unix instruction course. It is the only Unix text dedicated to a tutorial presentation of the Unix system. The text began as class notes to a "Beginning Unix" class taught at the Computer Training Centers, in Los Angeles, California. Our instructors are currently teaching Unix at UCLA. Educators will find the book an excellent beginning course for teaching Unix, including step-by-step instructions for laboratory exercises.

The appendices contain some useful information, such as a composite listing of Unix commands from Version 7, System III and V, Berkeley 4.2, Xenix, PC/IX, and UniPlus +. Appendix C provides a cross-reference matrix of the systems in which these commands may be found. You will find this information very useful for comparing the various Unix systems, to determine which commands have been added to each version of Unix and the differences between these systems.

At this point thanks are in order to Microsoft Corporation for providing the material on Xenix which is reprinted in Appendix D. Thanks especially to my wife for enduring countless late-night hours of writing, rewriting, reading, and editing. And finally, thanks to Irv Barilia and Tony Turgeon at the Computer Training Center in Thousand Oaks for technical consultation. These are just a few of the people who helped make *Unix and Xenix Demystified* possible.

Introduction: Learning to Use Unix

The aura surrounding the Unix system causes many potential users of Unix to shy away from it. For example, Unix training has been costing upwards of a thousand dollars for a week-long series of classes, and it is currently taught at only a few schools and colleges throughout the country. In comparison to DOS, which is taught in less than a day and seldom costs more than a hundred dollars, Unix appears as if it must be difficult to learn to use.

Unix textbooks are often lengthy, some as long as 400 to 500 pages; their appearance would suggest that it is important, if not required, to read and master it all in order to learn to use Unix. Even more lengthy are the Unix programmers' manuals from Bell Laboratories (AT&T), which are the bible on Unix. They are so lengthy and difficult to understand that even experienced computer users not only shy away from Unix, but run from it.

Opponents of Unix print banner headlines in magazine articles warning that Unix is very large, very cryptic, "user-unfriendly," and on and on. They claim Unix is a program written by programmers for programmers, and that it is totally unsuited for microcomputer installations or usage.

The pro-Unix articles also have not done much to help persuade users to learn to use Unix. Some of the so-called "Introduction to Unix" articles make Unix look more like an intermediate- to advanced-level course in computer programming.

It should therefore come as no surprise that Unix has been shunned by so many people. Had it not been for an error in my purchase order, I too would probably be standing among the ranks, throwing rocks at Unix, singing hymns of praise to DOS and CP/M, and bragging about making a wise decision.

However, after having learned to use Unix, I feel it was definitely worth the time. I think that computer installations using Unix are offered vastly superior capabilities and better software packages. I also found it as easy to learn to use Unix as lesser systems. My only problem was finding written material to study. I was not about to spend a thousand dollars, and at the time there were only three Unix books available.

My purpose for writing this book is to help you over the most difficult hurdles in getting started with Unix, first by explaining how to go about it, and second by providing a logical and complete introduction to the system. This will prepare you for reading more advanced textbooks and using the Unix programmers' manuals.

In the process I hope I will also be able to dispel the myth that the Unix system is difficult to learn. I believe that the Unix myth, like all myths, will no longer maintain its mystical air once it is exposed to the light of understanding.

By familiarizing you with the operation of the Unix system and by explaining what you need to know to use Unix effectively, this book will convince you that learning to use Unix in your everyday business needs is within the competence of anyone capable of learning to use a word processor or electronic spreadsheet program.

The presentation approach I take gives you a simplified overview of the Unix system before getting involved in any details on using Unix itself. If you first understand what is going on in your system, as well as the significance of this information to your operating needs, a discussion on how to use Unix will be of much more import. This is the main reason that the text is divided into two parts, a tutorial and lab exercise sessions. Although invariably you will learn more by using Unix than you ever will by just reading about it, going through the motions without understanding the significance of what it is you are doing, why you are doing it, and/or of what value the exercises will eventually have for you will limit your ability to retain this information.

Pushing buttons on your terminal keyboard and having the described responses appear on the terminal screen is satisfying. You will soon realize, however, that because the commands have no meaning to you, you will have to relearn these commands several times before they become a part of your vocabulary. You will find that your retention will increase if you reverse the learning process, learning first what you are doing before you do it.

I have tried to write this book for the beginning user in environments other than software development; it is sometimes difficult, however, to know just where to end a technical discussion and how much detail is too much. It is impossible to find the exact point that will satisfy all readers. Thus, when I felt that the detail might be beyond the needs of the beginning user or cause some apprehension, yet be necessary to the basic understanding of the subject, I have isolated it in such a manner that it will not interfere with the presentation of the main line of the subject.

Through reading *Unix and Xenix Demystified,* you will learn that it is necessary to learn to use only a small fraction of the Unix commands to derive extensive benefits from it. In fact, in the typical office installation the services provided by the Unix system probably will not even be apparent to you. From what you will see on your terminal, your system could be operating on something called

"Whatzis." Ninety percent or more of the time you spend operating your computer will be spent with application programs such as word processors; the remaining 10 percent, tasks like adding users or moving files between users, are likely to be performed with system administration programs provided by the computer manufacturer, programs designed for individuals who have little or no computer operating experience.

You might thus ask, "Why should I bother learning about Unix at all?" The answer is that at some time it may be necessary for you to correct a problem or effect some changes to your files that would be difficult or even impossible to do with the program functions available to you in the manufacturers system administration program, the word processor or electronic spreadsheet, or other programs that you have on your computer. If you have some familiarity with the Unix operating system itself, you will be more capable of correcting such problems.

An Outline for This Book

When I begin to read a new textbook I usually start by examining the table of contents of the book in order to get an overall perspective of the subject. This gives me a kind of road map of the material and helps me to develop an overview of the subject, an overview I can use to relate the detailed components.

Unfortunately, most tables of contents are usually filled with nonsignificant chapter and subject titles. Sometimes they are too cute; "The Old Shell Game," for example, is not very descriptive to a new user. Other times they are too detailed. For this reason I have included a graphic overview (Fig. I-1), which will enable you to see, almost at a glance, what is in *Unix and Xenix Demystified*, and the interrelationships among the various subjects.

The diagram, read from the top down, provides an overview of the Unix system with respect to the modular components which make up the Unix system, as follows:

Utilities. These are shown divided into eight categories, based on their specific functions and services.

Operating System. This is composed of a kernel, system calls, and a shell, which in turn contains three types of commands:

Built-in commands
Metacharacters
Conditional commands

The first type of command is included in the shell in order to expedite processing frequently used commands. The second and last types are used in conjunction with the development and entry of complex Unix commands and shell programming.

Subroutine Libraries. As the name suggests, these contain libraries of subroutines used principally in the development of C- and FORTRAN-based programs and in conjunction with the operation of the system.

Special Device Files. These are most often used to control flow of data between the computer and the prepherial devices attached to the computer, e.g., printers, disk drives, modems, etc.

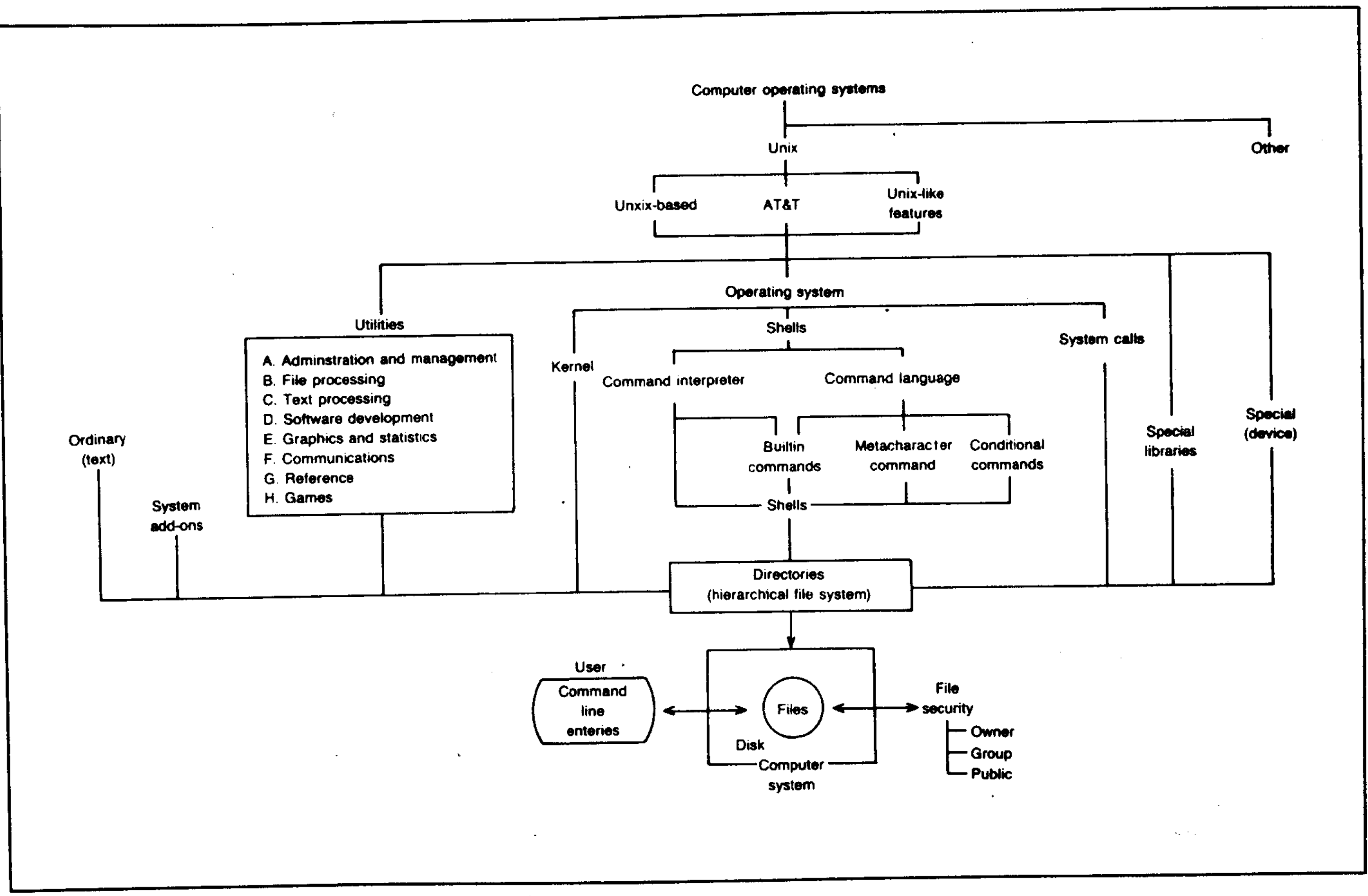


Fig. I-1. A graphic overview of *Unix and Xenix Demystified.*

From the bottom up, the diagram displays the use of the Unix system. The user creates a command line—the entry of a command at the terminal—which in turn calls upon the files in the computer's disk storage to operate the computer. The files in the system are stored in a hierarchically structured arrangement, based upon the disk address locations contained in the directories.

All data in the computer—utilities, the Unix operating system itself, the libraries of subroutines, special device files, system add-ons such as word processors as well as ordinary text—are all contained in files on the computer's disk. All of the locations for these files are contained in directories.

The composite diagram shows the interrelationship among the discussions on the physical breakdown of a Unix system and the interaction involved in using the Unix system to operate your computer.

Contents

Should You Read This Book? vi

Introduction: Learning to Use Unix viii

PART 1 A TUTORIAL APPROACH 1

Chapter 1 Introduction to Unix Systems 2

Computer Operating Systems 2
Utilities and the Unix System 5
Generic Unix Systems 7
AT&T Unix Systems—Unix-based Systems—Unix-like Systems
How Big Is a Unix System? 12
Unix System Add-Ons 13
Systems Nomenclature 14

Chapter 2 The Structure of the Unix System 15

The Unix Operating System 15
The Kernel—System Calls—The Shell
Unix Utilities 21
Applications Programs 22
Software Levels 22

Chapter 3 The Unix File System 26

Auxiliary Memory 26
The File 29
Ordinary Files—Directory Files—Special Files—File Contents—File Names
The Hierarchical File Structure 32

Building the Hierarchical Structure—Purpose of the Structure—The Hierarchical Structure in Operation—Directories in the File Structure—Typical Top-Level Directories
Pathnames 40
File Security 42
Registered File Owner—The Group
Linking Files 45
Mountable File Systems 47
Summary 47

Chapter 4 Commands 48

Shells 48
System Shells—Menu Shells—Application Program Shells—Selecting a Shell
The Command 55
Command Formats—Command Components—Command Terminology
Types of Commands 61
Utilities—Shell Commands—System Calls—Subroutines and Libraries
Summary 62

Chapter 5 Shell Command Language 63

Metacharacters 63
The Command Line 64
Using the Shell Command Language 65
Multiple-Command Command Lines—Pipes—Pipe Lines—Filters—The Tee Command—Background Processes—Subcommands and Command Groupings

Chapter 6 Introduction to Shell Programming 73

Command Files vs. Shell Scripts 74
Running a Shell Program 74
Shell Conditional Commands 76
Summary 76

Chapter 7 Tutorial Summary 77

PART 2 HANDS-ON UNIX 82

Chapter 8 Introduction to Lab Exercises 83

An Overview 83
Some Things You Should Know 84
The Shell Prompt—Command Interpretation—Entering a Command—Halting a Command in Process—Control Keys

Chapter 9 Login and System-Level Security 88

Login Name 89
Password 90
Logged-On Acknowledgment 91
Log Off 92

Chapter 10 Locating Directories and Files 93

Who Am I 93
Print Working Directory 94
Change Working Directory 95
List Directory Contents 97
Changing Directories with a Metacharacter 105

Chapter 11 Making Files and Directories 109

Making Files with the Redirection Command 110
Making Files with Appended Redirection 115
Making Files with the Move Command 117
Making a File with the Copy Command 117
Making a Directory 120
File Creation Using Pathnames 124

Chapter 12 File Security 129

Changing File Security 129
File Copying Permission 133
Changing File Ownership 135

Chapter 13 Linking Files 139

Chapter 14 Using Shell Metacharacters 144

Redirect Input 144
Multiple-Command Command Lines 145
Pipes 146
  Pipe Lines—Filters—The Tee Command
Summary 156

Chapter 15 Making and Using Shell Programs 157

Using Shell Variables 159
Shell Programs with Pipes 166

Chapter 16 Removing Files and Directories 170

Appendix A: Lab Exercise Command Summary 178

Appendix B: Consolidated Directory of Unix Utilities 188

Appendix C: Unix Utility Cross-Reference Matrix 204

Appendix D: The Xenix System 230

Index 242

Part 1
A Tutorial Approach

Chapter 1

Introduction to Unix Systems

"Unix systems are a coalition of computer programs which have been combined into one interactive system."
"Unix systems have the peculiar characteristic of becoming an industry standard that in and of itself lacks standardization."
"Seldom are two Unix systems the same."

These three statements express why Unix systems are so versatile and flexible—and at the same time so difficult to define and describe for the beginning user.

COMPUTER OPERATING SYSTEMS

The computer, terminal (keyboard and monitor), printer, telephone modems, and all of the other electronic gadgets you may have connected to your computer are referred to as the *system hardware* or the *computer system* (Fig. 1-1).

Computer systems in themselves have no capabilities to perform accounting or word processing, or solve engineering problems, any more than a telephone can dial a telephone number by itself. All of the capabilities you attribute to a computer are derived from the *software* (computer programs) that you install in your computer (Fig. 1-2).

If, for example, you have ever used an automatic telephone dialing unit, you know that you must first enter the series of numbers that make up the frequently called telephone numbers you wish to have available for use in the auto-dialer, before the unit can dial the number automatically for you. The series of numbers you enter into the unit can be likened to a computer program. It contains the

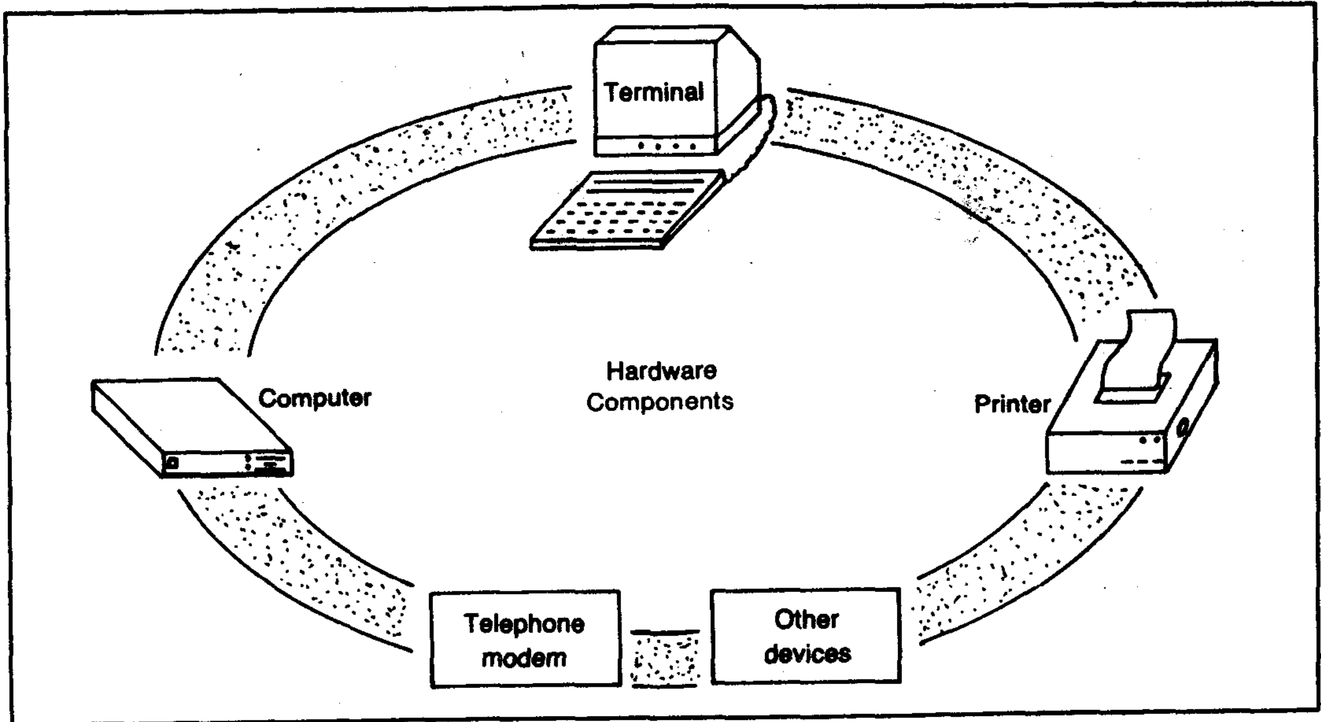


Fig. 1-1. The computer system.

dialing instructions (the telephone number) which, digit by digit, makes the unit place the call for you.

Computer programs are similar to the telephone number in our analogy. They are considerably more complex than a telephone number, but they perform the same type of operation in the computer as the telephone number does in the automatic dialing unit. Computer programs contain lines upon lines of instructions which together may serve as a word processor, an electronic spreadsheet, a database manager, etc. Like the telephone number entered into the automatic dialing unit, the purpose of the computer program is to make the machine perform some job for you automatically.

There are many families of computer programs you can install in your computer system. Word processors, accounting programs, and electronic spreadsheets are of the family called *application computer programs*. You are probably most familiar with this family of software because they are always in the spotlight. They probably are the major reasons you bought your computer.

A software family with which you may not be as familiar is the family called *computer operating systems*. You are most probably not familiar with computer operating systems because their operation is not readily apparent to you. However, if it were not for the work performed by the computer operating system, none of the more apparent application program software would be able to run on your computer.

The computer operating system (or simply the *operating system*) is the fundamental software required for operating any computer system. Unix, from Bell Laboratories (AT&T), is just one of the many operating systems available on the market. A few of the others whose names you may know are:

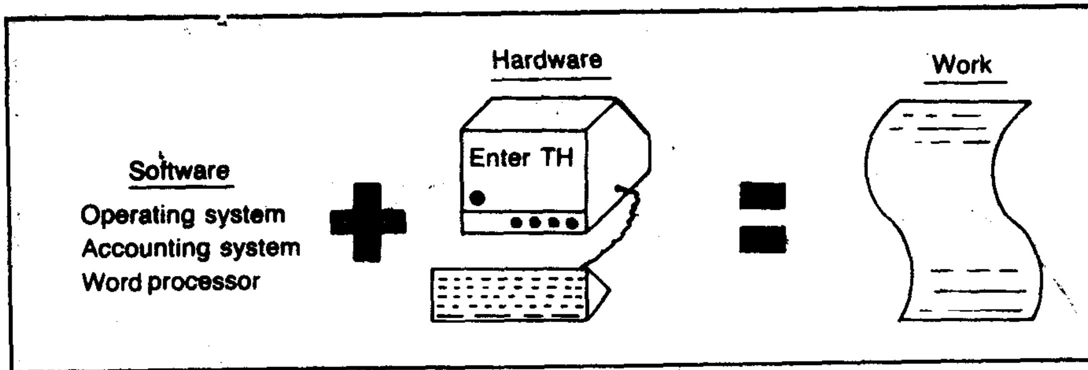


Fig. 1-2. Software drives the computer system.

CP/M	CP/M-80
MP/M	TRSDOS
VMS	Apple DOS
MS-DOS	Pick
DOS	DOSPLUS
CAPS	LDOS
CP/M-86	Apple ProDOS
MP/M-86	Tele DOS
PC-DOS	Commodore 64

An operating system, in very simple terms, is a library of computer programs, each of which provides the operating instructions that drive each basic computer system operating activity (Fig. 1-3). The computer programs in a Unix operating system contain the instructions for such basic operating activities as:

- Controlling the movement of data into and out of the computer's memory, called *random access memory* or *RAM*.
- Supervising the operation of the computer's central processing unit (CPU), keeping all programs isolated from one another, so that one program will interfere with the operation of another.
- Sending data to the peripheral devices connected to the computer, such as to the computer's printer for a printout, or to the terminal monitor, for visual display, etc.
- Managing the storage of data in the computer's *auxiliary memory* (disk and tape drives), for example making sure that one file is not inadvertently erased the information in another.
- Running the other computer programs installed in the computer, such as commands, word processors, accounting programs, database programs, etc.
- By means of a security system, keeping all data and programs uniquely identified, keeping unauthorized individuals from gaining access to the system or users from tampering with one another's files, and keeping a program that has not been debugged from running rampant through the computer and destroying other files and programs.

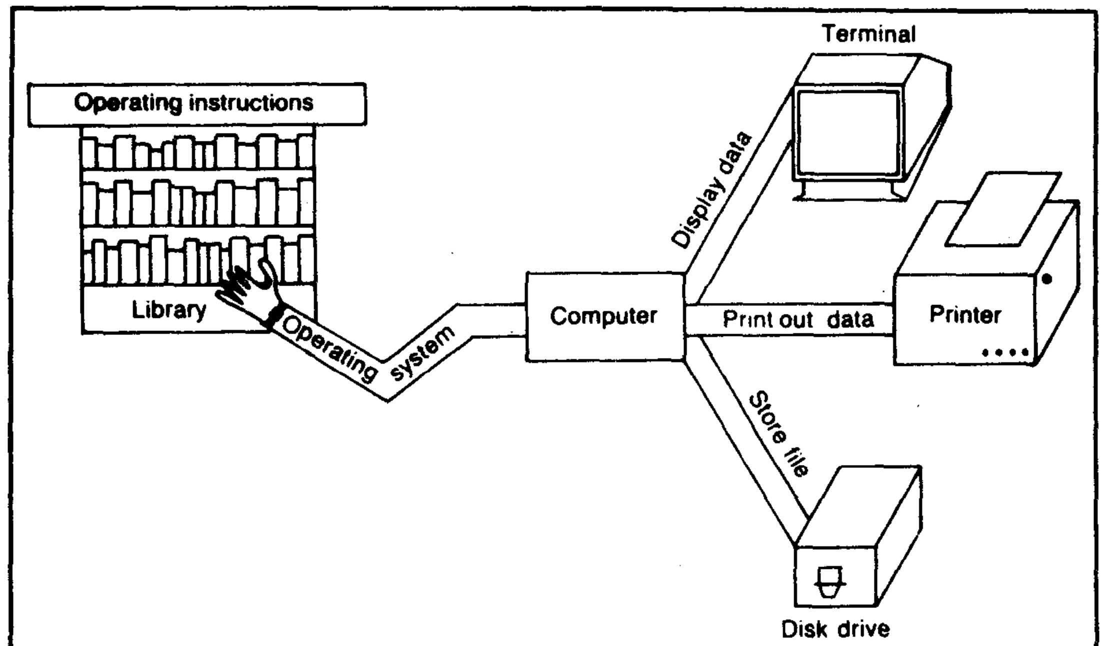


Fig. 1-3. The computer operating system manages hardware operations.

Not all operating systems are capable of performing this list of computer operating activities, or the operating systems may have been specifically designed for a particular computer, type of computer, and or application. The Unix system was designed for general-purpose usage. It is used for operating a variety of sizes (micro to mainframe) and types of computers, and supporting the operating requirements of most application assignments.

UTILITIES AND THE UNIX SYSTEM

The computer programs in the operating system were designed to be operated primarily through *commands*. Commands are merely computer programs that you run to provide the instructions that tell the operating system how to carry out a task. A command to direct the operating system to print out some information on the printer is a typical example of the type of task command programs can perform.

We might describe commands as "user support" computer operating programs. Commands comprise a library of fundamental and often-needed operational services. In the Unix community they are often referred to as a "tool box of utilities" because they provide utilitarian types of services. Hence, users started referring to Unix commands as *utilities*.

Some examples of the many types of functions or services that these Unix utilities provide for you are:

Editing (simple word-processing-type programs)
Data sorting
Checking for spelling errors
File maintenance and system administration
Communications between computers
Software development support for programmers

Technical Comment

C. Miller and R. Boyle in *Unix for Users* point out that, "In the strictest sense, Unix consists only of the kernel [core of the operating system]; it is perfectly possible to equip the kernel with an entirely different set of utilities, changing the outward appearance of the system altogether, and still have a Unix system; although this is occasionally done in tailoring Unix for particular applications, we shall usually use the term *Unix* in the looser sense, meaning the kernel and the 'standard' set of utilities which are distributed as part of almost all Unix systems."

Unix systems are known for their large libraries of commands (utilities). There are several hundred utilities available for purchase and installation on your computer. A typical library of Unix utilities may represent upwards of 90 to 95 percent of a Unix system's software (Fig. 1-4).

The simplest Unix system could be the operating system itself. Utilities are not explicitly essential to the operation of Unix or the computer, however, seldom, if ever, is a Unix operating system installed in a computer without at least a basic complement of about 40 utilities.

It is the utilities that provide the majority of the character and serviceability attributed to Unix systems. Some system installations may contain as many as several hundred utilities. Because of the nature of utilities and the great number of them contained in a system, they are often mistakenly considered to be an integral part of the operating system or to be the operating system itself.

In general conversation it is acceptable to be ambiguous with terms, but for our purposes here it will be necessary to be very precise in our use of terms. For clarity we will use the following definitions:

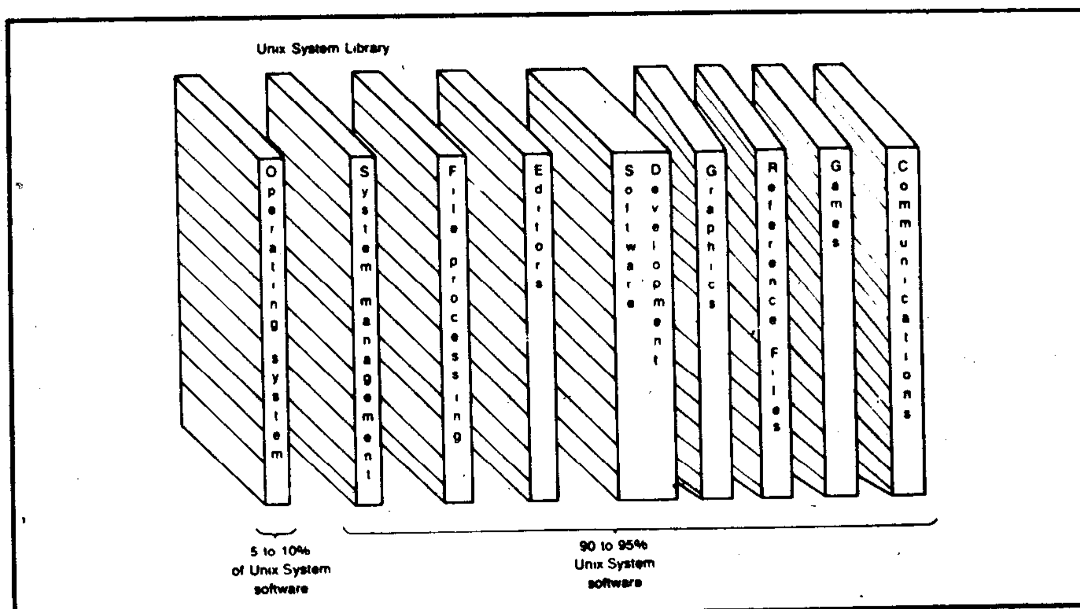


Fig. 1-4. The Unix system is a software library.

- The Unix operating system contains the software that controls the computer system.
- The Unix utilities contain the command software (utilities) that you use to drive the operating system.
- The combined Unix operating system and Unix utilities will be referred to as the *Unix system*.

The term *Unix system* is not intended to imply any particular size or system configuration—only that the system contains at least the basic Unix operating system.

GENERIC UNIX SYSTEMS

Since the development of Unix in the early seventies, a number of versions of AT&T's Unix, as well as a number of systems developed by other companies and referred to as "Unix-like," "Unix-compatible," etc., have been introduced to the market. Table 1-1 organizes and lists these systems according to their relationship to AT&T's Unix.

The first group of systems, "Unix Systems," contains the versions of the Bell Laboratories Unix. The next grouping, "Unix-Based Systems," contains the systems that were derived directly from the AT&T Unix computer program, and were developed under a licensing agreement from AT&T. The last section, "Unix-like" systems, lists those that have no direct relationship to the Unix computer program, but exhibit the characteristics generally attributed to the AT&T Unix system. These characteristics include:

- **System Structure.** The Unix system structure is basically a core operating system enhanced by a library of user-oriented computer programs called *utilities*.
- **Comprehensive Library of Functions.** Unix systems are generally assumed to have very comprehensive libraries of utilities, independent from the operating system itself.
- **System Flexibility.** A Unix system may be altered with remarkable ease by purchasing and/or creating utilities.
- **Hierarchical File Structure.** Unix systems employ an organized file structure in which files in common may be grouped, as opposed to a simple sequential listing method.
- **Multi-user Operation.** Unix systems are designed to manage simultaneous processing requests from several users.
- **Multi-tasking.** Unix systems are designed to manage the simultaneous processing of more than one command or computer program from a single user.
- **Redirection of Standard Input/Output.** The operating system can be instructed to redirect the input/output of data to or from files, instead of

Table 1-1. Versions and Types of Unix, Unix-based and Unix-like Systems.

Unix Systems:

Version 5
Version 6
Version 7
32 V Unix
System III
PWB/Unix
System V (Release 2, Version 3)

Unix-based Systems:

Berkeley Unix University of California at Berkeley 4.1bsd and 4.2bsd

AUROS	Auragen
CP/IX	IBM
Eunice	Wolongong
Fos	Fortune Systems
GENIX	National Semiconductor
HP-UX	Hewlett-Packard
IS/1	Interactive Systems
IS/3	Interactive Systems
IN/ix	Interactive Systems
OSx	Pyramid Technology
PC/IX	IBM (Interactive Systems)
PERPOS	Computer consoles
Pro/Venix	Digital Equipment Corporation (DEC)
Rainbow Venix	Digital Equipment Corporation (DEC)
SERIX	COSI and CMI
Sys3	Plexus Computers
TSS/Unix	IBM
Ultrix-11	Digital Equipment Corporation (DEC)
Ultrix-32	Digital Equipment Corporation (DEC)
UNI-DOL	Science Management Corporation
UniPlus+	UniSoft
UniPlus+ System 5	UniSoft
Unisys	Codata
UNITY	Human Computing Resources
UNX/VS	Data General
UTS	Amdahl
U-II	Unidos Systems
Xenix	Microsoft
Venix	VenturCom
Venix/86	VenturCom
VM/IX	IBM (Interactive Systems)
Zeus	Zilog

Unix-like Systems:

Coherent	Mark Williams
Cromix	Cromemco
Idris	Whitesmith Ltd.
Micronix	Morrow Designs
PNX	Perq Computers
QNX	Quantum Software
QNIX	Quantum Software
Regulus	Alycon
uNETix	Lantech Systems

the standard input (the terminal keyboard) and the standard output (the terminal monitor).

- **Data Channeling.** Unix systems provide interprocess connectors, called *pipes,* which channel the output of one process (e.g., commands, computer programs) directly to the input of the next, without the need for creating intermediate files.
- **Shell Programming.** The Unix operating system has a self-contained "operating language" which can also be used to create new commands (utilities).

These Unix features will be further described and explained in more detail as we continue through the book.

AT&T Unix Systems

The approximate dozen versions of Unix systems included in this group, up to a proposed Unix System VI (anticipated release date 1985), have all been developed by Bell Laboratories (AT&T) since the early seventies. Figure 1-5 shows the genealogy of these AT&T Unix systems.

The significant differences among this group of Unix systems is found in the number and mix of system calls and other modifications to the operating system, and the number and mix of utilities. These differences are perpetuated in the Unix-based systems that use these variations of Unix in developing their own systems.

Unix-Based Systems

Unix-based systems are direct variations of the AT&T Unix system. Figure 1-6 provides a genealogy of two prominent Unix derivative systems, Xenix and the new PC/IX. Version 7 and System III form the basis from which almost all of the currently available Unix-based systems, sold by other companies under their own system names, were derived.

This group of Unix-derivative systems employ the same basic Unix operating system, contain a mixture of the utilities from the AT&T Unix and Berkeley Unix systems, and often include, in addition, utilities created by the companies developing these derivations.

These systems have been created by computer hardware and software companies in order to meet the operating specifications of specific computers or types of computers, to satisfy the needs of a particular market or application, or to provide user supports not available with AT&T's Unix. The latter category could include enhancements such as user administration programs, user menu systems, etc., which improve the quality of their system useability—often referred to as the "user-friendliness" of the system. They might also design their system to take advantage of the operating characteristics of their own computer, and thereby run more efficiently.

One of the most popular Unix-based systems is Microsoft's Xenix, developed as a service for microcomputer companies who did not have sufficient in-house skills for interfacing (called *porting*) AT&T's Unix to their own microcomputers. In addition to providing Unix ports, Microsoft has made its product friendlier to the

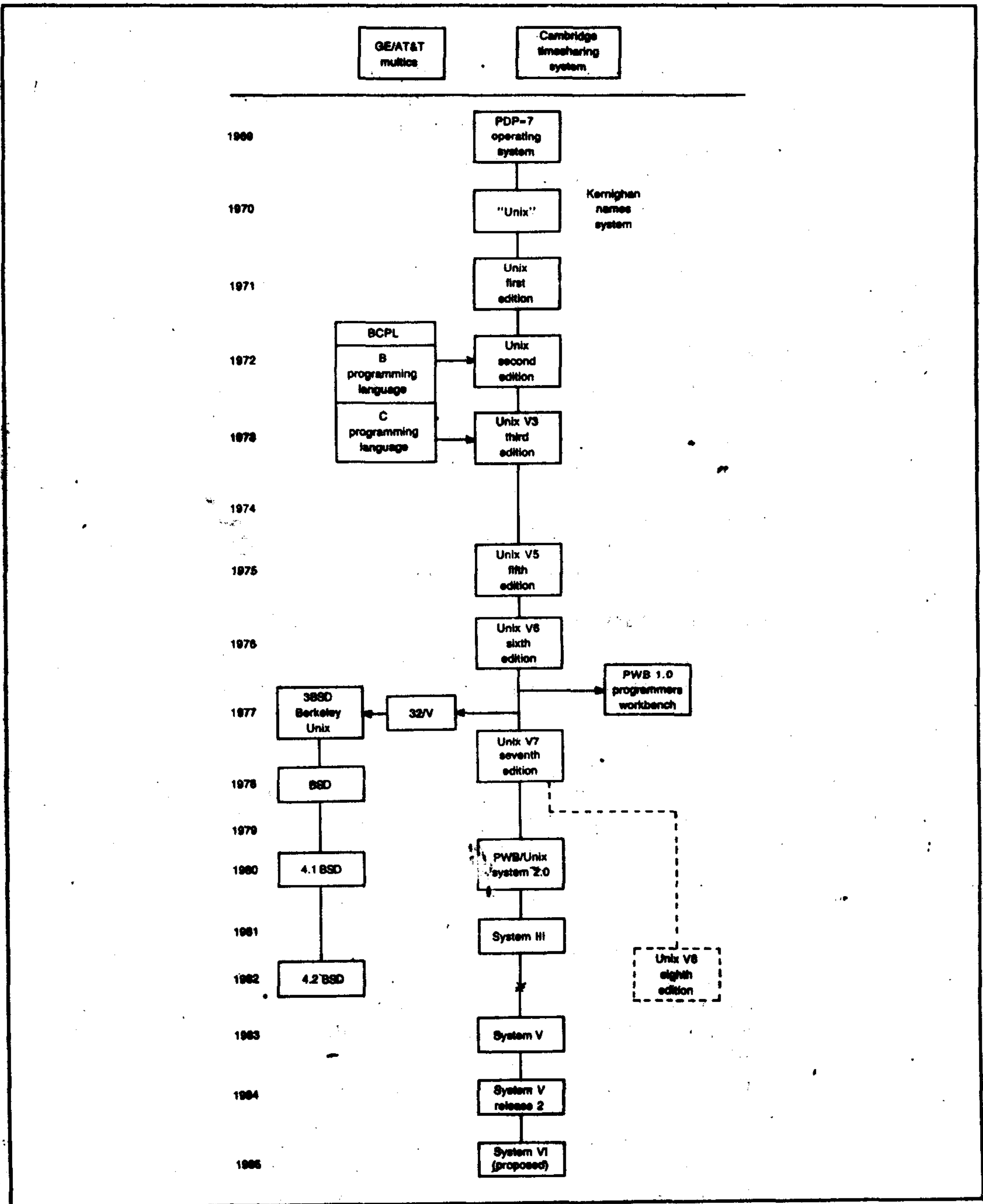


Fig. 1-5. Genealogy of the Unix system.

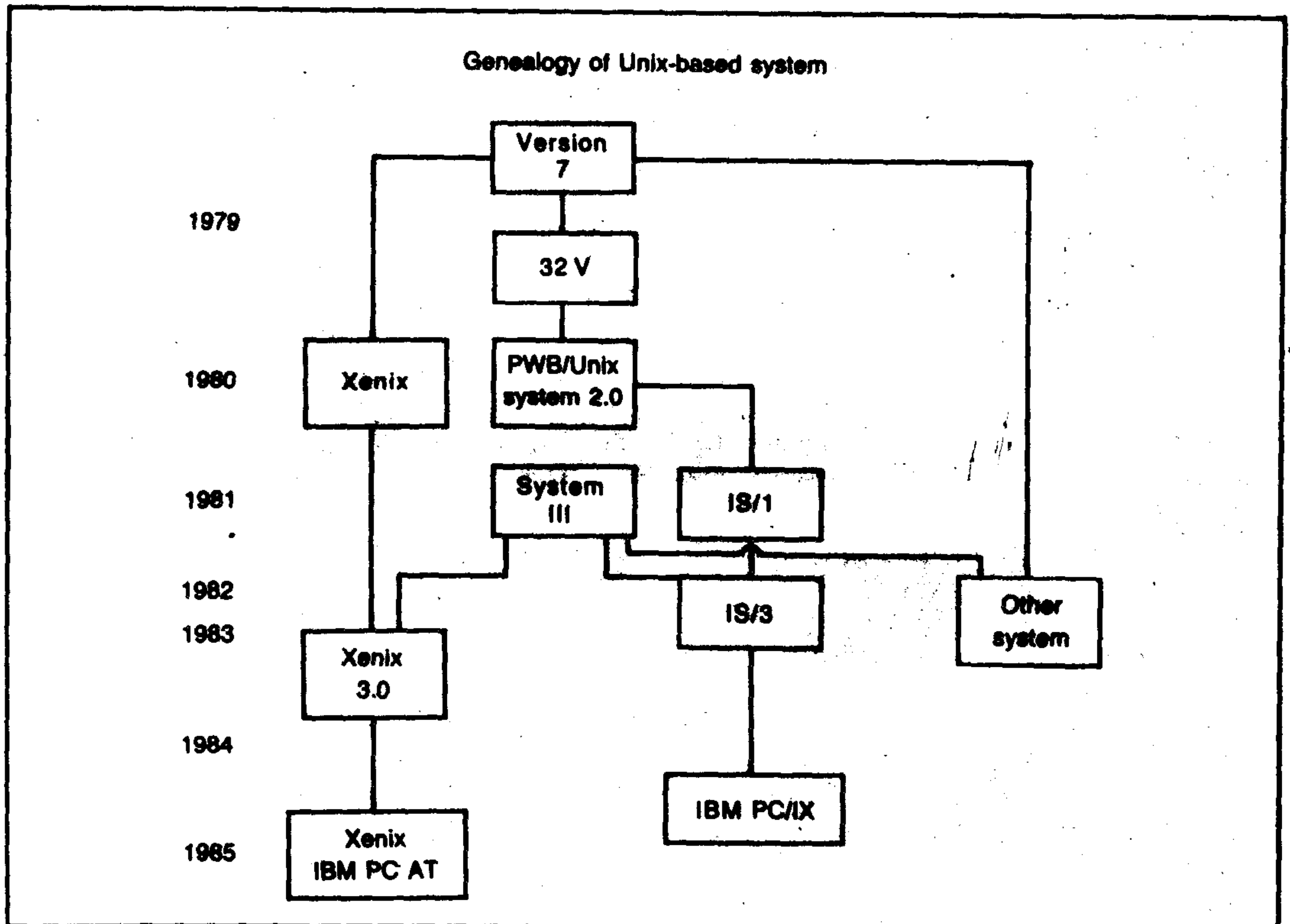


Fig. 1-6. Genealogy of the Unix-based systems Xenix and PC/IX.

type of user to whom microcomputers are usually sold. Microsoft has altered Unix by eliminating some of the less used and more complicated Unix utilities, adding some of the Berkeley Unix utilities, and by creating some of their own.

The differences between the AT&T Unix system and the Unix-based systems, and among the group of Unix-based systems themselves, is generally found in the number and variety of utilities included in each of these systems, and in the way that some of them may have been reprogrammed to operate in the computer. As an example, Appendix D contains a detailed listing of the functions and utilities contained in the Xenix system, so that you can get some idea of the capabilities in Unix systems.

From a user's point of view, Unix-based systems are for all intents and purposes the same as using an AT&T Unix system. The information in Unix textbooks and manuals is generally equally applicable to these systems. From a programmer's point of view, however, there may be some significant differences between Unix and Unix-based systems. I suggest that programmers contact the individual software vendors to determine if the differences will affect the programs that they are developing.

Unix-like Systems

Computer hardware and software companies in addition to AT&T have developed their own Unix-like systems. Some of the reasons they have gone to the trouble to produce their own systems are:

- To produce a system expressly suited to the operating characteristics of their computer.
- To avoid the cost of a Unix license and the restrictions imposed by AT&T.
- To avoid surprise changes and having to rely on AT&T for future developments and support.

Unix-like systems are not in any way related to Unix. The only reason that they are mentioned here is because they are characteristic of "Unix" systems and because you should be aware that they are significantly different from AT&T's Unix.

These comments should not be interpreted to in any way indicate that you should not consider buying one of these systems. For the reasons mentioned above they may be better suited to your needs. PC-DOS 3.0 and 3.1, used to operate the IBM PC-AT computer, are not comparable with the systems included in this category of operating systems but do provide a number of features similar to those of Unix systems. For those of you considering buying DOS 3.0 and 3.1, the Xenix system also offered is no more difficult to learn to use.

A more explicit discussion on the types of Unix systems could be presented, but this information is probably sufficient to give you an insight into the nature of the variety of Unix systems. This information should also help you to understand that there are a great number of software packages that used to operate computers that are referred to as Unix systems.

HOW BIG IS A UNIX SYSTEM?

At this point you might surmize that the term *Unix system* is loosely applied to any size system with any combination of Unix utilities, and you would be correct. A Unix system does not necessarily have to include all of what we might call the commercially available Unix utilities. Nor do Unix systems have to include the same mix of utilities. When you buy a Unix system, you may be getting:

- A very basic, "bare bones" version of the Unix system, which generally includes the computer operating system and about 40 of the most widely used utilities.
- A comprehensive library of utilities, called by some companies and users a "full-up," "full development," or "software development" Unix system. These usually contain several hundred utilities, many of which are useful only to experienced computer programmers.

Some computer companies sell stripped-down and/or "unbundled" Unix systems in order to be able to offer a more competitively priced package that still has the Unix label on it. The utilities not included in their basic package are usually

offered as optional items, just as automobile manufacturers offer power steering, air conditioning, etc., to be added to the basic vehicle.

AT&T is marketing its Unix System V unbundled, as is IBM's marketing of Xenix for their PC-AT. This is good from the standpoint that it will reduce the price of these systems, by enabling users to purchase selectively the parts of a system geared to their specific needs. However, this poses a problem for software developers. When a user installs an application program (such as a word processor) written for a system that is expected to contain specific utilities, and they are not present in the system, the application obviously will not work.

It should also be noted that there are no industry-standard system sizes or system size designators. The "full Unix" system sold by the Altos Computer Company requires approximately 7 to 8 million bytes (called *megabytes*) of disk storage space. The Pixel computer's complete offering requires about 17 megabytes, and Digital Equipment Corporation's system for its VAX-11/780 computer requires over 20 megabytes of disk storage space.

There are technical differences between these computer systems and the basis for measurement is not exactly equivalent; however, the technical differences only account for a portion of the difference in their sizes. The major difference in system sizes is due to more or fewer utilities having been included in one or the other of the systems, thereby indicating more or fewer system capabilities (functions) being made available. You should be aware of these differences when comparing and purchasing Unix systems.

UNIX SYSTEM ADD-ONS

User-Created Utilities. In addition to the commercially available Unix utilities, you may create your own utilities to enhance the usefulness of your system. User-created utilities, like vendor-created utilities, may be computer programs written in the C programming language or other languages, or they may be a series of existing Unix utilities joined together to create another utility (referred to as a *shell program*).

You can create your own utilities to run routine tasks such as backing up your data files, executing a payroll program, updating a data file, etc., so that you do not have to enter repeatedly the series of Unix commands required to perform these tasks.

Many experienced users generally regard any and all utilities, regardless of whether they are purchased or "home grown," to be a part of the Unix system. I suspect that this acceptance stems from the early Unix development days, when all utilities would have been considered "home grown." It was not until the mid-seventies that Unix took on its present commercial character, thereby establishing some distinctions between commercially sold and user-developed utilities.

Application Computer Programs. Application computer programs comprise a family of software with which you are probably most familiar. This family includes computer programs like word processors, accounting programs, electronic spreadsheets, and hundreds of other special-purpose computer programs. Application programs are similar to utilities in that they are software that you use to operate

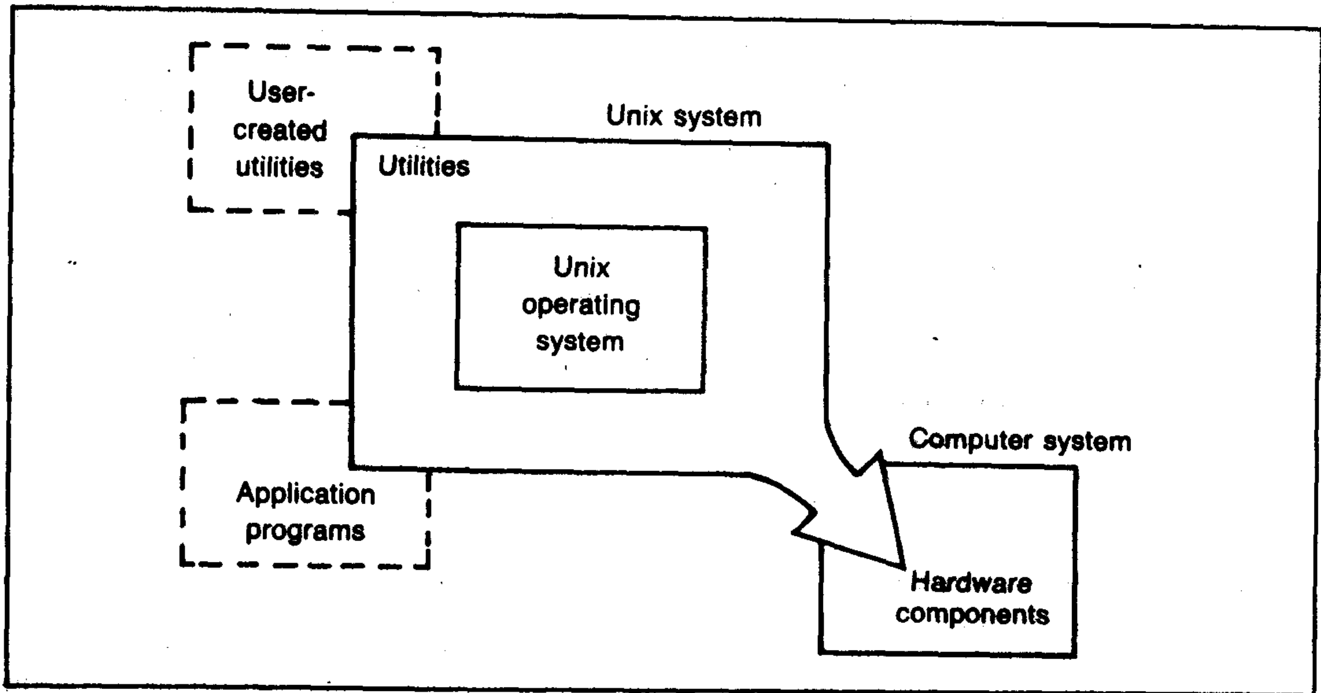


Fig. 1-7. The system.

your computer system. They differ in that they contain software especially suited to the purpose of the application.

Applications packages contain libraries of computer programs, many of which access the utilities in the Unix system in order to perform the tasks they are programmed to perform. These programs use utilities in much the same way that you would use a utility—by entering a command. Application programs are addressed here because they are important to your understanding of their operation in conjunction with the operation of the Unix system.

SYSTEMS NOMENCLATURE DEFINED

Throughout this part of the book we have been referring to several types of "systems." Each represents a group of interacting hardware and/or software components that represent a fundamental unit (Fig. 1-7). There have been four systems discussed:

- **The computer system.** This includes all hardware components, such as the computer and all the peripherals (printers, terminals, etc.) attached to it.
- **The Unix operating system.** This is the software responsible for the basic operation of the computer system.
- **The Unix system.** This refers specifically to the software comprising the operating system and the utilities. It may also include user-created utilities and application programs.
- **The system.** This includes all hardware and software.

Chapter 2

The Structure of the Unix System

If you have ever tried to read the *Unix Programmers' Manual* or one of the Unix textbooks, you have discovered that the Unix system appears to be very large and complex. This is primarily because Unix systems are composed of a lot of small, independent computer programs. That is, Unix systems are composite systems of computer programs, not a solitary program.

After you work with Unix systems for a while, however, you will begin to see that while these systems are large, their appearance can be greatly simplified by properly organizing and categorizing the component computer programs. In this chapter we will break down the operating system into its component parts and categorize the various utilities according to their individual function in the system and/or the service they provide to you for operating the computer. Figure 2-1 graphically portrays an overview of the categories of software components that make up a Unix system. The diagram is designed to give you a topical perspective of the components of a Unix system and their respective interrelationships in forming a system.

THE UNIX OPERATING SYSTEM

In the center of the diagram is the computer system (hardware) enclosed by the operating system, which is delineated by the boundaries of something called the *shell*. The operating system is the heart of the Unix system. It contains all of the software required for the operation of the computer system and serves as a bridge between the operation of the computer hardware and the user.

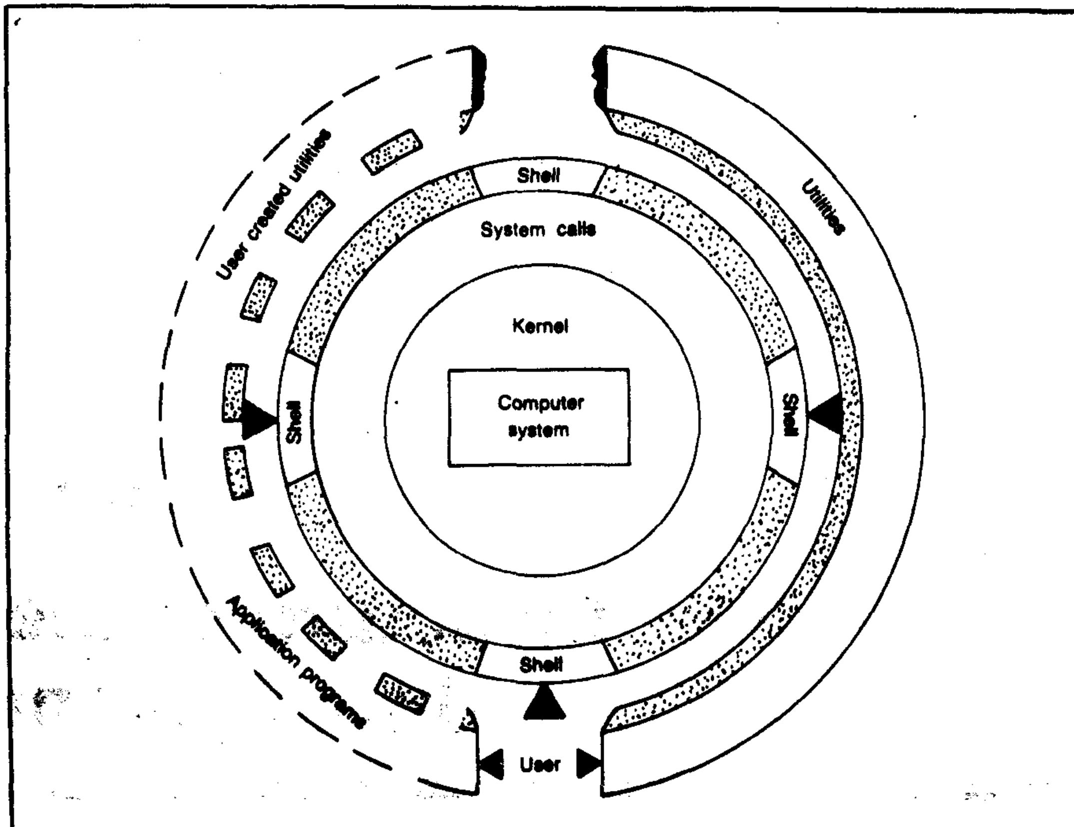


Fig. 2-1. Unix system interrelationships.

The operating system greatly simplifies the operation of the computer system for the user. Were it not for operating system software, the operation of a computer would be beyond the capabilities of all but highly trained technicians.

All Unix systems have similarly structured operating systems, with the exception that parts of the operating system may have to be custom-tailored to accommodate the particular hardware operating specifications of each manufacturer's computers. (The customization process is referred to as *porting* the Unix system to a computer.)

A Unix operating system is composed of three uniquely identifiable parts: a *kernel*, a *library of system calls*, and a *shell program*. Figure 2-2 is a simplified graphic illustration of the interactive operation of the components of the operating system.

- **Kernel.** This is the central part of the operating system. It controls the electrical and mechanical operation of all of the components in the computer system related to the processing of a computer program, data input and output, and the movement of data through the system.
- **Library of System Calls.** This contains the instructions which tell the kernel how to perform the commands that you enter at the terminal keyboard.

- **Shell Program.** This contains the operating system software responsible for initiating and supervising the processing of commands and the operation of the computer operating system.

The operation of these components is analogous to the procedure that might be followed in a parts fabrication company when an order for a part (the command) is received. When the order is received, the shop foreman (the shell) finds the specifications and blueprints (the system calls) for the part (the command) to be fabricated, and sends the package to the shop to be built. The shop worker (the kernel) takes the package of instructions and starts building the part according to these specifications. While the job is in the queue to be built, the shop foreman is responsible for the productivity of the shop by keeping the work queue filled and for making sure that each job is processed.

The Kernel

The kernel, in very simplified terms, is primarily a computer program for processing the instructions, called *primates,* that control the operation of the computer. Primates are the computer programs that provide the basic instructions for operating each activity of the computer system (hardware). The term is derived from the fact that each computer program contained in a primate is capable of interfacing directly with the computer system itself. The primate is the primary level of Unix system interface with the computer.

Some examples of the major activities that the primates are responsible for making the computer system perform include:

- The transmission of data to the printers, terminals, or other peripherals attached to the computer.

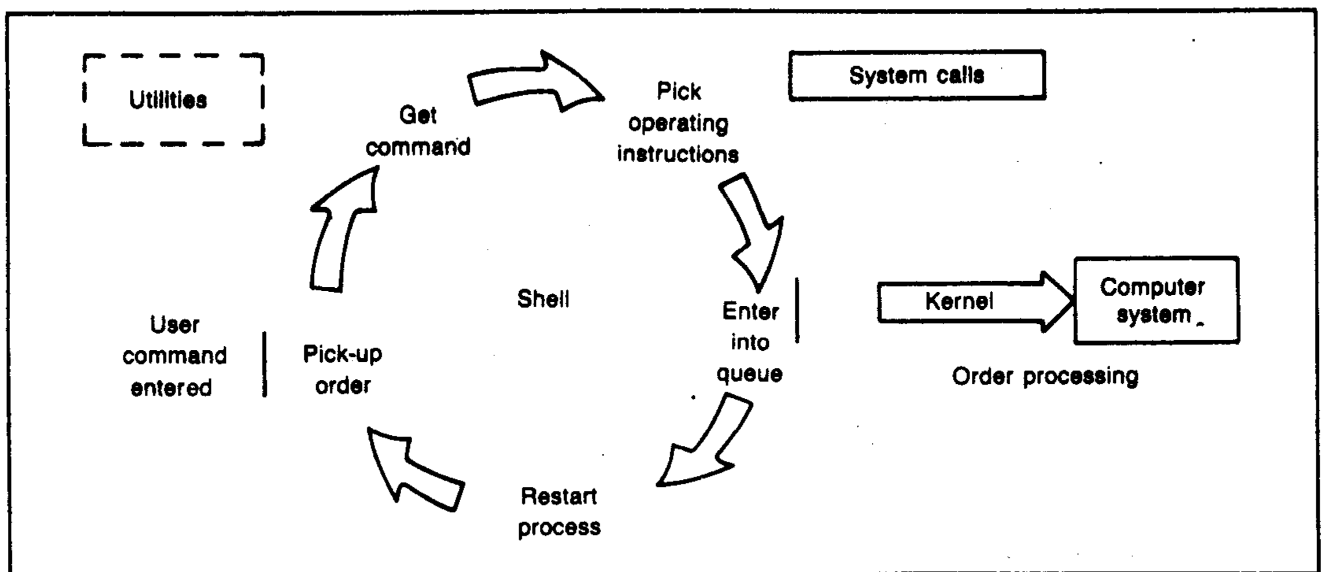


Fig. 2-2. Operating system command processing overview.

Technical Note: The Kernel

The primates, which comprise the bulk of the software of the kernel, serve two very important purposes in the Unix system:

- They serve as a means for "standardizing" the dissimilar operating specifications of different manufacturer's computers, reducing or eliminating the need to tailor the whole operating system to each computer.
- They form a "bridge" connecting the system software to the operation of the computer system, by supplying the software which can converse with both.

The primates serve as a standardizer between a somewhat uniform library of Unix system computer programs (utilities, system calls, shell programs) and the mechanical differences between the different brands and models of computers. The primates are custom-tailored (ported) to each computer, and serve as a basis for building the higher levels of system software.

The primates themselves are a library of independent computer programs. They are not an integral unit. Thus, the modular design of the library of primates could conceivably allow the kernel to operate even if some of these computer programs were missing (as long as the missing modules were not called upon).

The kernel's operation of the computer system, in simple terms, is based on processing an ordered arrangement of primates, much as you would create words with ordered arrangements of the letters of the alphabet. The particular arrangements of primates required to perform tasks are provided by the instructions contained in the system calls in the operating system.

- Movement of data and computer programs into and out of the computer's memory and disk storage auxiliary memory.
- Assignment of disk space to new and/or expanded files, and returning disk space no longer occupied to a "free list."
- Track the flow of data in the computer.

Figure 2-3 provides a simplified diagram of the kernel's operating routine. The system calls that are queued to the kernel by the shell provide a source of instruction to the kernel, indicating which primates are to be selected and in what order they should be executed to make the computer system perform the user-commanded functions.

System Calls

When you operate a computer, you will find that there are a number of routine functions that you have to perform frequently, such as printing out a report on the printer. In order to make the computer as useful and efficient a tool as possible,

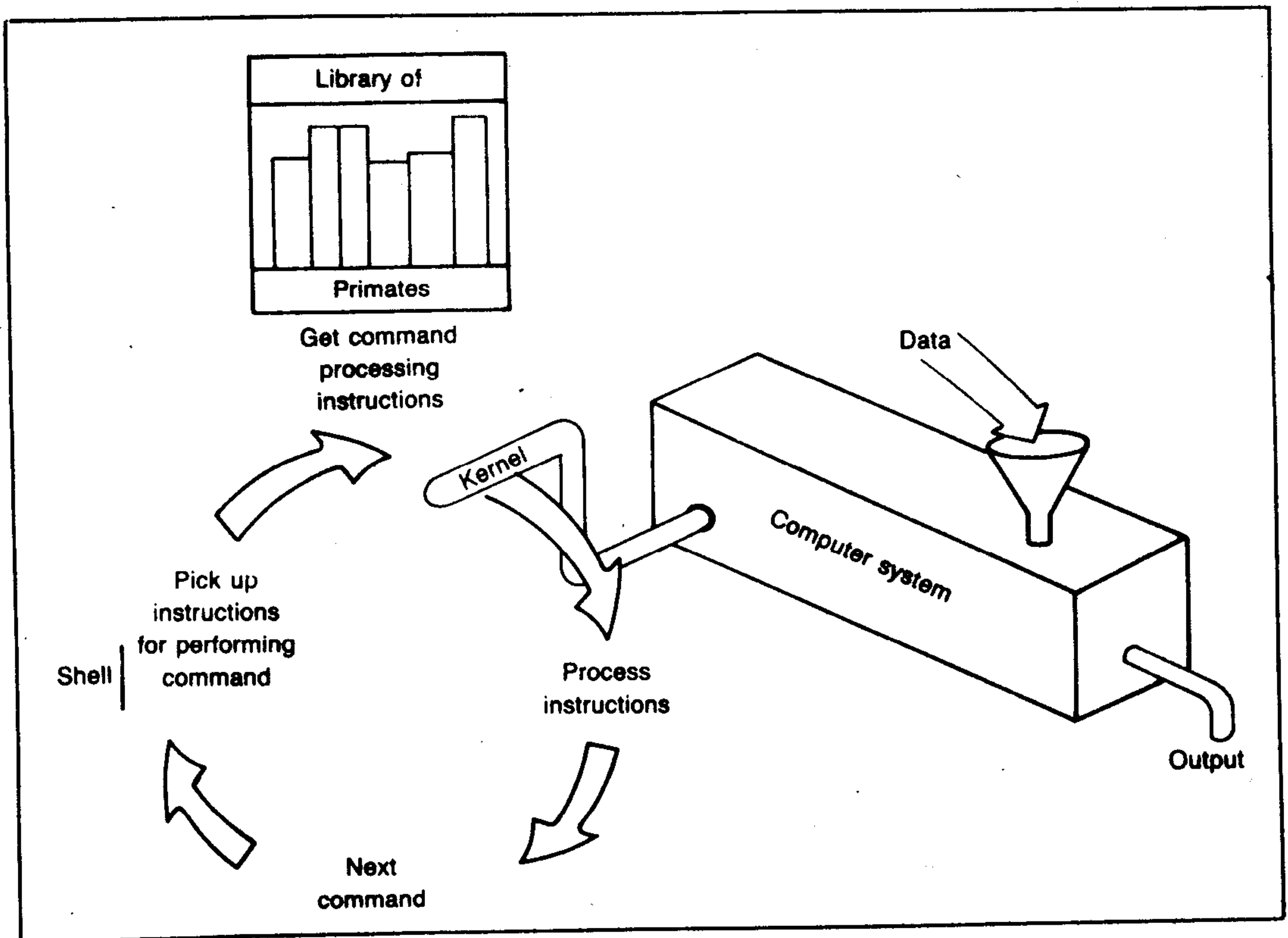


Fig. 2-3. The kernel's command processing routine.

computer programs have been written which, in essence, package the instructions for performing these routine functions. Thereafter, when you want to make the computer perform a routine function, you need only enter the name of the computer program, instead of a whole series of commands.

We have already discussed the lowest level of computer program that performs routine activities, the primates. The next level of service programs are called the *system calls*. They are described as being at a higher level because the instructions in these programs are designed to perform requested operating system functions by "calling upon" or "calling out" the required operating system activities that the primates are programmed to provide. Hence the origination of the term "system call."

System calls are similar to the primates in that each system call is designed to perform a simple function. They differ from the primates in that they provide a broader scope of service for operating the computer. System calls are, figuratively, the "words" that are built with the "letters" or primates.

System calls provide the basics for the operation of all Unix and user-developed utilities, as well as some applications computer programs installed in the computer.

Technical Note: System Calls

The system calls provide a standardized basis upon which the computer programs in the utilities may be written. That is, all utilities can be written based on the same basic set of system calls, which provide the essence of a common language. Utilities and application programs, therefore need not be designed exclusively for the operating specifications of any one computer.

It is often inferred that the system calls included in a Unix system are generally the same regardless of the computer in which such a system is installed. If you get technically involved with Unix systems, however, you will learn that there can be fairly substantial differences between system call libraries.

For example, the source code for the library of system calls in System V is approximately 40 percent greater than the comparable source code in System III. Also, some of the System III system calls have been deleted and/or modified.

As a user, these differences will not affect your learning to use any of the Unix systems. The differences between systems will mostly affect software development and operating system performance.

The Shell

In Fig. 2-1 the shell was shown to delineate the boundaries of the computer operating system. Thus, the shell is often described as the "protective shield" of the operating system, hence the origin of its name.

"Protective shield" figuratively represents the idea that the shell is the software in the operating system that interfaces with the environment, usually inferred to be the user. In other terms, the shell is the software in the Unix system that is responsible for initiating the processing of the commands that you enter at the terminal keyboard.

Figure 2-2 provided a simplified diagram of the routine that the shell follows when you enter a command at the terminal keyboard. It shows the shell:

- Looking up the command in the files.
- Determining what system calls are required to make the computer perform the requested command.
- Queuing the primates into the kernel as required by per the system calls.

The shell is referred to as a "command interpreter" because it "deciphers" what a command means. When we put this statement into lay terms, it means that the shell finds the commands (that you enter at the terminal keyboard) in the Unix file system. It does this by finding a matching set of characters, specifically the name of the command or the name of an application program.

After the shell finds the command, it is then responsible for making certain that the computer operating system processes the computer programs contained in the command. Hence, the shell is also described as acting like a supervisor for the computer operating system.

UNIX UTILITIES

Radiating along the right side of the shell in Fig. 2-1 is a ring labeled *utilities*. The utilities are an assortment of user-support computer programs (called commands) which you use to make the computer carry out frequently needed tasks. Utilities also provide a base for the design and operation of many application computer programs; application programs can use the utilities in the same way and for the same reasons as you would—to make the computer perform a task.

As it has been pointed out, Unix systems are rich in utilities. The bulk of a Unix software program is primarily due to the great number of utilities contained in the system. A small Unix system may contain as few as 40 Unix utilities, while large systems may contain 300 or more Unix utilities and an unlimited number of user-created utilities. Now, maybe, you can begin to understand the apprehension that users have for learning to use Unix.

Granted, a library of hundreds of utilities provides a broad range of functions for you to use to operate your computer, but it is also a great number of utilities with which to become acquainted, let alone memorize and learn to use. Fortunately, once you begin to learn what each of the utilities do, you will soon realize that you will probably never use the vast majority of them, or you will learn to use a select few and never bother to look at the rest until a need arises. At the end of the chapter is a list of the types of services provided by utilities. Most of you who read this text will have very little use for the utilities that provide software development services. Likewise, you will probably not be too interested in using the text processing utilities because they are not designed for heavy wood processing activity. Graphics also are easier to use in an application program. Communications utilities are only useful when you need to establish communications between two or more computers, and the games are not too interesting.

This leaves "General Systems Operations" and "File Handling" with which to deal. Of the hundred or so utilities in these two categories, the average user will never use more than two or three dozen of them. This brings us back to our original statement that you will only need to learn to use a small number of utilities in order to learn to use your Unix system quite proficiently.

If you are wondering why such a great number of utilities have been created, it is because the keynote of Unix system design is to "keep it simple." By convention, all utilities were designed to perform only very precise tasks. Simplicity forces the designer of a utility to create a specific functional attribute in each utility. In this way, each utility may be used in a greater number of applications, like using words to make sentences.

By design, utilities are supposed to be simple and precise. However, you will find that some of the utilities are not what anyone could consider to be simple and precise. Some utilities are quite verbose and complex. For example, the **ed** and **vi** editor utilities contain a considerable library of text editing functions in themselves. These two utilities deal only with text editing, but you can spend many hours to several days learning to use all of the capabilities designed into them.

The history of the development of the Unix system also sheds some light on this subject. During the early days of Unix, utilities were programmed to resolve specific needs and routine work requirements as they arose—such as providing a

Technical Note: Utilities

We have discussed thus far two levels of Unix system software, the primates and the system calls. A third level is the *utility*. Utilities interface with the system calls in a similar fashion as the system calls interface with primates. That is, the utilities contain instructions for assembling series of system calls, which in turn assemble the necessary series of primates, ultimately to make the computer system perform the commanded function.

computer program that would format inter-office memos. *There was no particular conceptual plan for the development of utilities.* Only later was any attempt made to show some system-like structural design and to limit the further uncontrolled accretion of utilities.

APPLICATION PROGRAMS

Radiating along the left side of the shell in Fig. 2-1 is a ring labeled *applications computer* programs. These programs are unique sets of specialty functions not provided by the library of utilities, and/or "user-friendly" containerizations of Unix utilities.

These programs may utilize the Unix utilities to perform their functions, or they may be programmed to address the system calls directly. From the standpoint of time to develop a system or efficiency of operation, there are advantages to each approach; however, a discussion of these points is beyond the scope of this book.

Some people consider application computer programs to be a part of a Unix system. Whether you do or not will not affect the course material contained in this text. That is, you are free to agree with or disregard their inclusion as you please. The logic behind considering application computer programs as a part of a Unix system is due to the similar way in which utilities and application computer programs operate within the computer.

SOFTWARE LEVELS

In this chapter we have been discussing levels of software in the Unix system. The levels principally refer to the use of one category of software by another and their applicable use in the operation of the Unix system. Figure 2-4 displays the different software levels, their interrelationships, and the purpose of each level of software.

The primates, as we saw, acted as the hardware interface to the Unix system. The primates are written in a machine-conversant language (the assembly programming language), which is designed to interface with the operation of the actual electronic circuitry inside the computer system. The primates form the basis for establishing communications with the machine itself. They are the first level of software.

The system calls are the second level of software. They are computer programs that contain instructions for operating the primates. That is, they are capable of

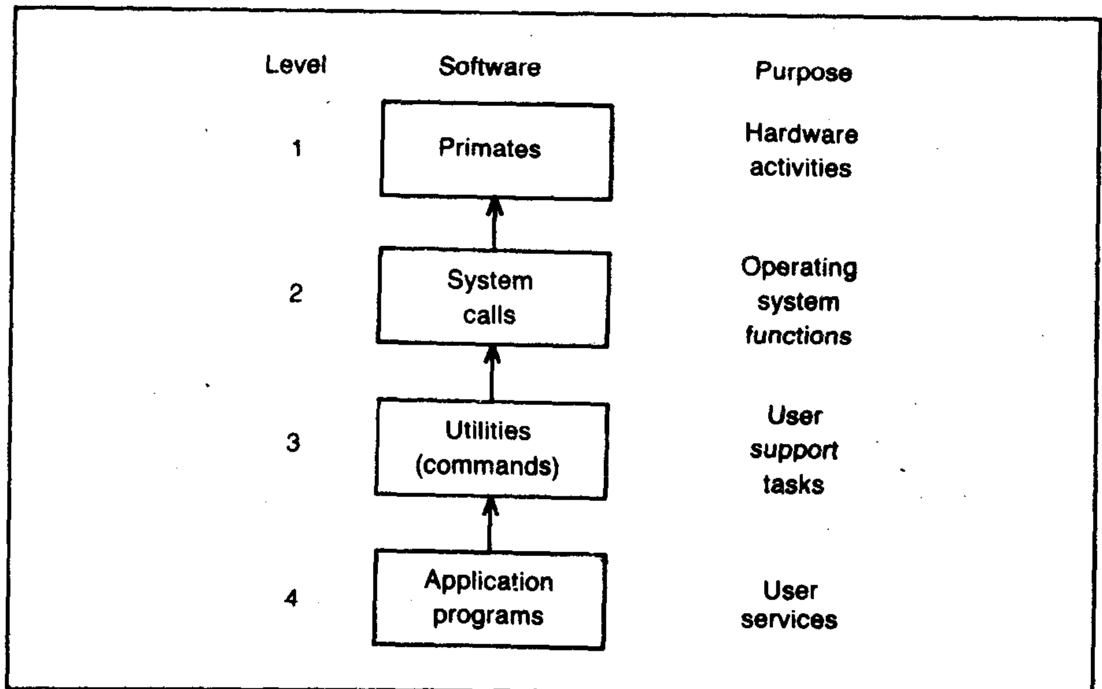


Fig. 2-4. Software levels.

conversing with the primates, just as the primates are capable of conversing with the machine.

It would still be difficult to write computer programs that had to rely upon the use of system calls, so a higher-order software based on the system calls has been constructed. This is the utility. The utility computer program thereby establishes a third level of software.

Categories of Unix Utility Services

This table lists the eight categories of utilities that Unix systems provide for you to use in operating your computer system.

A) GENERAL SYSTEM OPERATION

This group contains the administration and file management utilities. It also may be considered a catch-all for those utilities that provide system administration capabilities for controlling, monitoring, and managing the operation of the computer. These can be divided into two sub-categories:

File Management	Used to find, move, copy, display, link, rename, back up, and load files. Also used to make, change, and list the contents of directories in the Unix Filing system.

System Administration	Utilities to change the ownership and user access security of files, establish and/or change user passwords, find out which users are currently on the system and how much disk space is available, find out what jobs are in process, and, if necessary, terminate a job.

B) FILE HANDLING UTILITIES

The largest category in the Unix system library are those used for processing, reordering, and/or manipulating the data contained in user files. This group differs from file management utilities in that they operate on the contents of individual files, where file management utilities deal with files as units within the file system. These utilities can be used on data or text files, and fall into the following classes:

Comparison	Utilities to determine where two files differ, what text is common, what text is unique, and how many words, characters, or lines are found within.
Tabular Processing	Utilities to delete columns from a file, combine the columns in two or more files, or combine corresponding lines.
Search/Modify	Utilities that search for a specific pattern of words or characters in a file and modify the matched pattern according to user—specified instructions.
Split/Combine	Concatenation utility used to display the contents of a file or merge two files.
Sorting/Ordering	Utilities to order the contents of a file according to user-specified criteria, or eliminate duplicate lines.
Compression/Encryption	Utilities that compress the contents of a file (so that it will consume less disk space), and/or encode the contents of a file according to a user-supplied encryption key.

C) TEXT PROCESSING

The Unix system is richly endowed with utilities that can be used to create text files. These utilities are similar to word processor application programs; in fact, many of the word processors that operate on Unix systems utilize the available text processing utilities. These utilities even include phototypesetting capability which, due to the wide number of

Unix system installations, is being supported by phototypesetting timesharing service companies. Utilities in this category include:

Text Editors	Single-line and full-screen text editor programs.
Text Formatters	Typesetting programs for both printers and phototypesetters.

D) SOFTWARE DEVELOPMENT UTILITIES

The Unix library supports a variety of computer languages and software development tools. The Unix system is teamed with the C programming language, but the system will support a number of other popular high-level languages such as FORTRAN, COBOL, BASIC, Pascal, etc. In addition, the Unix system also contains a number of utilities that assist in the software development process and the management of software development projects.

E) GRAPHICS AND STATISTICS FEATURES

Some graphics utilities are available on Unix systems for graphing numerical information. A library of statistical utilities can be used to perform arithmetic calculations and statistical analysis, and to prepare numerical data for graphic representation.

F) COMMUNICATIONS UTILITIES

This category of Unix utilities includes those for communicating between users on a system, as well as between users on different systems or different computers. These utilities can be used to transfer simple messages or complete files, as well as to provide for remote job entry. Communications utilities support office automation and provide desirable, economical alternatives to mail services.

G) REFERENCE UTILITIES

Reference utilities are included in many Unix systems. These generally provide spelling checker programs that can operate on other files, copies of the various Unix user manuals, and more.

H) GAME UTILITIES

Some Unix systems are provided with, or at least support, a variety of games such as Adventure, Wumpus, chess, checkers, backgammon, cribbage, etc.

The utilities have been, by design, kept simple so that users could write programs based on the functions that are designed or programmed into them. That is, applications are developed based on the functions provided in the utilities. Thus, the application program becomes a fourth level of software.

Chapter 3

The Unix File System

In this chapter we will discuss the marriage of the Unix system (software) with the computer system (hardware). This discussion will provide the basic fundamentals for explaining:

What a file is.
The purpose of the file.
The relationship between files and the operation of the computer system.
Different uses of files in the Unix system: directories, data files, special files, program files.
The hierarchically structured Unix filing system.
Role of the directory in forming the hierarchical file structure.
File naming.
File security.
File linking (alias file names).

This chapter should help to resolve many mysteries surrounding the actual operation of the computer and Unix.

AUXILIARY MEMORY

A discussion of the Unix file system begins with a study of the computer's auxiliary memory or *mass storage devices*. This will explain how the software is stored in a computer and how the computer user gets the software into the computer for processing—without getting all the computer programs and data mixed up.

Examples of mass storage devices are hard disks, floppy disks, drums, bubble

memory, magnetic tape, and other devices on which computerized information may be recorded or stored until it is needed for processing, at which time it will be transferred to the computer's main memory (called *random access memory* or simply *RAM*). Auxiliary memory should not be confused with the computer's RAM or main memory, which is the part of the computer system that holds computer programs and data during the actual processing of the program or data.

Most of the microcomputers currently manufactured use the hard and floppy disk for auxiliary memory. Therefore, we will direct our discussion to these types of devices for explaining how computer programs and data are recorded, stored, and later located in the microcomputer's auxiliary memory.

Information is electronically recorded on the surface of a disk just like music is recorded on the tape on a cassette with your home recorder. The only physical difference is that computerized information is recorded along concentric circles of magnetic tracks on the surface of the disk, which are laid out like the grooves on a record. You can not see these tracks on a disk, but they are every bit as real as the physical grooves on a record.

To facilitate locating specific information on the disk, the magnetic tracks on the disk are segregated into *blocks*. Figure 3-1 is an example of the layout of the blocks and tracks on a disk. (The diagram is not drawn to an actual scale, because a disk contains thousands of blocks)

The blocks serve in a similar function to that of the tape counter on your cassette recorder, which helps you to locate different songs on the cassette. Each block on the disk, regardless of the number of tracks and blocks on the disk itself, is given a unique *address* called the *i-number* or *i-node*. It is this block address that is the key to the interface between the software and the hardware. The address is used by the operating system to locate computer programs and files that are stored in the blocks on the disk. Unix utilities, the kernel, the shell, the library of system calls, all

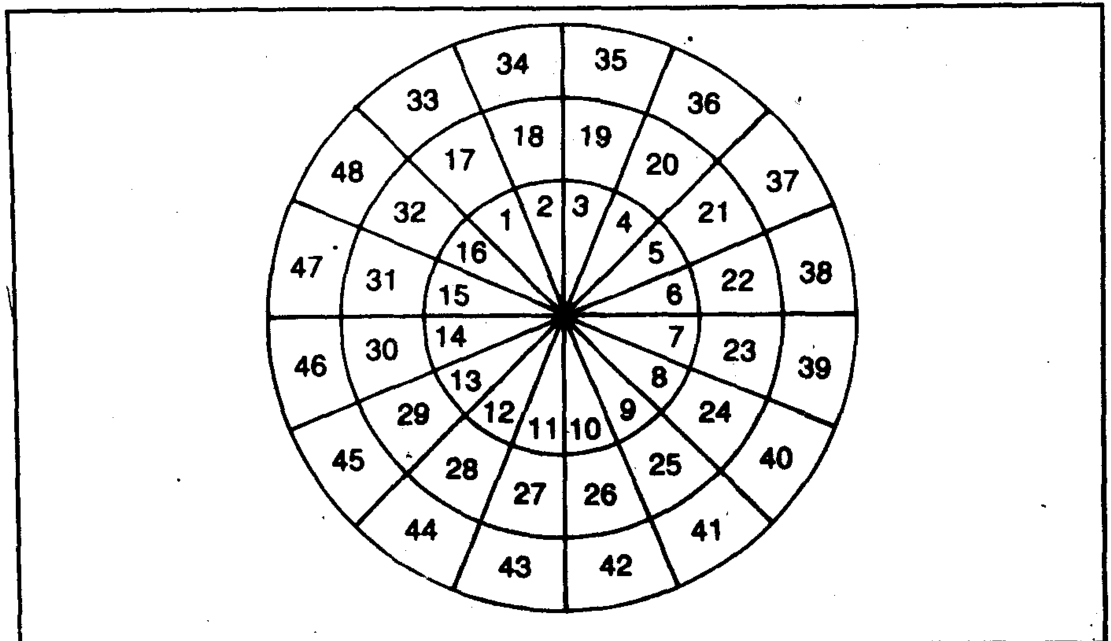


Fig. 3-1. Example of data storage blocks on a disk. (Not to scale.)

user-generated utilities, data, computer programs, and all purchased software applications stored on the computer's disk are uniquely identifiable and locatable by the block address so that they may be retrieved when they are needed.

Technical Note: Disk Configurations

The number of tracks on a disk will vary according to the design of the disk drive on which the disk is used. For example, many microcomputers, currently using Unix systems employ the 5 1 4-inch disk drive, with 96 tracks on the disk, but other disk drive sizes and track configurations are available.

The maximum storage capacity of each block at each address is fixed by the design of the Unix system installed in a computer system, not by the disk drive. Most microcomputer systems currently have Unix systems designed to store up to 512 bytes of information in each block. A byte is the equivalent of a single character, such as a letter, digit, etc. This means that each block can contain up to 512 characters.

The storage capacity of a block, however, is not a fixed standard, Unix or otherwise. Some Unix systems have been modified by various suppliers to use a block size capable of storing up to 1024 bytes or characters, and sometimes more. For example, Unix System V uses the 1024-byte block as its standard format, but will also support a 512-byte block as well. Berkeley versions of Unix use a 4096-byt 'block size.

A large block size can enable a computer to run faster, because more data is contained in each block. The computer can read data in a block much faster than it can physically move the read/record head to a particular block location on the disk; much of the time consumed by a computer to process a program or command is the time the disk drive needs to move the heads to the block where the information is stored. A larger block can hold more data; therefore, the disk drive does not have to move to as many different block locations.

The drawback to large block sizes is the high "overhead" that has to be paid in wasted disk space. For example, if a computer program is 1025 bytes long, it will consume three blocks (1536 bytes) on a system formatted for 512-byte blocks. In the third block, 511 bytes of disk storage space are wasted. (You can not mix data from two programs in a single block.)

In comparison to a system that is formatted to use the 1024-byte block, the same 1025-byte computer program will require only two blocks (2048 bytes), but now 1023 bytes of disk storage are wasted.

Wasted disk storage space was once of more concern, but the constant reduction in the cost of disk storage (e.g., $7000 for 140 megabytes has been advertised from Dragon Industries for the IBM PC) has made the overhead cost versus improved processing less significant a problem.

To calculate the number of blocks in auxiliary memory, divide the known capacity of the device by the system block size. For example, a 40-megabyte disk (40,000,000 characters) divided by 512 bytes per block equals 78,125 blocks.

The sidebar in this section presents a fair amount of technical information. It is probably not essential to your learning to use Unix, but it will help you to understand better how the computer system works and thereby how the Unix system interacts with the computer system. More important, it may help you to understand some of the operating principles of the computer and Unix, so that you will be able to choose a system with better operating features.

THE FILE

A file is merely a computerese term for related information recorded in one or more blocks on the computer's disk. A file may contain any information at all, from the text for a novel to a computer program for statistical analysis. The contents may be written in computer programming languages like assembly, BASIC, FORTRAN, Pascal, etc., or may simply contain data in ASCII code (the standard code used by microcomputers for representing the different characters on the terminal keyboard).

While the operating system treats all files as simply strings of characters (any continuous series of characters), the purpose of the contents of a file does pose some distinct differences between files. The following categorization of files, based on the purpose of their contents, may help to clarify this point.

Ordinary Files

Text and Data Files. The information contained in these files represent printable characters, and are called text or data files. Word processing, statistical data, accounting data, etc., make up this class of files.

Executable Files. Files in this class are similar to data or text files in that these files also contain strings of characters. The difference is that the characters contained in these files are computer programs, written in programming languages such as C, FORTRAN, BASIC, Pascal, etc. Executable files are also known as *commands*. (Executable files and commands will be discussed in detail in Chapter 5.)

System Files. The system utilities and the files which comprise the Unix computer operating system itself form this class of files. These files are also executable, but the distinction made between these files and the class called "executable files" is that the system manages these files and isolates them from users' files on the disk.

Directory Files

These files are unique in that only statistical information about file locations, sizes, etc., stored on the computer's disk may be included in them. Directory files have a strictly prescribed internal format for recording this data, which the operating system is responsible for administering and formatting.

Special Device Files

All input and output devices, such as printers, disk drives, etc., are made to look like they are files. When you wish to print out the contents of a file, you direct the output to a device file. Device files contain the specifications for interfacing with any of the named devices. This is an advanced subject and no immediate value will be gained by pursuing the subject in further detail here.

File Contents

The Unix system places no constraints on the contents of a file. The design of the Unix system treats all files simply as information or strings of characters. It is assumed that you will know what is contained in a particular file and, therefore, you will be able to instruct the operating system on what to do with the contents of your files.

Files that have to occupy more than one block are "daisy chained" together (Fig. 3-2). As characters are added to a file, the operating system allows them to fill the block to capacity and then assigns another block for the continued recording of data on the disk. In the Unix system it is not necessary that the blocks containing a single file reside in adjacent blocks on the disk. The management of the chain and the recording of the locations of the files stored on a disk is the responsibility of the operating system. Provisions have been made in the system for logging the locations of the block or series of blocks in which each file is contained.

If you are operating your computer with a word processor or other application computer program, the program takes care of the details of telling the operating system what to do with a file. Likewise, the Unix operating system is programmed to take care of its own files.

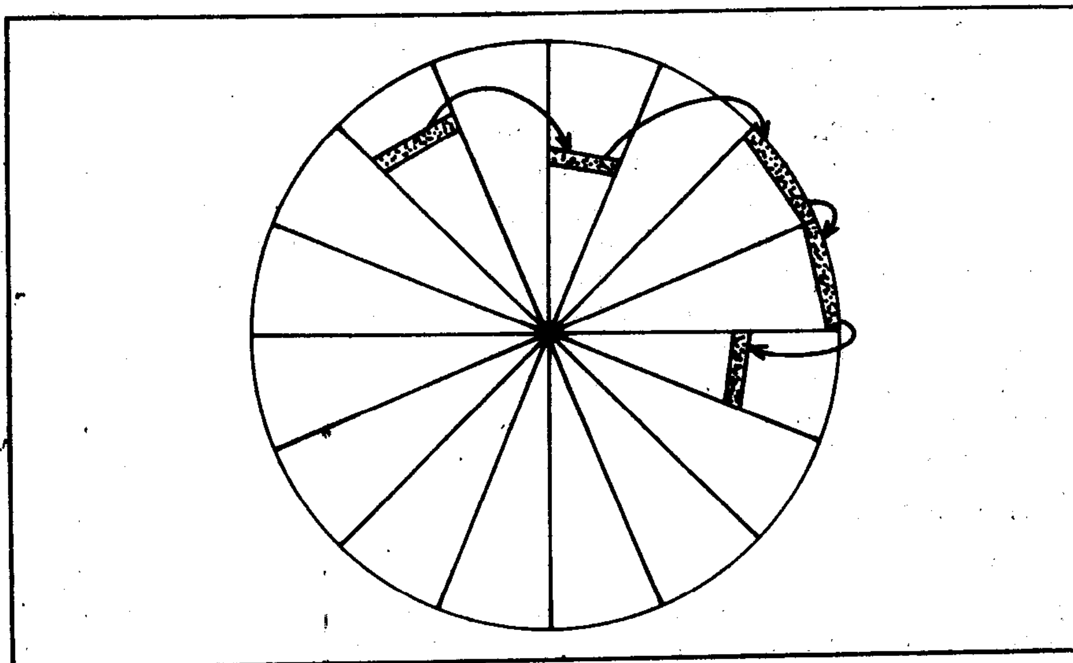


Fig. 3-2. "Daisy-chained" blocks for storage of a large file.

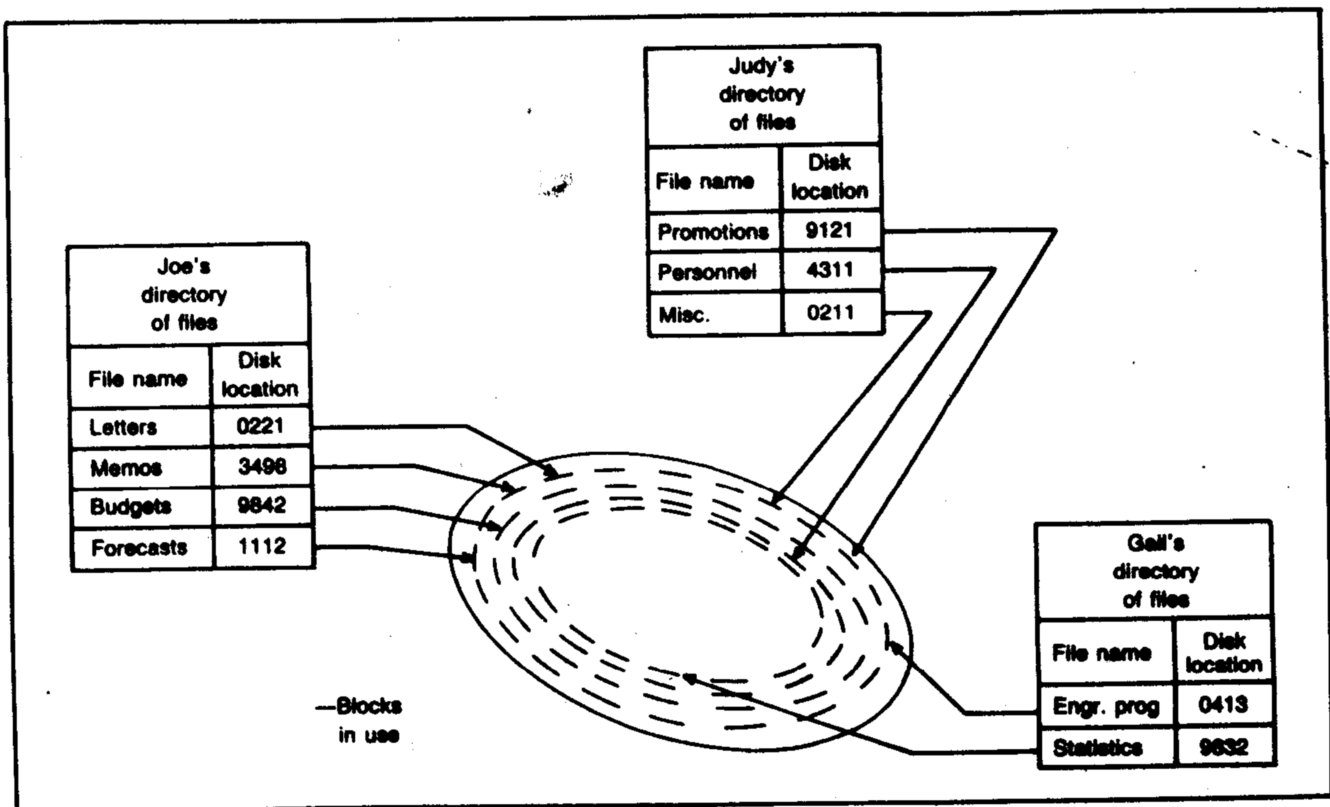


Fig. 3-3. Directory of file locations on a disk.

The theoretical limit of the size of a file in System III, for example, is 2 billion characters, which is well beyond the maximum physical storage capacity of all but very, very large computer systems. The actual details of the management of the daisy chaining of blocks is beyond the technical scope of this text and the information, once again, is not explicitly essential for you to know in order to learn to use Unix.

File Names

Files are located on the disk by the block location address at which the operating system has recorded them. Since it would be difficult to memorize numeric file location addresses for every file recorded on the computer's disk, the Unix operating system allows you to assign names to your files. The operating system will maintain a directory of the cross references between the file names and their numeric block location addresses (Fig. 3-3).

Unix allows you to name a file just about anything that you want to name it. File names can be descriptive or in a private code. The rules for naming a file are as follows:

- Any sequence of characters up to 14 characters long is legal.
- Upper- and lowercase letters are allowed and are unique. For example, these file names are uniquely identifiable:

```
Fort  FORT  ForT  fort  forT
```

- Avoid using metacharacters other than the period, dash, comma, and the underline. (Metacharacters are the symbols on your terminal keyboard that Unix uses like commands—Chapter 5.)
- Spaces are not allowed. If you want to show some separation between words in a file name, use the period, dash, comman and/or underline to separate words in a file name. For example

  text.draft alpha-company grocery_bills

  are legitimate file names.

THE HIERARCHICAL FILE STRUCTURE

A hierarchical structure is a very common way to organize and display information. The most common examples of hierarchical structures are company and corporate management organization charts. The objective of the hierarchical structure is to show the commonalities among the items on the chart.

For example, the objective of the corporate organization chart is to show the major management organizations within the corporation, such as accounting, manufacturing, etc., and the submanagement organization structure within these organizations, such as the department manager, staff, function heads, supervisors, etc.

The objective of the hierarchical file system format in the Unix environment, likewise, is to organize your files into groups and subgroups, as you can see in Fig. 3-4. The hierarchical structure enables you to store common files together

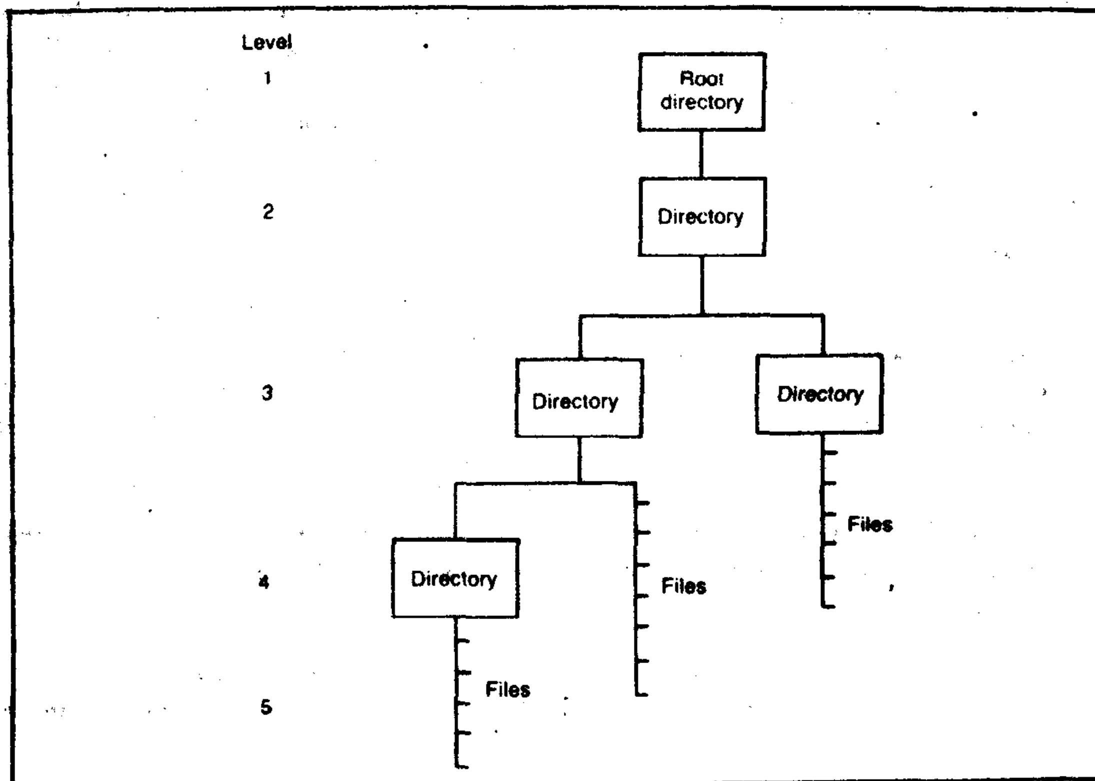


Fig. 3-4. Files do not have to be on the same level of the hierarchical structure.

logically, even though their physical locations on the disk may not be adjacent. This feature is particularly important for conveniently adding new files to the filing structure at a later date. With some operating systems you have to continually restructure your file locations on the disk.

Building the Hierarchical Structure

If you were to open your computer, you would not be able to see a hierarchical structure of files, any more than you are likely to find the corporate executives' offices situated according to their positions on the organization chart. I also do not know of any utilities that graphically display the hierarchical structure of the files on the terminal monitor.

The hierarchical structure is itself only a conceptual presentation, a kind of "visual aid" serving as a table of contents for the directories and files in the system. You can manually create a graphic representation of your hierarchical structure so that you can locate the names of the directories and files in the system, just as you would use a table of contents to find chapters and subchapters in a book. However, the hierarchical structure itself is only an illusionary device created to demonstrate the interrelationships among files.

The building of the hierarchical file structure begins when you install the Unix system on your computer. A master index of the system files and directories in the Unix system is created for the system as part of the installation routine provided by the computer manufacturer or software vendor. Thereafter, it is your responsibility to add any additional directories to provide for the further growth of your file system.

Building the hierarchical structure begins when a user is registered on the system. Part of the administrative routine when adding a user to the system is to create a *home directory* for the user. Figure 3-5 presents an example of a user's home directory in the hierarchical structure.

The arrows in Fig. 3-5 indicate where this new user would add directories and files to expand his or her file system. The number of subordinate directories that may be added to a home directory is not limited by the Unix system. You may create directories and files until you run out of physical disk space.

Purpose of the Structure

Many computer operating systems, particularly those on smaller microcomputers, maintain a simple alphabetic and or numeric index (referred to as an *alphanumeric index*) of the files on the computer disk. Most of the operating systems installed on small microcomputers do not have to contend with very large numbers of files stored on their disks; a simple straight or serial listing of the files is an adequate method for indexing file locations. Serial listings are simple to administer, and the operating system software to do the job is easier to design.

The large computer systems on which Unix was originally designed to operate often contain thousands of files belonging to a number of users. To locate a file in a serial listing of files, the operating system would have to start at some point in the listing and check each file on the list until it found the requested file name.

78/06.54
B3

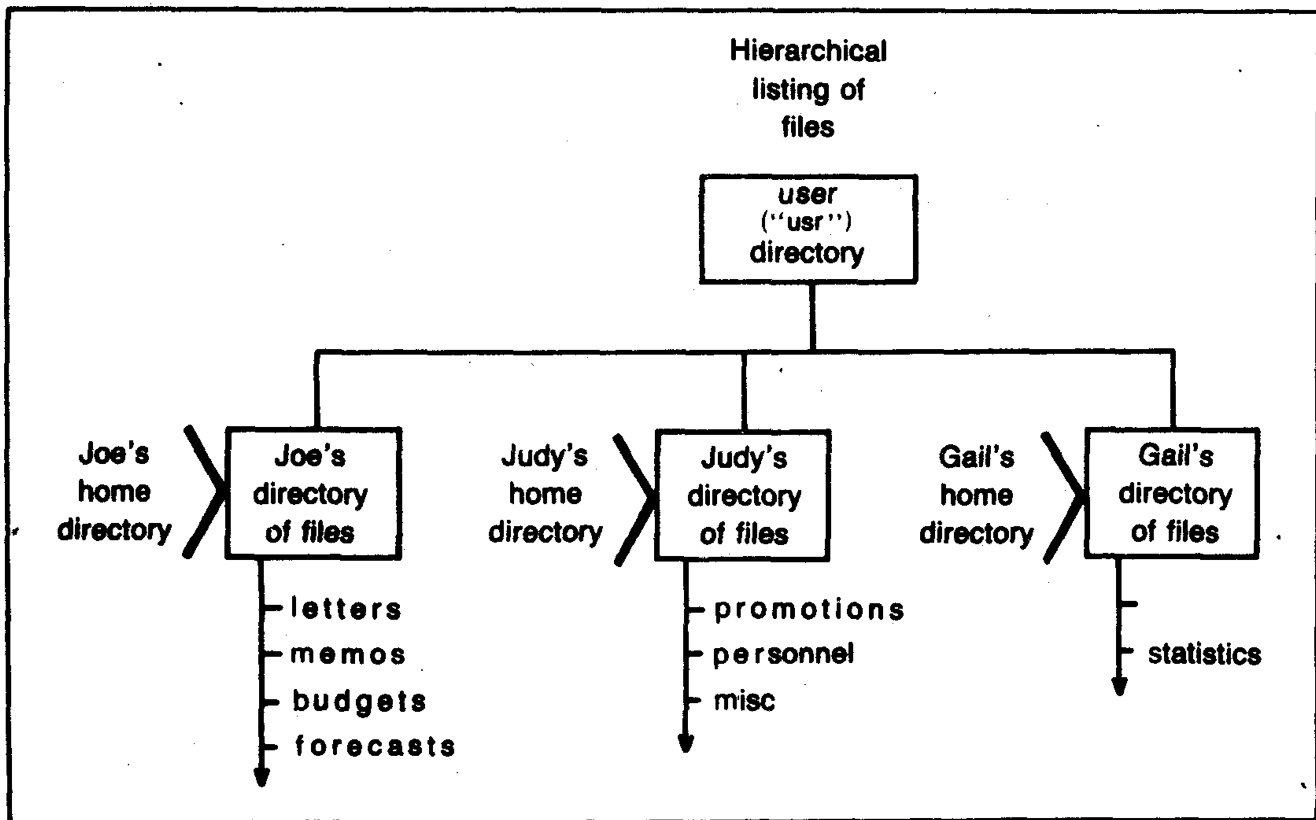


Fig. 3-5. Users' "home" directories.

Operating even at the super speeds that modern computers do, it could take an annoyingly long time to locate a file with this method.

To expedite the file management processes, the designers of the Unix system incorporated the hierarchical format or structure for indexing files. Locating a file in a hierarchical file structure is faster because the operating system does not search for a file in a part of the hierarchical structure which it knows can not contain a specified file.

For example, if Judy requests the operating system to get her file **promotions**, the operating system knows that it will not find this file in Joe's directory of files. Figure 3-6 (also Fig. 3-8) shows how the operating system would search for the file **promotions** in a serial and in an hierarchical listing of files.

The Hierarchical Structure in Operation

From an operational point of view, the hierarchical structure is a network of directories. The directories (system- or user-created) can best be described as sub-indexes or junction boxes, which are used for routing the operating system through the hierarchical file structure when searching for a desired directory or file.

Figures 3-7 and 3-8 may help you to understand how this works. Figure 3-7 is a partial graphic representation of the more comprehensive listing of actual files (Fig. 3-8) in one part of my computer's hierarchical file structure. There is no actual, physical pattern in which the files are stored on the computer's disk. For all

intents and purposes, this listing of files and directories is a continuous listing. There is no first, second, or third level.

It is only by virtue of the fact that each directory contains the addresses of the directories and files of the next subordinate level of the hierarchical structure that we can create the illusion of a structure. That is, a directory contains the directions to the subordinate directories (subdirectories) and files at each level of the structure. Figure 3-9 demonstrates graphically how the operating system would follow the disk address locations contained in each of the directories, to find a file several levels "down" the structure.

You could direct the computer to step you through the file structure a level at a time but, as we will discuss, the Unix system provides a much more convenient means for accessing directories and files in the hierarchical file structure. These are called *pathnames.*

Directories in the File Structure

The cross-reference between file names and their numeric locations on the computer disk is maintained in a file called a *directory file,* or simply a *directory.* As it has already been pointed out, the directory (like all software and data recorded on the computer disk) is itself only a file. It contains the systems information related to the files and directories subordinate to it.

Many users, new to Unix, have a difficult time accepting and/or remembering that a directory is only a file with a special purpose, just like a command is a file

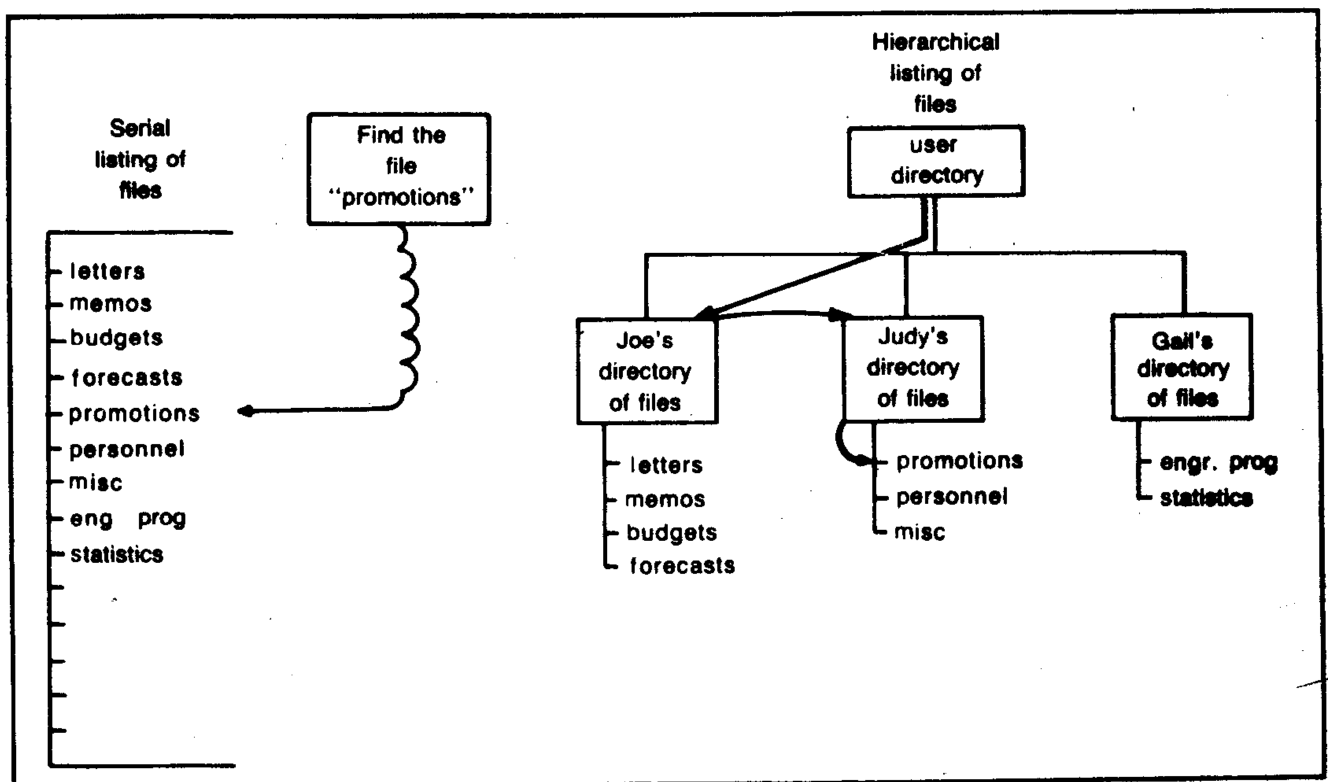


Fig. 3-6. Serial vs. hierarchical file searching.

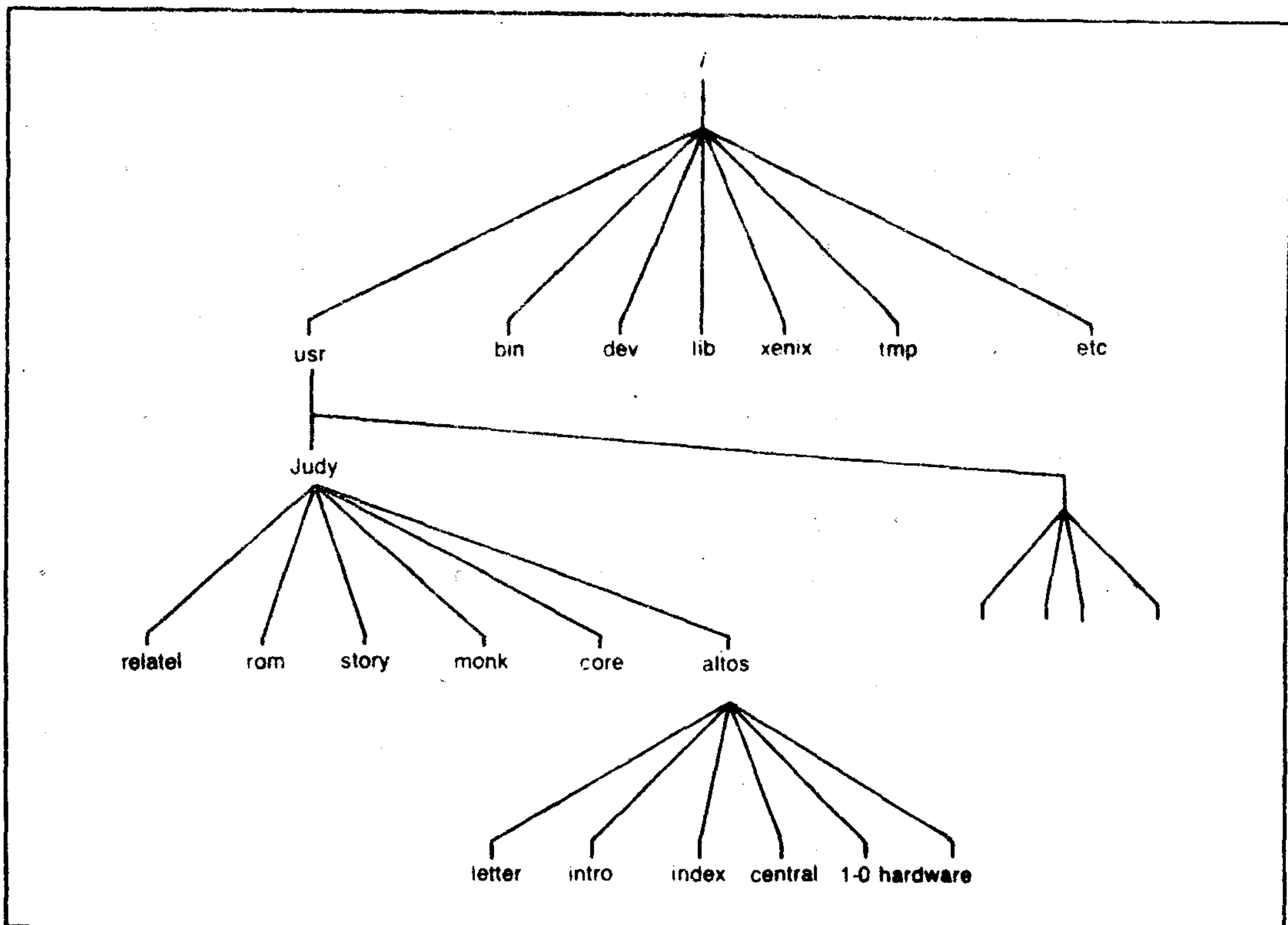


Fig. 3-7. Example of the structure of files and directories.

with a special purpose. Directories are treated like any other file in the system. They have a block location address, and you can name your directories just as you do any of your other files. The rules for naming a directory are the same as for naming a file.

Some users prescribe beginning directory names with an uppercase letter so that it is easier to differentiate between files and directories when the computer displays a listing of them. This is not necessary, because the operating system provides a command which will designate the difference between the two when they are displayed on your terminal monitor. (Read the explanation on the use of the **ls -l** command in Chapter 10.)

The only restrictions on directories is the type of information that may be contained in them. Directories, in addition to the block location of its files, contain the following information about the files for which they are responsible for maintaining records:

The size of each file in the directory.
The various names by which each file is known.
The name of the owner of each file.
Who may have permission to access each file.

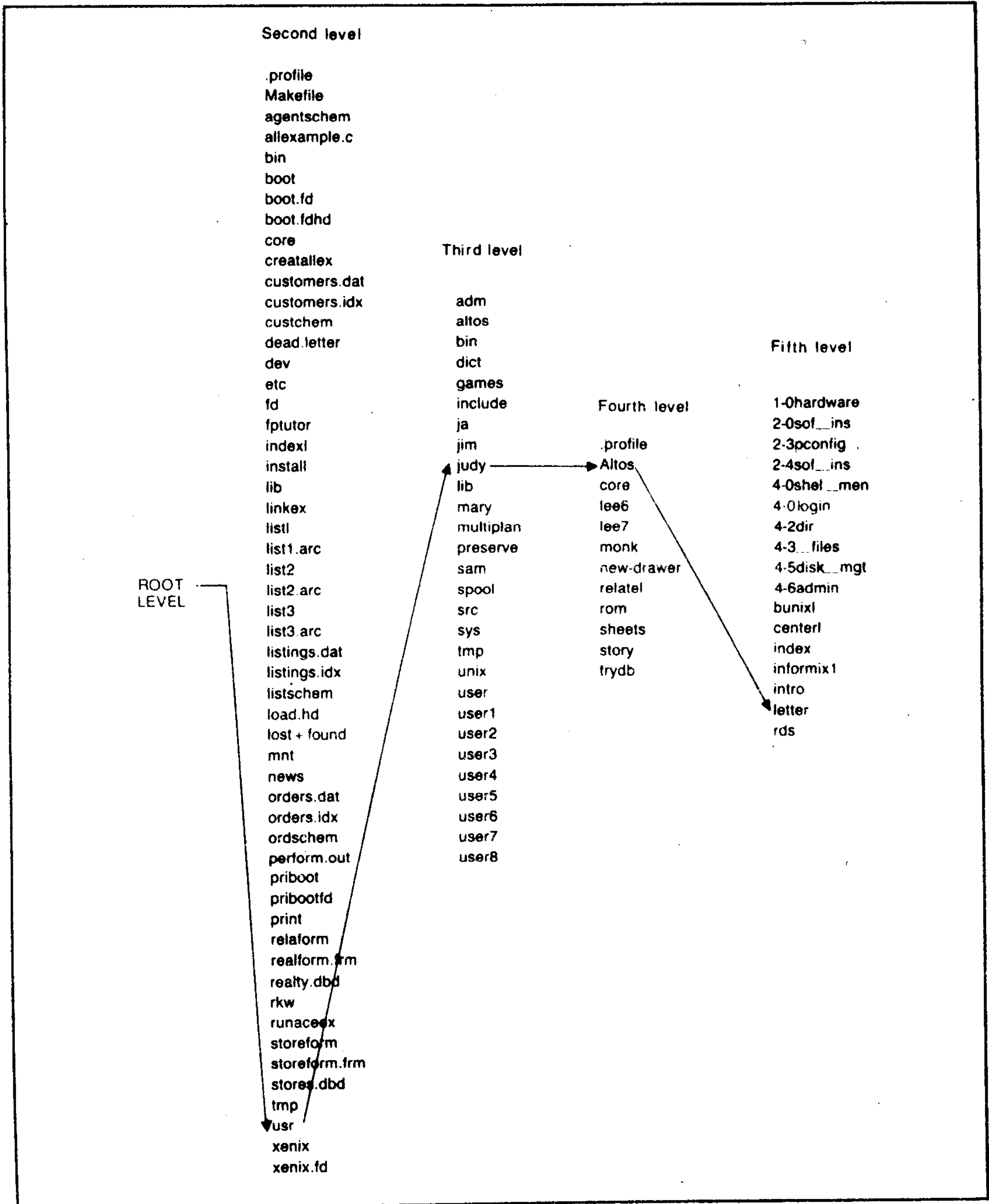


Fig. 3-8. The path to the file /usr/judy/Altos/letter.

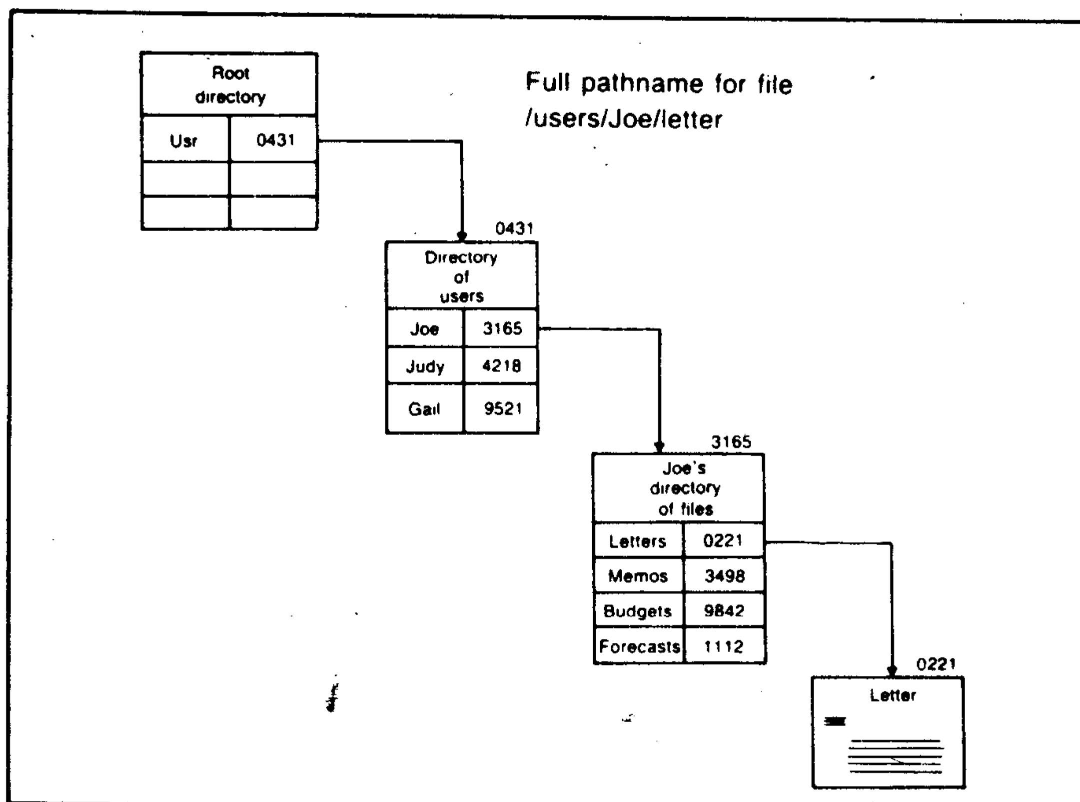


Fig. 3-9. Locating a file.

Whether a file is a file or a subdirectory.
The date that each file was created.
When each file was last accessed.

Do not be concerned with the details for the administration of the directory file; the operating system is programmed to take care of all of these technical details for you.

Directories may contain directories and or files. There is no Unix system limit to the number of each that may be included in a single directory (up to the physical limits of the disk storage memory).

Sub-directories in a directory may further contain any mix of and any number of directories and files. There is also no system constraint on the number of levels that may be added to the hierarchical structure.

Typical Top-Level Directories

The top levels of the hierarchical file structure, by convention, are usually structured the same and contain the following directories (Fig. 3-10):

First level:

root The top level in all Unix system file structures.

Second level:

usr Users' home directories are usually contained in this directory.
bin This contains the library of Unix utility programs.

dev This contains a library of operating specifications for many different brands of printers, terminals, etc., that you may have connected to your computer. You "configure" your computer to operate these devices by indicating to the operating system which device is attached to your computer. The operating system will create a file containing this information.

etc Contains system files such as **passwd** (password), which contains a list of all of users' passwords.

tmp Used to store programs temporarily.

It is also possible to create a directory level above the system *root*. An example of this is the installation of the network software on the Altos computer. The networking program puts the directory for each computer in the network at the second level of the hierarchical structure. Therefore each computer can access the directories and files in another computer just as it would access a directory or file in its own hierarchical structure.

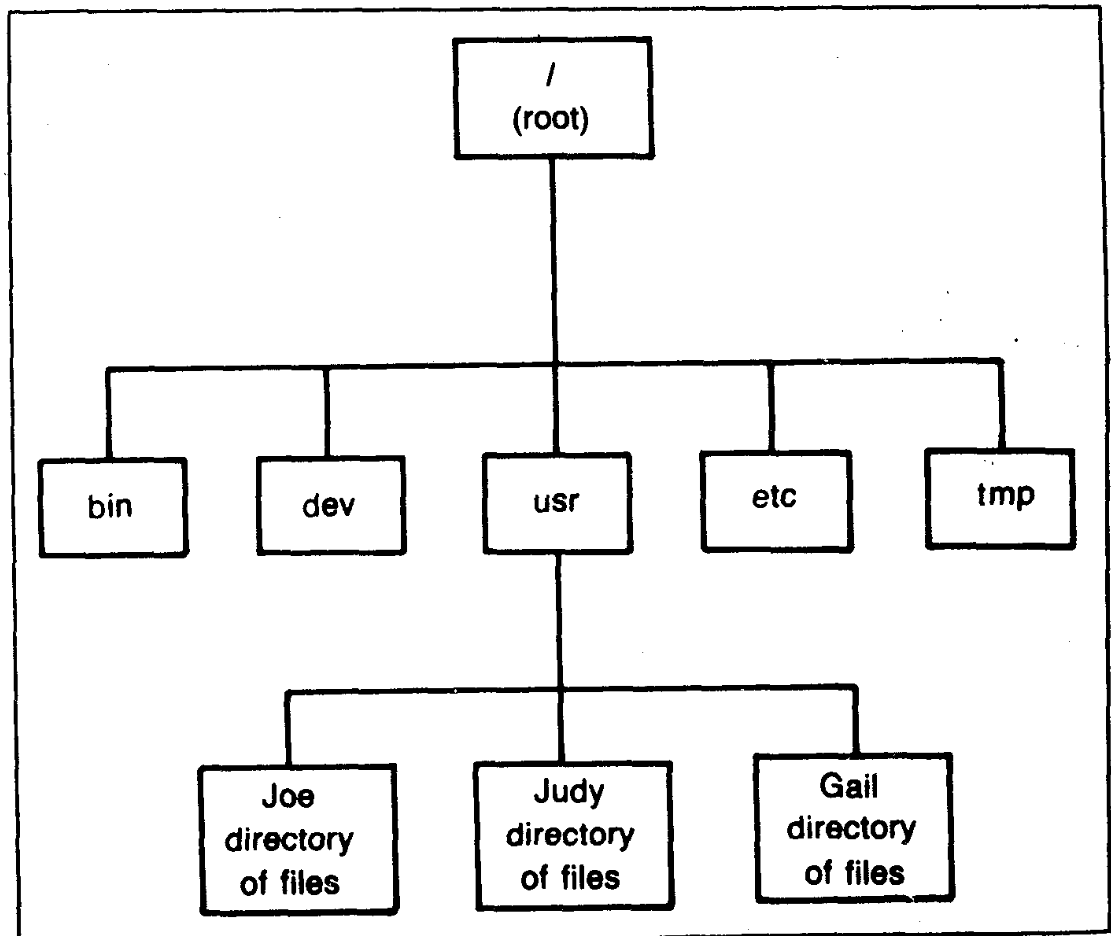


Fig. 3-10. Conventional top-level Unix hierarchical file structure.

PATHNAMES

When using a simple, straight listing of file names, the way you would tell the operating system to find a file on the disk would be to enter the name of the file at the terminal keyboard. Since all file names are effectively on the first level, the entry is simply the name of the file, as we demonstrated in Fig. 3-6.

The Unix system, however, employs hierarchical filing structure, and file names are not on one level or even necessarily on the same level of the structure. Therefore you have to provide the operating system a little more information in order to direct it through the maze of directories in the hierarchical file structure. You have to provide the *pathname* of the directories that lead to the desired file, as we demonstrated in Figure 3-9.

A pathname is exactly what it might seem to describe. It is a road map through the various junctions of the hierarchical structure. A pathname consists of the names of all the directories on a path from the top of the hierarchical structure, the root, to the file you wish to access, ending with the file name itself.

By convention, pathnames begin with a slash (/), and the names in the pathname are separated by slashes. For example, the pathname for Joe's file **letter** would be:

```
/usr/joe/letter
```

The pathname makes each file name a unique file name. That is, the complete pathname is itself a unique file name. Therefore, the same names may be repeated in a pathname and/or appear in the pathnames of different users, since each user's home directory makes anything in his whole file structure a unique path from the root.

For example, the name **activity** could be both the name of a directory and a file in the pathname:

```
/usr/user1/activity/activity
```

or a like name could be used by two different users:

```
/usr/user1/letters
/usr/user2/letters
```

or a user may use the same name under different directories:

```
/usr/user1/activity1/games
/usr/user1/activity2/games
```

All of the above are unique file names because the pathname in total is unique to the Unix hierarchical file structure. This is another major benefit of the hierarchical structure over a serial listing of files in which *every* file name must be unique.

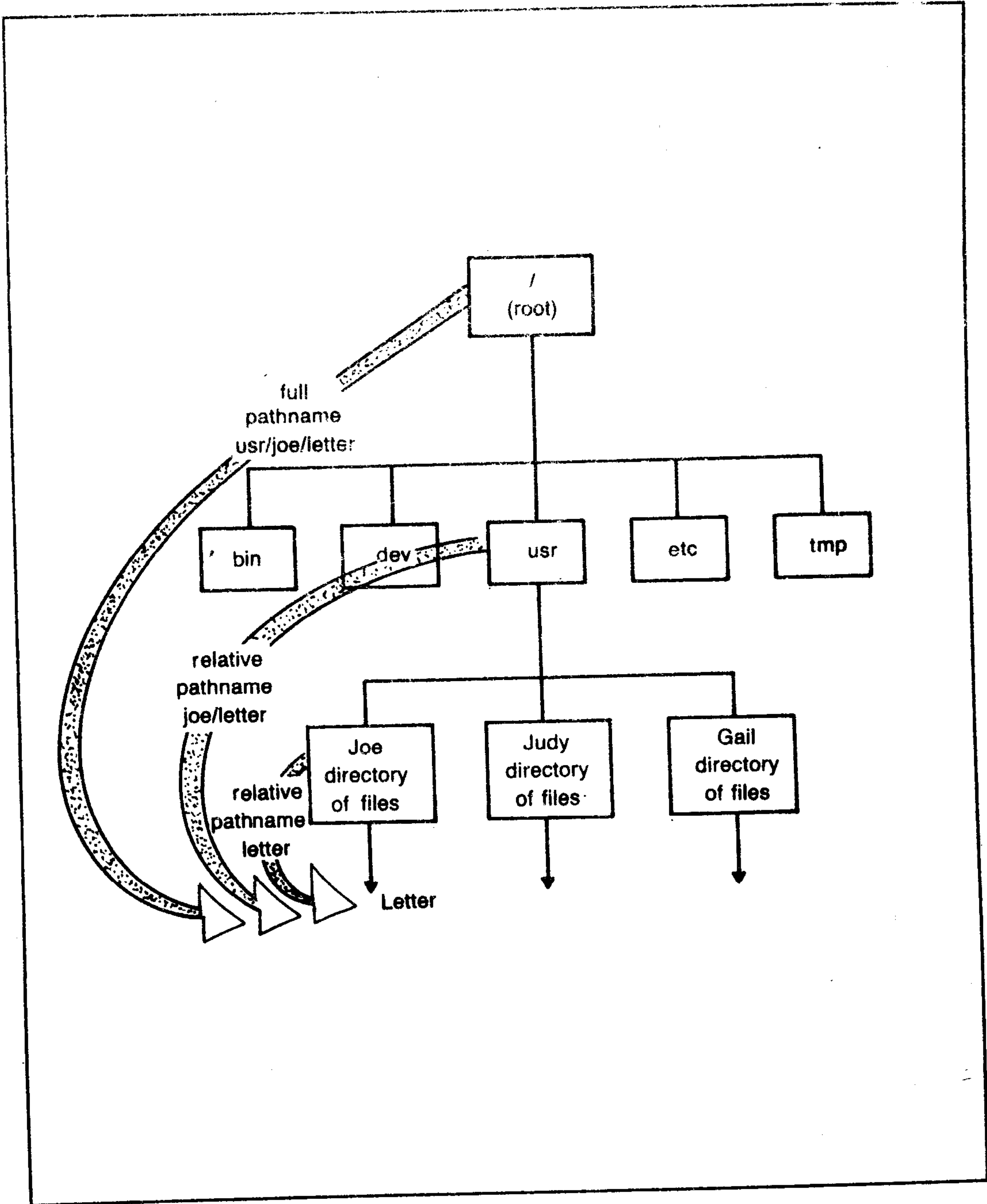


Fig. 3-11. Full and relative pathnames to a file.

As efficient as the hierarchical filing system is for filing and locating files, it would be tedious to have to enter a full pathname every time you wanted to access a file. To make accessing files easier, the Unix system was designed in such a fashion that you can "position yourself" at any directory along any path in the hierarchical structure, and then list only the names of the remaining directories in the path from the current directory down to the file. The pathname of a file is relative to your current location in the hierarchical structure. This is why it is called a *relative pathname*, i.e., it is relative to your location (Fig. 3-11).

For example, if you are at the directory **usr**, to get to Joe's file **letter**, you would need only enter the relative pathname:

joe/letter

If you were positioned at the directory **joe**, which contains the file **letter**, you need only enter the file name **letter**, for example:

letter

Earlier I stated that pathnames begin with a /. The / in the pathname indicates "begin at the root directory." If you include a / at the beginning of a pathname the operating system will interpret the / as root. Therefore you do not include a / at the beginning of a relative pathname.

The Unix command to reposition yourself at a lower level or to another directory is **cd**, for *change directory*. The use of this command will be used and explained in more detail in the lab exercises.

FILE SECURITY

File security for a personal computer is fairly easy to maintain because it supports a single user at a time. Unix systems operating in a multi-user environment have many users on line simultaneously. It is therefore important to provide each user with the ability to secure the files they have in process and stored in the system. It is also important to keep unauthorized users from gaining access to the vital parts of the Unix system, where they may inadvertently delete or modify the operating system itself.

Unix system users may set security for each of their files stored in the system. They can elect to allow or disallow a specific group of users, or anyone having access to the computer, to have access to any or none of their files or directories.

Each user may set selectively the security for each of their files to allow or disallow users to be able to read a file, modify (write to) a file, and/or run (execute) the program in the file.

Permission for one or more individuals to have access to one or more of the designated functions (read/write/execute) resides in the setting of each file's *permission bits*, also known as *mode bits*. Each file has a group of nine permission bits which can be individually activated or deactivated by the owner of the file.

Three groupings of bits are reserved to control file security for each of the three user access classifications:

Owner	*The owner of the file*
Group	Selected individuals
Public	Anyone with access to the system

Each group contains a bit reserved for each of the types of security functions:

Read	Permission to read the contents of a file.
Write	Permission to change the contents of a file.
Execute	Permission to run the computer program contained in a file.

The format for permissions as they are maintained in a directory is defined by three series of three mode bits, as follows:

```
rwxrwxrwx
|  |  └──Public access permission bits
|  └──Group access permission bits
└──Owner access permission bits
```

The **r** stands for *read,* the **w** stands for *write,* and the **x** stands for *execute.* If a letter (permission bit) is absent from any one of these groups, the missing letter indicates that the particular permission for that access level has been denied.

Since most user files contain data, the **x** (execute) character would not be present in the permissions code, and the permission pattern would be:

```
rw-rw-rw-
```

The dash (-) is entered in place of a missing permission mode bit. When a user makes a file, the operating system automatically excludes the execute permission bits. It assumes that the greater majority of the files that users make will be data files.

If the owner of a file did not want any other users to have any access to his file, the pattern would appear as follows:

```
rw-------
```

Further, an owner may wish to protect a file from his or her own inadvertent write-over. In this case the pattern would appear as follows:

```
r--------
```

The permissions bits may be changed with the **chmod** (change mode) command. Details on using this command will be covered in lab session 5.

Registered File Owner

Every file and directory in the hierarchical file structure has a registered owner. The owner of a file is generally the Unix system or a user. The name of the owner is maintained in the parent directory along with the disk block address, file size, etc.

The registered owner of a file is automatically changed by the operating system when another user (who has been allowed to do so by virtue of the permission bit

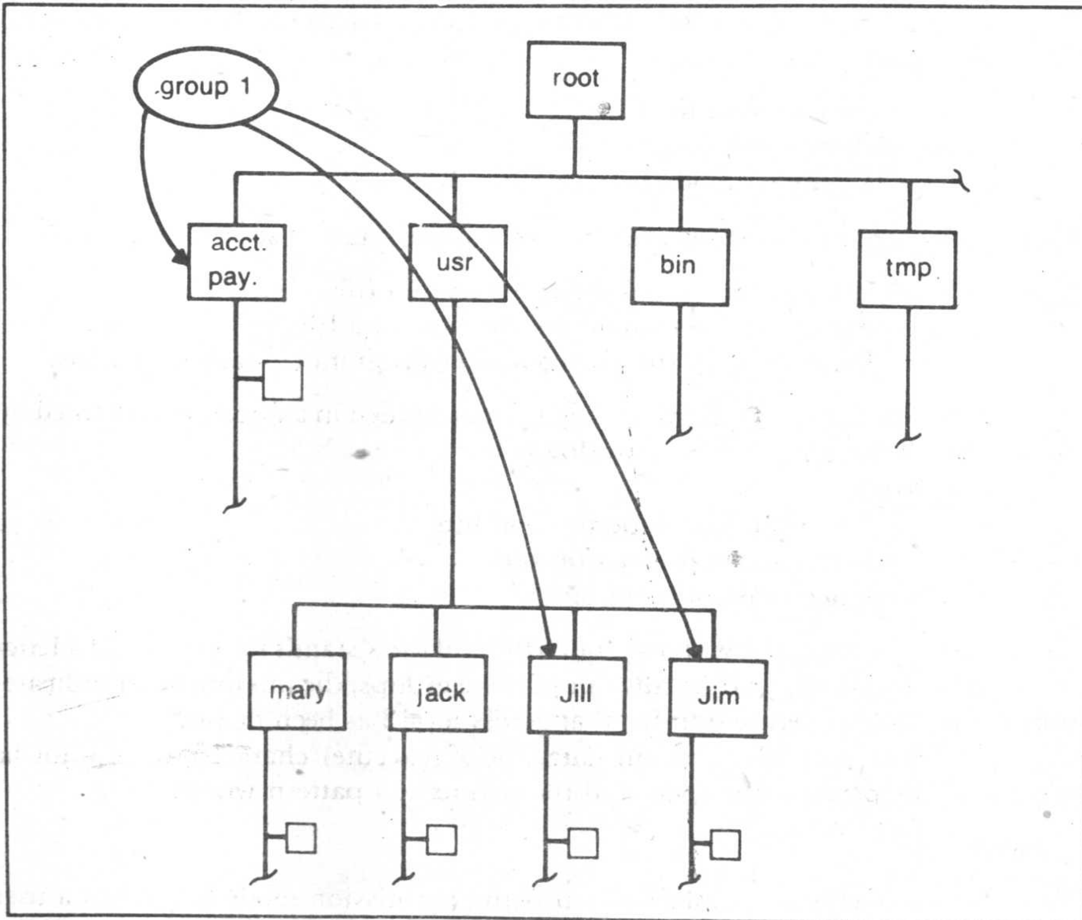


Fig. 3-12. Group overlay on the hierarchical file structure.

settings) makes a copy of another user's file. The copy command includes instructions for changing the name of the registered owner. It also may be changed by the system administrator.

The Group

We have mentioned three levels or types of users with regard to file security: the owner, a member of a group, and all of the users registered to use the system (the "public").

Access by an owner and by the public are self-explanatory. The purpose of the "group" designation is to effect a method whereby selected users (a group) may have a level of access to a file without opening the files to the public. A typical example would be to allow the accounting clerks to have limited read, write, or execute access permission to the accounting files.

The group can be shown pictorially to better explain the interrelationship between the group structure and the hierarchical file structure. Figure 3-12 shows how the group structure is a single-level structure joining users to a common set of files.

A user may be given access to one or more groups. A member of a group may gain special file access privileges or likewise be denied file access privileges. The permission bit settings for the group designation takes precedence over the public settings. For example, a member of a group who has the "group read" permission bits turned off will not be able to read the file—even though the "public read" permission bit is turned on.

LINKING FILES

Files stored in the Unix file structure may be known by several different names, or they may be listed in several different directories—just as a telephone number may be listed in several different telephone directories.

The Unix linking feature serves some very practical purposes. Since Unix commands (remember, commands are files) can be known by several names, if you do not like the name of a particular command you can give it an *alias*. For example, the command **cd** could be given an alias of **ch.dir**, **move.to**, or any other name that suits your purpose. (Linking file names is demonstrated in Chapter 13.)

Several users may be given direct access to a file without making actual copies of the file. Linking a file to more than one file name does not duplicate the file and,

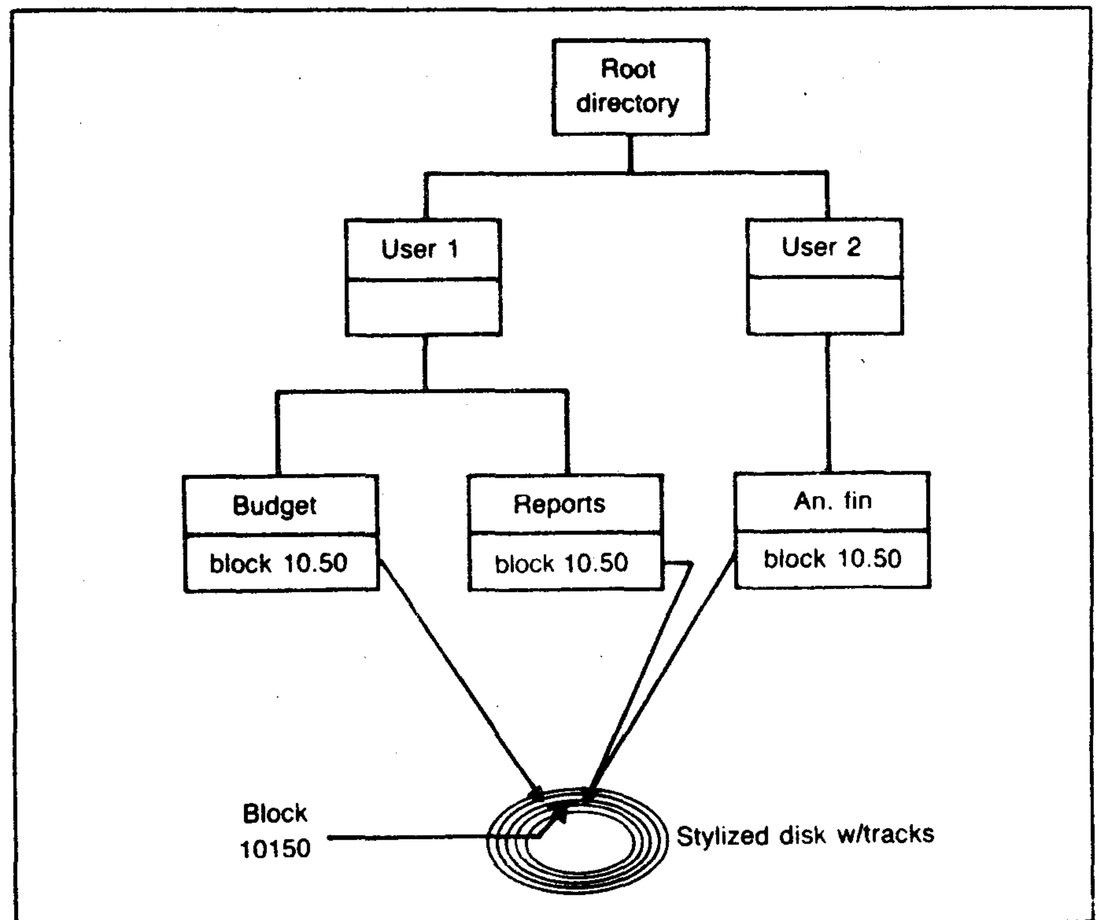


Fig. 3-13. Multiple directories with a common file location.

therefore, does not consume additional disk storage space. Different users may each give the file a unique name or use the existing file name; the pathname will constitute a unique file name for the file itself.

Linking also enables you to keep updated a single file, such as a company telephone directory. Anyone who has access to a common file may update it. Since the file is common to all linked file names, all changes made to the file will be readily available to all users who have access to the file.

If you have several directories in your file structure, you may have a need to make links to a file in one of them that has some data common to each. Linking would give you easy access to this file without having to move to the directory containing that file.

Figure 3-13 shows several files linked to a common file. To explain how this occurs in the system, we have to go back to three previous discussions: the discussion on how a file is stored and located on the computer's disk (auxiliary memory), the discussion on the contents of files, and how directories link the names of files to their locations on the disk.

If you remember, we discussed that a file is one or more blocks on a disk, and that every block on the disk has a specific address, just as every house has a specific street address for receiving mail. To get a file from memory, you need only tell the operating system to get the contents of a block at a particular disk block address.

Directories maintain the listings of files versus their location on the disk, and they allow us to name our files so that we do not have to remember the numeric location of each file, just as the telephone directory links the names of people and companies to their numeric telephone number.

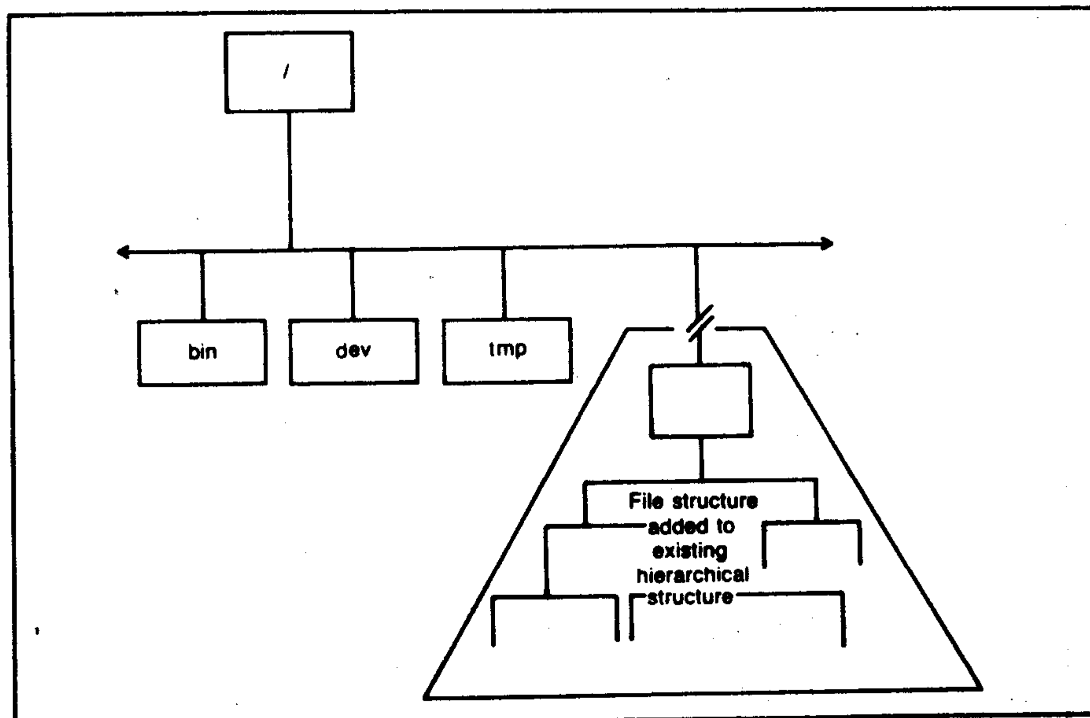


Fig. 3-14. Mounting a file structure.

Each file in the hierarchical structure is unique, and the information contained in the file is totally isolated from the information in any other file in the structure. Therefore, the information that we put in one file has no bearing on what we put in another file.

Since the information in a directory (a file) only provides information for finding a file—that is, a file's block address on the disk—it should be reasonable to assume that we could list the block address of a file in as many directories as we pleased, just as we could list the same telephone number in any number of telephone directories.

Now, let's say that in your copy of your telephone directory you have a friend listed by her maiden name. After she marries, her new friends list her telephone number in their directories under her married name. She is the same person, and she is called at the same telephone number, regardless of whether you call her Mary Smith or Mary Jane.

Linking files does not alter the integrity of permissions on a file. That is, just because you have created a link to another user's file does not mean that you can do more than the permissions granted. The permission bit status determines whether you will get in, just like you can walk up to the front door of any house, but if you are not recognized you will not be admitted.

Changing the permission bits is demonstrated in the lab exercises in Chapter 12, and examples of file linking will be demonstrated in the Chapter 13 lab exercise.

MOUNTABLE FILE SYSTEMS

Separate file systems may be *mounted* and *unmounted* (Fig. 3-14). This feature usually pertains to Unix systems installed on larger computer systems that use disk packs and/or reels of tape.

The directories and files contained on the disk or tape that is being mounted on the system or being added to the computer system is a file structure. The top directory on this disk must be joined to the system, providing a reference between the resident file system and the disk being added. Likewise, when a disk is removed from the file system, the directory reference to the top directory on the disk must be terminated.

The details concerning this subject are beyond the scope of this text, but it has been mentioned because the technology is advancing into the microcomputer world, and it is something of which you should at least be aware.

SUMMARY

The reason for having presented such detailed technical information about directories and the hierarchical file structure is that they are very important to your learning to use the Unix system. This discussion sets the stage for discussions on moving about the hierarchical file structure, storing and retrieval of files in the structure, understanding how the location of a command in the structure can affect its operation, transferring files, shell programming, and other operations in the Unix system.

Chapter 4

Commands

Chapter 1 discussed computer operating systems and the fact that Unix is but one of many computer operating systems available. Chapter 2 took the Unix system apart, so that you could gain an understanding of the relationships of the inner workings of the system itself. Chapter 3 explained the interface of the Unix system with the computer system. Now we will discuss how the shell and commands operate the Unix system.

The topics we will be covering in this chapter are:

- The types of shell programs available for operating Unix systems.
- The roles of the different shell programs in operating your computer.
- A detailed discussion on commands and command names.
- A detailed discussion on the different command formats for Unix commands.
- The role of the command option and command argument.

This fourth chapter is of particular importance for two reasons. First of all, it begins the explanation on how you operate the Unix system. Second, it provides important background information for the lab exercises that follow.

SHELLS

In an earlier chapter you learned that the Unix system is operated with or through a computer program called a shell. We discussed how the shell program (hereafter called the *shell* so as not to be confused with *shell programming*) is

designed to read the commands that you enter at the terminal keyboard, find the file containing the computer program that will perform this command, and finally initiate the execution of the command. In this chapter we will discuss in detail the operation of this function of the shell, referred to as *command interpretation*.

First of all, there are several categories of shells and a number of different programs within each of these categories. The difference between the categories of shells is based primarily on the type of operating service of the shell or the application of the shell. For example, there are different shells for different computer operating experience levels of users, for different operating needs of users (business, program development, etc.), and, finally, for particular application requirements (accounting programs, word processors, etc.).

In general, there are three types of shells:

System shells
User menu shells
Application computer program shells

System Shells

The primary purpose of a shell is to run the other computer programs resident in the computer's memory. These include utility programs (commands), word processors, electronic spreadsheets, etc.

We discussed this point earlier, but we did not mention that there are several different shells available for use with Unix systems. The standard AT&T Unix shell, also known as the Bourne shell (file name **sh**), is the basic shell.

The well-known Berkeley C shell (file name **chs**) is the standard shell supplied with the Berkeley Unix system.

A lesser-known shell is the Korn shell (file name **ksh**), named after its inventor, David Korn. It is described as being compatible with the Bourne shell. Information about the Bourne shell is applicable to the Korn shell.

There is also a special-purpose or special-application shell called the *restricted shell* (file name **rsh**). It is similar to the Bourne shell, but it is designed to restrict a user's access to certain parts of the Unix system. For example, it will not allow the user to:

Move to another directory.
Change his or her PATH file for accessing commands.
Enter a command line containing the / character.
Redirect output with > or >>.

Other restrictions also may be added. (The use of the > and >> symbols will be explained in Chapter 5.

One of the chief uses for the restricted shell is to isolate new users from inadvertently getting into areas of the system where they have no business. The operating system security measures will eliminate the possibility of a new user from accessing secured areas and files, but it never hurts to insure a sure thing. It also helps to keep the new user from getting lost in the hierarchical structure.

The principal shell in IBM's PC/IX is based on the AT&T Unix shell and is enhanced with shell features from the PWB/UNIX (Programmer's Workbench).

UniSoft's UniPlus+ shell is also a composite program. It is composed of features from the AT&T Unix shell, the C shell, and other Unix-based systems shells.

Users familiar with the operation of the AT&T and C shells claim the C shell (and shells with C shell features) is more serviceable than the conventional Bourne shell. The features unique to the C shell were designed specifically to contend with the problems of distributed utilization of the computer throughout the University of California Berkeley campus. The C shell is generally considered to be better for interactive use.

Some of the additional features of the C shell are:

history	This feature is used to record and recall from a history file the most recently entered commands.
command alias	This feature allows you to alter easily the operation of standard commands.
negation switches	These are internal switches which can be set to negate the normal operation of specified commands, e.g., such as, eliminating the accidental overwriting of a file when using the redirect output command, >.

There are also advanced user-support attributes which are highly valued and praised by experienced C shell users.

Menu Shells

Menu shells generally operate "on top of" the systems shell; that is, they are a shell on top of the shell. The menu items are linked to Unix commands and merely provide an alias name for the Unix commands.

Menu shells add a "user-friendly" attribute to the operation of a system shell. They were designed for use by individuals with little or no experience in operating computers. Just as the shell serves as an interface between the complex operation of the kernel and the computer, a menu shell serves as an interface between the somewhat cryptic Unix commands and a user unfamiliar with the operation of a computer.

Menu shells are usually a hierarchically structured network of menus or lists of Unix commands. Figure 4-1 is an example of part of the hierarchical menu structure for the Altos 586 computer. The menu shell will display the "Master Control Services" menu when you log on. Thereafter, you can display the subordinate level of menus by entering the item letter adjacent to the desired menu item.

The menu items also provide the full command name so that you do not have to memorize cryptic command names. Some menus even provide command descriptions so that you do not have to even memorize Unix command functions.

Figure 4-2 provides two examples of shell menus for the Altos 586 computer, as used in conjunction with the Xenix system Bourne shell. Table 4-1 provides a comparison of the menu items with their counterpart Unix commands.

Systems operating under a menu shell are generally referred to as *menu-driven* because of the nature of the menu shell operation. Commands are typically entered by simply entering the letter or number associated with a particular menu item. For

example, to enter the change directory command (command name **cd**) in Fig. 4-2, you need only enter an **a**.

Application Program Shells

Many application programs include their own shells or shells that operate on top of another shell. Most of these shells are similar to the menu shells in that they are used to drive the system shell.

The difference between application program shells and menu shells is that application shells contain menu items that are specifically related to the operation of the application. For example, a word processor menu may contain items such as

Technical Note: Shell Operation

It may have surprised you to learn that several shells may be resident in one system, i.e., stored simultaneously in the computer's disk memory. It may be even more surprising to learn that several different shells may be in operation simultaneously for different users on a single system.

This actually should not be too surprising if you remember that Unix is a *multi-user system*, from the point of view that the system operates as if you are the only user currently using the system. Concurrent with this explanation, it does not matter what program each individual using the system is running on a multi-user system, and the shell is just another program.

One of the main features of the Unix operating system is that it is able to manage the processing of several or many users' programs by processing in rotation a portion of each, thus sharing the computer's processor. This is called *timesharing*. However, the sharing is done in such small fractions of a second that it does not appear that you are sharing the computer's processor.

As each user's program is entered for processing, the operating system puts the appropriate tools into the processor for accomplishing each task. The operating system keeps track of all of the processes by assigning a unique process number to each process, thus avoiding a conflict between the operation of various shells and other computer programs.

There can be much more technical explanations of timesharing, discussing i-nodes, process swapping, time-slicing, etc., but this subject matter is of peripheral importance to the type of user for whom this book was designed.

In Chapter 2, I implied that the shell is an integral part of the Unix operating system. From an operational point of view, this is true. However, now that you have been introduced to the reality that a shell is a stand-alone unit, you may begin to question the validity of the statement. From a technical point of view, the shell is not truly part of the "operating system," which is primarily the kernel.

(Start) Altos Computer Systems
User: admin Business Shell Wed Jun 29 02:32 1983

Basic Utilities

a. Change Directory
b. Change Password
c. List Directory
d. Create a Directory
e. Remove a Directory
f. Edit a file (ed)
g. Remove a File
h. Copy &/or Combine Files
i. Display Files
j. Print Files

System and Help

k. System Administration
l. Electronic Mail
m. Run a program
q. Quit (logout)
r. Help

Type a letter to make your selection >

ALTOS PORT CONFIGURATION UTILITY

now loading port configuration information . . . loaded!

Port	Speed	Login?	Device-Type
console	9600	yes	ty1950
tty2	9600	yes	dumb
tty3	9600	yes	dumb
tty4	9600	yes	dumb
tty5	9600	yes	dumb
tty6	9600		dumb

Type the only first letter of a command; then press RETURN
Each command prompts for input as appropriate.
All input should be in lower case.

COMMAND ACTION
c Change a port-device-speed-login assignment

c Change a port-device-speed-login assignment
d Display all port assignments
h Display this "help" message (same as typing ?)
q Update the system port configuration tables (if necessary), then terminate the program
x Exit the program immediately; make no changes

RETURN Pressing RETURN leaves a setting unchanged
DEL Pressing DELETE or RUBOUT aborts the current operation

Commands: (c)hange port, (d)isplay, (h)elp, (q)uit, e(x)it
Command?

(Diskmgt)
user: admin Wed Jun 29 17 28 1983

DISK MANAGEMENT

a - Remove Accountant Tutor
b - Install Accountant Tutor

c - Backup
d - Restore

h - Help

Please make your selection or press Return to go to the Master Menu

(Admin)
user admin Wed Jun 29 02.23 1983

MASTER CONTROL SERVICES

a - The Altos Business Shell
b - The Altos Business Solution
c - The Altos Port configuration utility

d - Install Software Packages

e - Disk Management
s - Shutdown the System

h - Help
q - Quit (logout)

Please make your selection -

Fig. 4-1. Shell menu screen display.

(Start)
User: admin Wed Jun 29 02:43 1983

THE ALTOS BUSINESS SOLUTION

Applications

a. Accountant
f. Financial Planner
w. Word Processor

e. Electronic Mail
b. Business Basic III

k. Backup Important Files
r. Restore Files

h. Help
q. Quit

Type a letter to make your selection>

(Installabs)
User: admin. Wed Jun 29 17:26 1983

THE ALTOS BUSINESS SOLUTION

Do you want to install:

a. Altos Accountant
b. Business Basic III
e. Altos Electronic Mail

f. Executive Financial Planner
w. Executive Word Processor

p. Altos Port Configuration Utility

h. Help
q. Quit

Type a letter to make your selection>

OFF

To become the System Administrator, login as "admin". This will allow you to execute the Business Shell as well as the Altos Business Solution

You'll install the different Software Packages from "admin" also.

The Altos Port Configuration Utility allows you to CONFIGURE the terminals that you'll be using. So, if your terminals do not work properly please run this utility.

NOTE: When you finish the Altos Port configuration you have to shutdown the System and Boot it up again.

(Type return to continue)

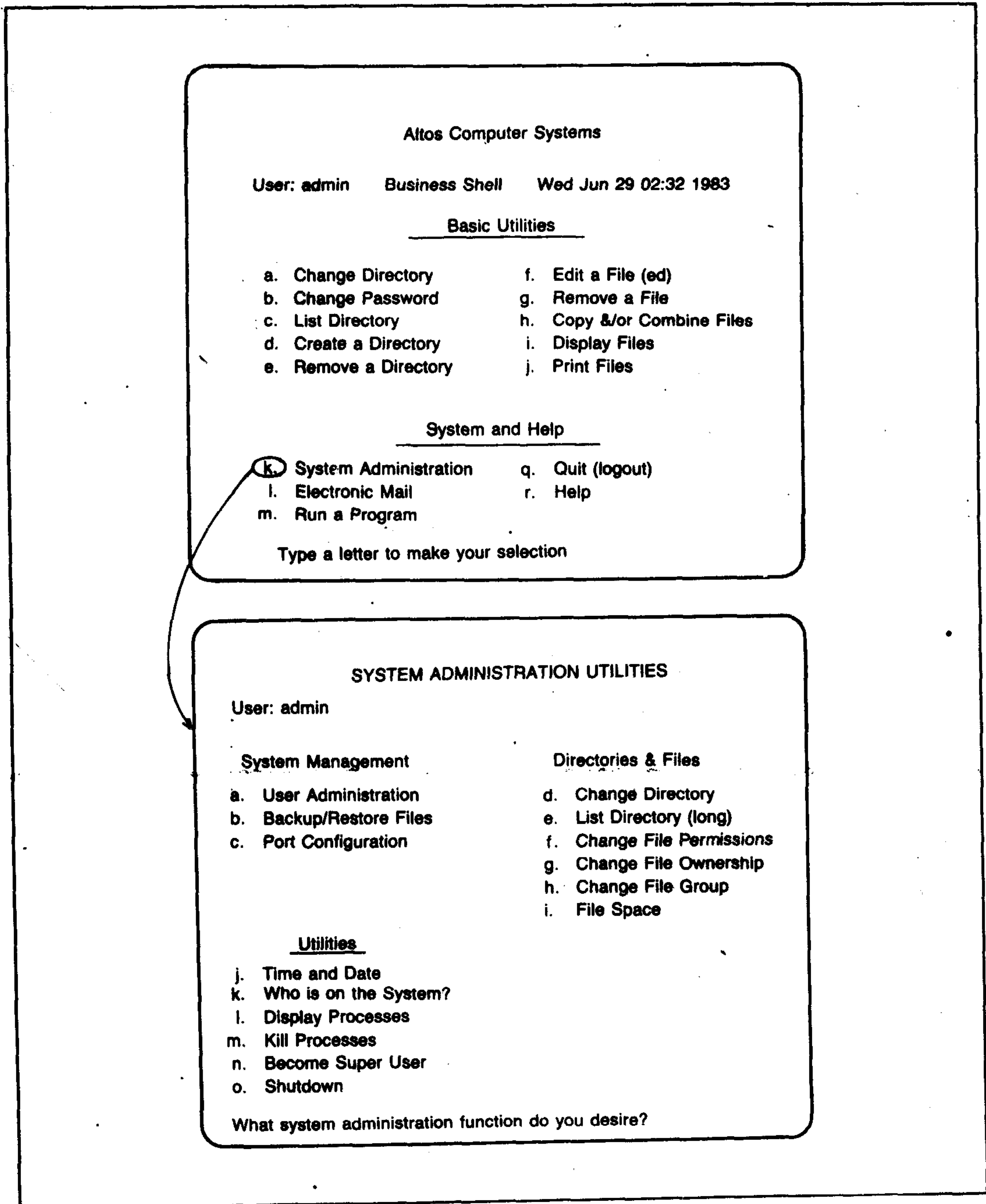


Fig. 4-2. Examples of shell menus used in conjunction with the Xenix system Bourne shell.

"cut and paste," "delete a line," "center text," etc. The commands on the menus may be Unix utilities, Unix utilities with alias names, and/or commands unique to the application.

Selecting a Shell

Many Unix systems are being shipped with multiple shells. For example, Xenix on the IBM PC-AT computer and UniSoft's UniPlus+ contain both the AT&T and Berkeley shell. In addition, Xenix also contains a shell called Vision. The prime significance of this is that a user accustomed to the operating characteristics of one or the other of these shells can operate another Unix system with little or no familiarization with the new computer, itself.

When a user name is registered on the system, one of the registration requirements is to indicate the shell under which the user will "come up" when he logs on the system—assuming, of course, that more than one shell is available. The shell selected is based on a user's anticipated system usage, shell preference, or operating requirements. It is even possible for a user to log on and automatically be brought up under the operation of an application shell such as an accounting system.

The advantage of having several shells resident on a system is that users with varied experience levels may operate the Unix system with a tool suited to their individual levels of operating capability. For example, a user with no experience or training in the use of the Unix system would be able to operate the system competently under a menu shell. On the other hand, an experienced user would be frustrated by having to step through a network of menus in a menu shell.

Since a shell is a file, a user may change shells (i.e., operate under another shell) at any time simply by entering the name of the file containing the desired shell, just as you would enter a command. After the new shell is entered into the computer's processor, the operation of the old shell will automatically be suspended.

As mentioned, many application programs contain their own shells. As soon as the application is in the processor, the application shell is put in control of the processing of any further commands. When you terminate the application program, the processing of the last instruction contained in its termination command program will cause the resumption of the operation of the shell that was in operation when the application was initiated. The system shell is terminated when you log off.

THE COMMAND

In Unix, commands are merely the names of executable files. These files contain the instructions for making the computer perform a desired function, task, or operation. When you enter a command at the terminal keyboard, you are in effect entering only the name of the file containing the program that performs that specific function.

If you should enter the name of a data file instead of a command, the shell will not be able to process it, and in some cases a message like "not found" may be displayed on your terminal monitor.

Table 4-1. Shell Menu Items vs. Unix Commands.

Menu System Menu Items	Unix/Xenix Commands
Port configuration utility	pconf
Backup	tar
Restore	tar
System shutdown	shutdown
Quit or Logout	quit
Change directory	cd
Change password	passwd
List directory	ls
List directory (long)	ls -l
Create a directory	mkdir
Remove a directory	rmdir
Edit a file	ed
Remove a file	rm
Copy and/or combine files	cp
Display files	cat
Print files	lp or lpr
Time and date	date
Change file permission	chmod
Change file ownership	chown
Change file group	chgrp
File Space (available)	df and du
Who is on the system	who
Display processes	ps -alx
User administration	ua
Kill processes	kill
Run a program	'program name'
Login	login
Electronic mail	mail
Becoming super user	su
Format a diskette	format
Copy a diskette	fcopy

Some of the most frequently used commands actually are contained within the shell itself. By putting these commands in the shell, the shell does not have to take time to search through the hierarchical file structure for them. That is, when the shell is loaded into the computer's memory, these commands are also loaded into memory—because they are part of the shell program itself.

The following is a very small sampling of the commands contained in the shell. A complete listing of these commands will be found in your operating system command reference manual under one of the shell file names, such as **sh**, **chs**, etc.

alias — A command used to create alias for standard Unix commands. This is similar in some respects to the idea of linking files, but is more specifically directed at establishing modified versions of Unix commands.

cd (file) — This is an order to change the current or working directory to the one enclosed in the parentheses.

echo (. . .) — This is used to "echo" the specified argument enclosed in the parentheses.

exec (command) — The shell will execute the specified command without

"forking" (making a copy of the command). This overrides command currently in process.

read (file) This reads lines from a file.

login (user name) This command used to log in a new user directory without first exiting from the current user directory.

Technical Note: File Search Procedure

The shell is programmed to search for the words that you enter at the terminal keyboard, such as a file name or command. The shell begins its search principally in one of four directories, in the order specified in your **profile** file. (More directories can be added and/or the sequence changed.)

The normal ordered sequence of directories through which the shell will conduct searches is as following:

The shell itself
Current directory file structure
The directory /**bin**
The directory /**usr/bin**

If a relative or full pathname for an item (command or other file name) is used, the shell will search only the specified directory for the words (file name) that you enter.

The order of directories to be searched can be changed or directories added by modifying the shell variable, **PATH**, with information stored in the user's **.profile** file. This file contains each user's operating parameters, including the order of the directories in which the shell is to look for a specified file name. The **.profile** file is set up initially by the operating system when a user is added to the system, but the **.profile** file can be changed.

The order of directories in which the shell looks for an item is important—particularly if you have set up, for example, a file with a name identical to that of an existing system command. Though the full pathname for both your file and the command are unique, the file that is executed will depend on the sequence in which the directories are searched. The first command found in the ordered sequence of the directories will be the one that is executed.

This information is particularly useful if, for example, you want to run your own modified versions or specific command options instead of the standard Unix commands. (This will be explained further in Chapter 6 and examples will be provided in the lab sessions.)

At this point it is only important that you understand what is going on in the system, so that when you get a command function different from what you expected, you will have some idea of why it might have happened. Other than this, it is not likely that you will have to get this involved with your Unix system at this level.

Technical Note:
The Interaction Between Commands and Shells

The operation of a shell is temporarily suspended whenever a command is being processed. After the shell initiates the running (execution) of a command, it is the command that contains the instructions for controlling the operation of the kernel for the performance of that task. The execution of the last instruction in each command program will cause the operating system to resume the operation of the shell. At this time the shell is ready to process the next command.

This discussion of the interaction between the shell and commands, however, is purely academic, because the suspension of shell operation during the processing of each command is measured in microseconds. Though this point is moot, it is presented so that you can understand how the instructions contained in a command program actually interact with the operation of the system.

As you can see, Unix commands (file names) are often very short abbreviations or acronyms for the names of the Unix commands. For example:

ls List the files contained in a directory.
pr Print out the contents of a file.
cat Concatenate, which is used to display the contents of a file.
ps Process status, i.e., list the "processes" (programs) currently being run by the computer.
pwd Print working directory, i.e., display the pathname of the directory in which you are currently located in the hierarchical structure

Shorter command descriptions often maintain their full descriptive name in the command, however.

date Display the current time and date.
mail Send or read mail sent between the users.
find Find the pathname for a specified file that you cannot find.

The objective for keeping the command names as short as possible is so that they will require as little typing as possible to enter them. Many critics consider the command names "user-unfriendly." If the command names were longer or more descriptive, however, I am sure that there would be critics who claim that the extra time that it would take to enter the longer command names would also be "user-unfriendly."

Command Formats

The format for a typical Unix command is the command name, followed by options to the command, other file names, and/or text expressions. Each of these items is delineated by entering a space after it. For this reason, file names cannot

include spaces. Items containing more than a single continuous string of characters (*text expressions*, for example) often need to be enclosed in quotes.

Command Components

Commands are composed of two or three types of components, based on the two schools of thought described in various textbooks. Some authors and experienced users feel comfortable in describing the Unix command as being composed of the command and zero or more optional words called *arguments*, in the following format:

```
command argument argument argument . . . . .
```

where . . . . . means more arguments. Others describe the command format as being composed of the command, command options (sometimes called *flags*), followed by arguments, in the following format:

```
command options argument argument . . . . .
```

Technically speaking, anything following the command is an argument. Some users, however, like to consider the argument to mean the file, directory, or anything that could be described as "that upon which the command works." They take the option to mean one of the command alternatives, a special application feature, or a command modification provided in the utility program itself.

The reason for the differences between the formats for commands (i.e., lack of standardization) is that Unix commands were developed by many people, and in a relatively relaxed working environment. I do not have trouble accepting either or both formats.

Some command structures are very complex and may have options and arguments intermixed; I therefore suggest to you that it is more important simply to learn how to use the commands that you need, rather than worry about their lack of adherence to some standardized format. We will let "/usr/group," the recently formed Unix standards committee, resolve the standardization problem.

Command Terminology

Command. A command (Unix utility) is the name of an executable file. There are more than 300 Unix utilities currently available (see Appendix B). Commands also can be created by users, with programs placed in a file and the appropriate execution permission bits set.

Command Options. A few of the utilities in the Unix system have no *command options*, but most of them do. Options modify the behavior of the command depending on the option (or combination of options) that are used. Some commands may have upwards of 10, 20, or more options.

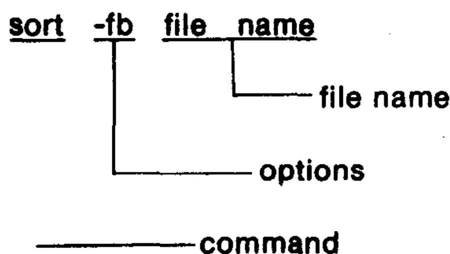
An option is generally a conditional enhancement of a command. For example, the **ls** command will list the names of the files subordinate to a directory, but the **ls -l** command will list considerably more information about each of the files in the directory.

An option also may embody modifications to the operation of a command. For example, there is a command named **sort** that you can use to sort the contents of a

file alphanumerically. The basic command (used without options) is programmed to sort automatically the contents of a specified file in alphabetic order, A to Z. The available options with which you may choose to alter the sort will, for example, sort Z to A, sort in normal or reverse and, numeric order, etc.

If you were to consider each option to be a unique command, which in many respects they are in actual usage, the library of Unix System commands contains more than a thousand, and the list is still growing.

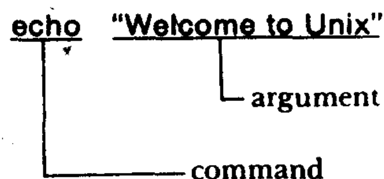
An option is normally preceded by a dash (-), but there are a few exceptions. If a second or further option is used with a command, however, the second and following options generally are not preceded by a dash. That is, a dash is not required before the second, third, or subsequent options. For example:



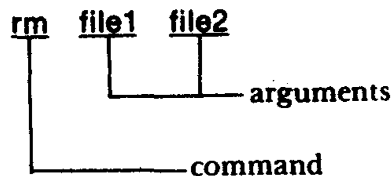
This command (command, options, and argument) will direct the operating system to sort a list of names or words in a file in alphabetic order. The f option will cause the command to consider all uppercase to be lowercase, and the b option will cause the command to ignore any blanks in the name. These options will enable the **sort** command to handle names like McCloud, and get them in the anticipated alphabetic order.

Argument. An *argument* is a value, a file name, a text-type expression, or other information acted upon by a command. Arguments begin with a space and end with a space. If an argument is to have spaces included within the argument itself, the argument (in most cases) should be enclosed within single or double quotes.

For example:



This command (command, argument) will cause the shell to display the statement, **Welcome to Unix**, contained within quote marks in the argument. For another example:



This command (command, argument1, argument2) will direct the operating system to delete the files named **file1** and **file2** from the file system.

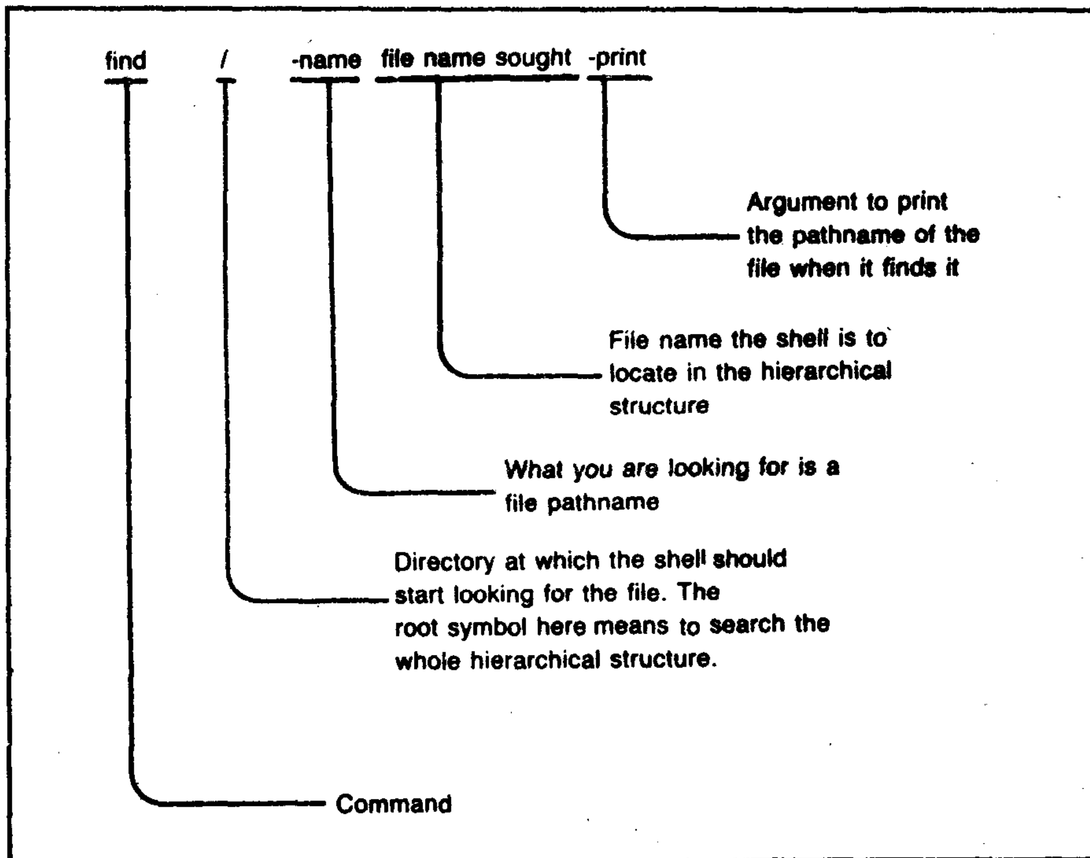


Fig. 4-3. Command line syntax for the find command.

The **find** command is an example of one of the more complex but often-used commands. This command is used to locate and display the full pathname of a specified file in the hierarchical file structure when you do not know its pathname. You might expect the command to be simply **find** and the file name; as shown in Fig. 4-3, however, the actual command is not that simple. (The **find** command is discussed in more detail in Appendix A.)

TYPES OF COMMANDS

When we use the word *command* in Unix, it is usually the Unix utility to which we are referring. Throughout this chapter we have been discussing the Unix utility command. There are other types of commands in Unix, but most of them are only used in more advanced utilizations of Unix, such as for software development.

A discussion of the types of commands is itself quite a complex subject, because the different types of commands are not all accessed (used) in the same way. That is, there are so many exceptions to any set of rules one could devise for defining the commands in any type category. Some commands in a type category may be accessed directly, some are merely the names of libraries of subcommands, and others may be accessed only by other commands.

Here is a summary overview of the different types of commands in Unix:

Utilities

- **Common commands.** Utilities are generally described as *publically accessible* commands. Appendix B provides a comprehensive listing of the types of services that these commands can provide.
- **Subcommand libraries.** The most common example of this class of command would be the ed (editor program) utility. The editor program contains about two dozen commands or command structures that you can use for entering and editing the contents of a file.

Shell Commands

Shells are listed in the Unix manuals with the utilities. Like the editor utility, shell utilities contain a number of command libraries. These are usually divided into three further subcategories:

- **Built-in Commands.** Some very frequently used commands, you will learn, are not utilities but are part of the shell software. They are there primarily to expedite processing.
- **Metacharacters.** These are a special type of built-in command. They are used frequently and appear to be utilities-themselves. The major difference is that they are usually symbolic characters instead of word expressions; they are used usually in conjunction with the other types of commands instead of as stand-alones.
- **Conditional Commands.** Conditional commands also are used in conjunction with other commands, primarily in creating shell programs. They provide "decision-making" software for shell programs.

System Calls

Most system calls are accessed by other software, but seldom directly by a user. They are the point of interaction with the kernel.

Subroutines and Libraries

Software development Unix systems usually contain libraries of commands which support C and FORTRAN programs, and specialized libraries for lexical analysis of text, cursor control, database management, etc.

SUMMARY

In this chapter we have discussed the different kinds of shell programs available with Unix systems, as well as their operation and purpose. We discussed the command in some detail and explained the roles of command options and arguments. Now it is time to learn more sophisticated operations with the shell.

Chapter 5 will describe some of the fundamental shell operating features, such as entering combinations of commands, using the pipes and pipe lines, making a filter with pipe lines, using the tee command, and running several.

Chapter 5

Shell Command Language

The shell is the software in the Unix system responsible for reading the commands you enter at your terminal keyboard and for determining what has to be done to get your commands processed by the computer. While technically there is no difference between the shell and the operation of other Unix commands, the shell is a very remarkable program.

The reason that the shell is remarkable is primarily because of the library of subcommands contained in the shell program itself. These commands are referred to as the shell's *command language.*

In very simple terms, the shell's command language can be described as a set of commands that can be used in a fashion similar to the way punctuation is used in the English language. The first type of commands we will discuss, called *metacharacters,* can be interspersed with other commands to perform some very remarkable services, as you will learn in this chapter.

METACHARACTERS

In the last chapter we discussed command formats. Learning about the different command formats, in a matter of speaking, is equivalent to learning some new words or how to use words in a foreign language. Now that you have an idea of what Unix utility commands look like, what they mean, and how to use them, it is time to discuss using commands to communicate with the computer by entering commands at your terminal keyboard.

In Unix, as in English, we can communicate in simple sentences, or we can use the available grammatical forms of punctuation as a tool to create more complex

sentence structures. We use complex sentence structures in writing because it is cumbersome to have to limit our expression of ideas to the simple sentence. For example, if I had to write this book, restricting myself to simple sentences, it would have been considerably thicker and possibly not as easy to read.

The equivalent of punctuation in the Unix language is the metacharacter. Metacharacters are in themselves just commands. The most significant differences between the metacharacter and the utility commands are:

- Metacharacters are a part of the shell, as opposed to being a command file in the hierarchical file structure.
- Metacharacters were designed primarily to control the operation of the command line, as opposed to processing data. This concept is similar to the purpose of punctuation, where punctuation is used to make the sentence clearer—as opposed to providing information additional to that being presented by the basic sentence.

The program instructions contained in metacharacter commands are processed in the manner described in earlier parts of the book. Their job in the computer does not change the way that the computer handles or processes its instruction set. We will examine metacharacters in greater detail in Chapter 6.

THE COMMAND LINE

Up to this point we have discussed the shell as the software for interpreting the commands that you enter at the terminal keyboard, and the fact that it contains commands that can enable you to create complex command statements. However, we have not actually commented on just where all of this action occurs.

Commands entered at the terminal keyboard will be displayed on the terminal monitor as they are entered. The command display (the line containing the commands, etc.) is referred to as the *command line*. A command line may be:

- A simple command without any options or arguments.
- A complex command format.
- One or more commands separated by metacharacters.
- One or more commands joined with metacharacters to create a more complex command.

Pressing the return key end a command line and indicates to the shell that it should proceed with processing the command line. At this time, the shell begins reading the command line you entered, splits the line into its component parts, and determines what has to be done to execute the ordered command line.

Some of the things you can do with more metacharacters and the complex command lines you can create with them include:

- Entering a series of commands instead of entering them one at a time.
- Joining commands together to perform a special job that would otherwise require several command lines.

- Grouping commands to isolate their effect on execution of another operation.
- Processing commands in the background, so that you can continue using your terminal to enter another, possibly more important job.

Command line syntax is fairly straightforward. The first item on a command line must always be a command. Thereafter, commands may be interspersed on the command line with metacharacters, text-type expressions, options, and arguments, observing the rules for using the commands themselves.

When you enter a command line, i.e., indicate to the shell (by pressing the Return key) that you are ready for it to begin initiating the processing of the command line, the first thing the shell does is check the command line to make sure that all of the syntax is correct.

If the command line is acceptable, the shell begins searching the hierarchical file structure for the specified files named in the command line. The remainder of the procedure has been discussed earlier. (The routine is actually more complex; however, a more technical dissertation on the processing of the command line is beyond scope of this book, but may be found in a number of other references.)

USING THE SHELL COMMAND LANGUAGE

Compound command lines come in a number of forms. Let's take a brief first look, and then examine in detail how to use some of the shell's metacharacters to form compound command lines.

- **Multiple-Command Command Line.** The simplest compound line is the *multiple command* command line, which is simply a series of independently executed command statements.
- **Pipe.** Another form of compound command line is a series of commands that are joined with metacharacters to form a unit that will perform a function or service. Examples of such units are pipe lines and filters.
- **Pipe Line.** When two or more pipes are used in a command line the command line is referred to as a *pipe line.*
- **Filter.** When pipes are used in conjunction with Unix commands that alter the contents of a text file (such as sorting the file), the command line is referred to as a *filter.*
- **Tee.** The tee command is a Unix utility, but is included here because of its close relationship with the type of command line operations we are describing. Tee command operation is almost identical to that of the pipe. It is used to join command statements, but it also allows you to display the output of the individual command statements. You might think of it as providing you with a sampling of what is going on, a "window" into, for example, a pipe line function.

- **Multi-tasking.** Two or more commands or programs may be processed simultaneously, one in the foreground (as discussed in Chapter 5) and one or more in the background.
- **Subcommands.** Metacharacters can be used to group commands for processing, and/or isolate a command (or group of commands) from affecting the operation of other commands or programs you are entering at the terminal keyboard.

Now let's increase our level of detail.

Multiple-Command Command Lines

The shell allows you to enter several commands in succession. The shell will place them in a temporary storage area called a *buffer* until the previously entered commands have been processed. Commands may be entered one at a time by entering a command line and pressing the Return key, or you can put several commands on a single command line and then press the Return key to initiate the processing of these commands.

To create a multiple-command command line, simply separate each of the commands (command, option, and argument, as required) with a semicolon:

```
cd; cat file1; mv file1 file2
```

The shell will execute these commands sequentially, in the order that they are entered on the command line.

The command reference manuals provided with Unix systems contain detailed explanations of the command line formats for options and arguments. This appendix also contains an introduction to the use of the Unix command reference manuals.

Pipes

In our discussion of command strings, we explained that you can string several commands together, separated by semicolons. The semicolon indicates the ending of each of the commands in the series. Another shell metacharacter for connecting commands together is the pipe, represented by the vertical bar (|) character.

The difference between ending a command with a semicolon and joining commands with a pipe is that a pipe, in addition to connecting the commands, will direct the output from the previous command into the input of the next command on the command line.

The pipe saves you the trouble of having to perform the intermediate step of writing (storing) the output of a command in a file and then using the file as an argument for the next command. For example, suppose you have a file named **file.name** that you wanted to sort. You could do it the long way:

```
cat file.name > new.file.name
sort new.file.name
```

Where **new.file.name** is the name of the file you must write before you can invoke **sort**.

Or we could use a pipe. The command line in Figure 5-1 will direct the shell to read a file and send the file through a sorting program that will list the contents of the file in alphabetic order and display the results on your terminal screen.

Pipe Lines

You can also connect two or more commands on a command line with pipes. When there is more than one pipe in the command line, the command line is referred to as a *pipe line.* An example of a pipe line is shown in Figure 5-2. This command will perform the same job as the last example and, in addition, the **more** command will automatically display the result of the sort on your terminal screen with pagination.

The **more** command is similar to the **cat** command in that it will display the contents of a file or (as used in our example) the results of the sort command, but it has an added feature called *pagination.* The **cat** command will "zip" through, displaying the complete file before you are able to read it. The **more** command will display one screen of information and wait for you to instruct it to display the next screen.

Filters

A *filter* is a pipe line that employs commands that will modify the contents of a file, such as sorting the contents of a file in alphabetical order, as we demonstrated in the second example of our discussion on pipes and pipe lines.

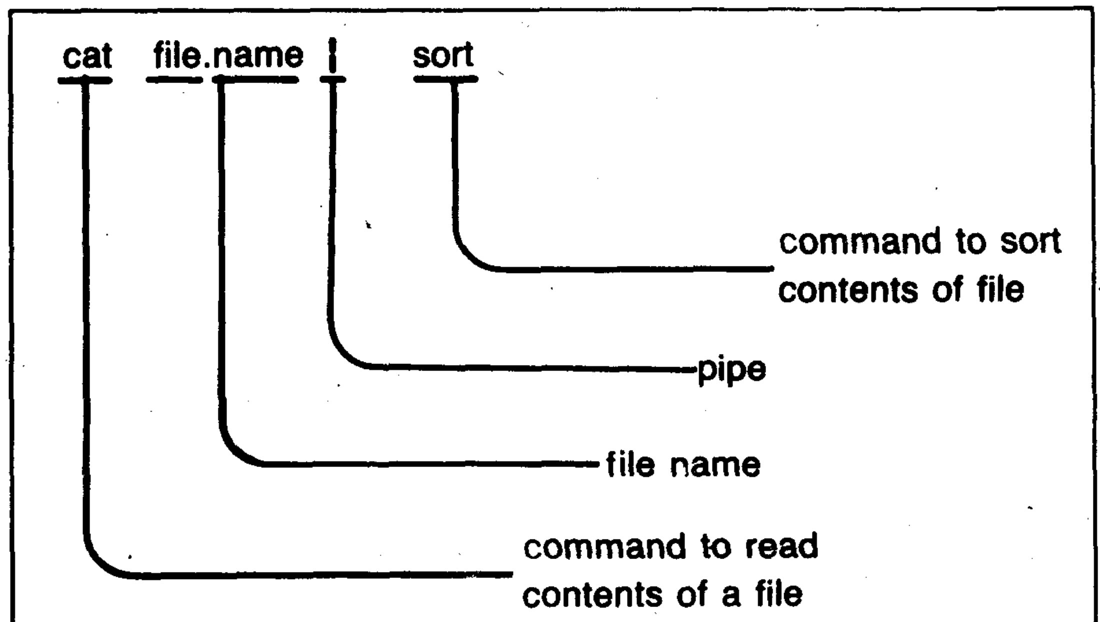


Fig. 5-1. The output of a command may be redirected into another command using a pipe.

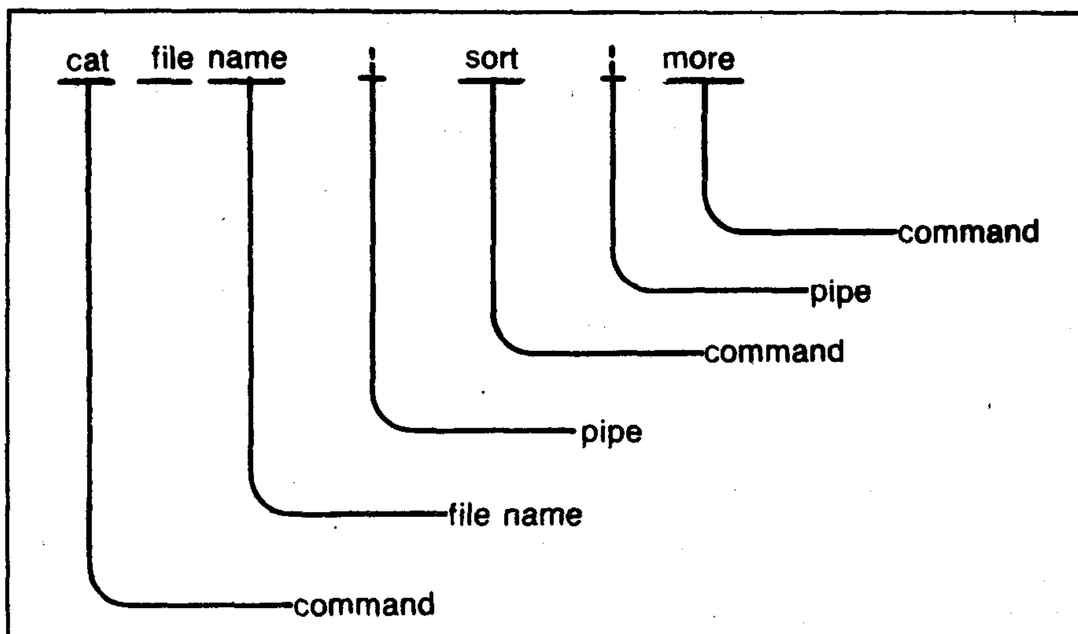


Fig. 5-2. More than one pipe in a command sequence forms a pipe line.

The Tee Command

The **tee** command is similar to the **redirect data** (>) command, with the additional feature that you can use it to direct data in two directions. The **tee** can direct data to a file and simultaneously to the terminal for display. The purpose of the **tee** is to enable you to see the intermediate output of processes on a command line.

A conceptual diagram of a command line using a tee is shown in Figure 5-3. The expression **tee** is a command just like **date, more, cat**, etc. You must therefore make the connection between commands with a pipe.

The command line for this diagram appears in Figure 5-4. This command line will list the contents of the files in the current directory, simultaneously displaying the list of file names on your terminal screen and saving the list in a specified file.

Background Processes

You have learned that the Unix system can accommodate the processing of instructions from more than one user logged on the system at the same time, because of its ability to timeshare the computer's processor. It should have been no surprise when you learned of background processes, also referred to as multi-tasking.

Do not confuse background processing of commands with the fact that you can enter several commands sequentially without waiting for the shell prompt to redisplay on your screen. Buffered commands, when processed, will be processed in the foreground unless you code them for being processed in the background.

You will note that the shell prompt reappears immediately after entering a command coded for background processing. By comparison with foreground

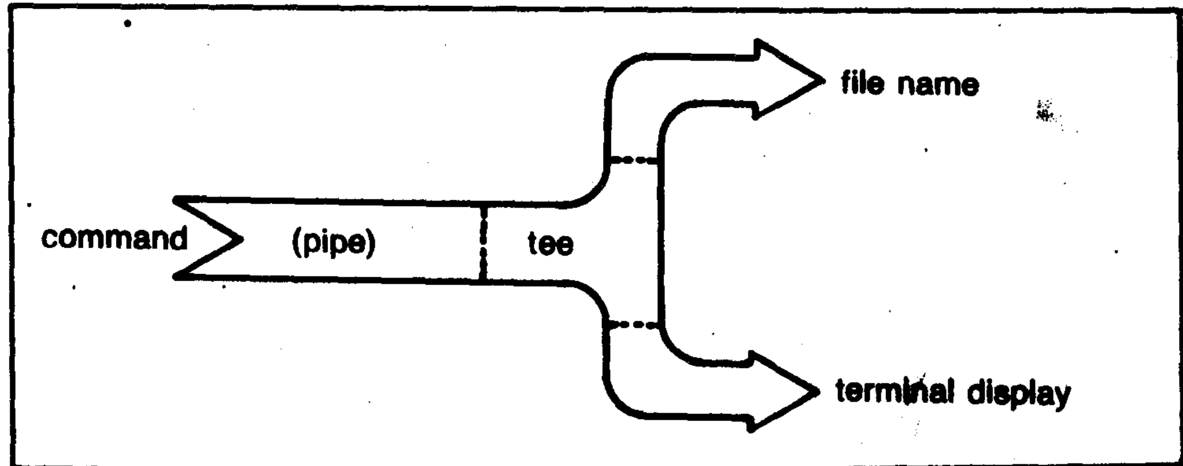


Fig. 5-3. Piping through a tee can send output to multiple destinations.

processing, the prompt only reappears after the command has been processed.

Background processes are scheduled for processing through the computer's processor at a lower priority, because the user is not waiting for the computer to finish the job and return the terminal to the user for entering the next command or program. At first this may sound as though the background process will be completed very slowly. In fact, however, there is so much processor time available on most microcomputer systems that there is ample time to "slip in" commands for background processing.

Examples of the use of the background feature of Unix are formatting a text file with the nroff utility, running a long program, printing a file, compiling a program, etc. To direct the shell to schedule a command or other computer program for background processing, you enter an ampersand (&) at the end of the command line.

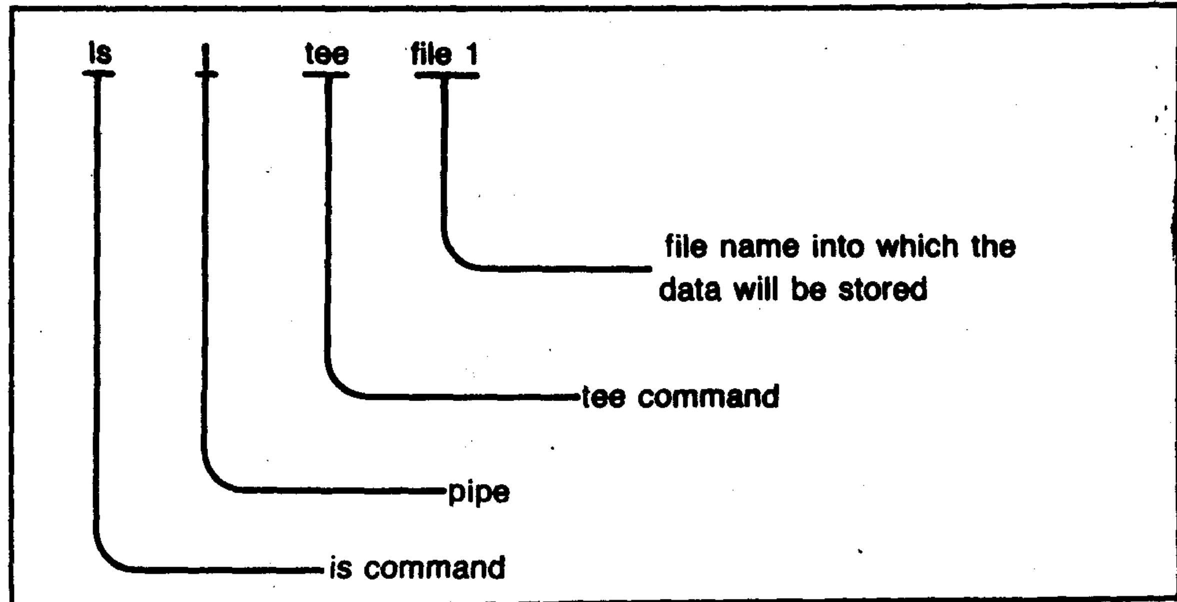


Fig. 5-4. Command line syntax for the tee command.

Figure 5-5 provides an example of the use of the **nroff** command, which is a utility that you can use to format the contents of a text file. After entering the command line ending with an **&**, the operating system will display a number. This number is the process identification number. If you had a very long program and you wanted to find out if the program is still running, you would enter the command **ps -alx**. The shell will display all of the programs in process. You would look up this number in the process status listing.

One of the problems with running a program in the background is this: When the computer has finished processing your program, the operating system will unexpectedly display the results on your terminal screen. Yes, this may interfere with something you are in the middle of doing. To avoid this possibly catastrophic situation, you can do something called *redirect output*.

Redirect output means that you instruct the shell to store the results of a background process (in this example, the output of the **nroff** command) in a file which you can read when you are ready. The command line format for this instruction is shown in Figure 5-6. The redirect command (>) is explained with examples in Chapter 11.

Subcommands and Command Groupings

Commands can be grouped, i.e., entered on a command line such that they will be executed as a package. If you remember from your high school algebra, there are differences in the way you process mathematical expressions, depending on whether they are enclosed in parentheses (grouped) or written without them. The same rules for processing algebraic expressions hold true for processing commands contained within parentheses.

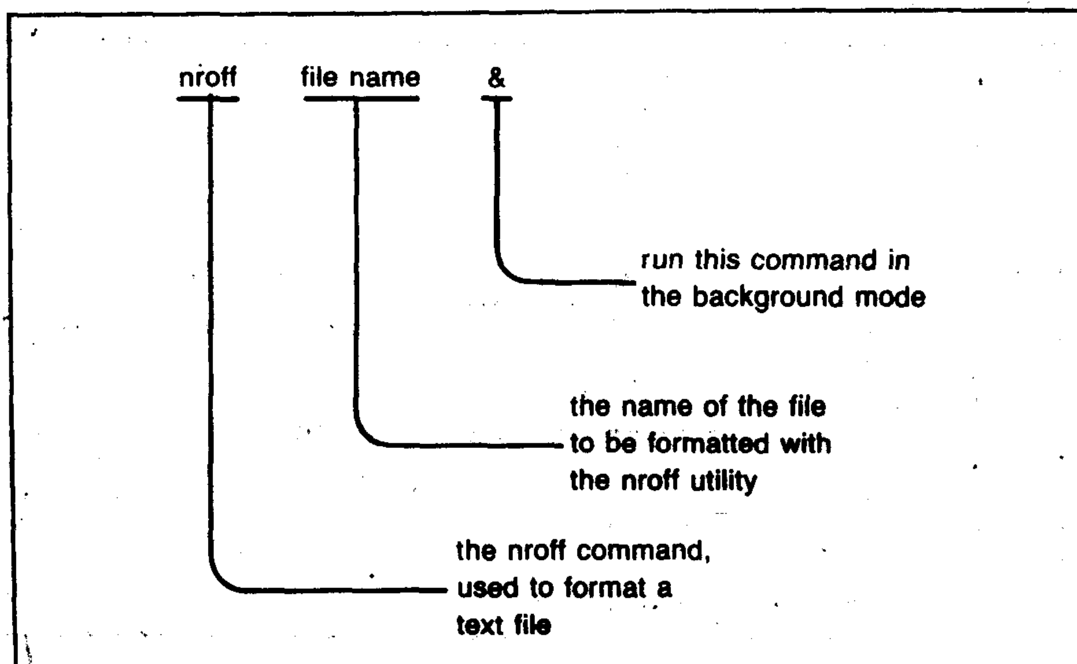


Fig. 5-5. Command line syntax for executing a command in background mode.

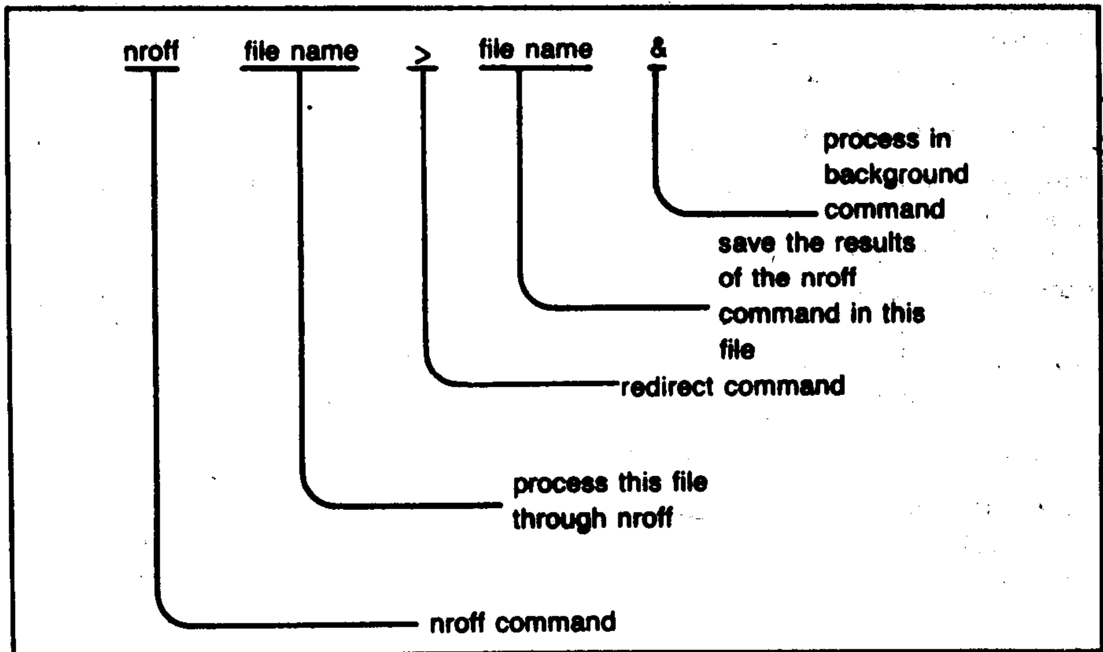


Fig. 5-6. Redirecting the output of a command processed in the background.

A typical example of the usage of the command grouping would be entering the following command to be processed in the background:

```
date;  who  &
```

The **date** command would be processed in the foreground and only the **who** command would be processed in the background, because the ampersand is not associated with the **date** command.

To move the whole command process in the background, you have to enclose the whole command in parentheses, as follows:

```
(date;  who)  &
```

Now the shell will read the command as a group of commands to be processed in the background.

In an earlier part of the book we discussed the pathname and the relative pathname. The pathname was described as the full routing through the hierarchical file structure to a desired file. The relative pathname was the part of the pathname required to direct the shell to a file with respect to your current location in the structure (also referred to as the working directory).

When you begin to work with Unix, you will learn that your activity will be relative to the working directory. By this I mean that accessing files and entering commands that will act upon these files are affected by your relative position in the hierarchical file structure.

The purpose for mentioning pathnames and working directories at this point is that you can also use parentheses to isolate a background command line from affecting the work you are currently doing. For example, if you were to enter the command to move to another directory (make another directory the working

directory) on a command line that you wanted to run in the background, the change directory **cd** command would also affect the work that you were performing in the foreground. For example:

```
cd /usr/joe/text; nroff file1 &
```

This **cd** command on this command line would move you to the directory **/usr/joe/text**. The **nroff** command is a text formatting command, and it will format **file1** in our example. The ampersand will instruct the shell to perform the **nroff** command in the background.

To save having to enter a second **cd** command to return you to the directory where you were originally working, you could have used parentheses in the first command line, as follows:

```
(cd usr/joe/text; nroff file1 ) &
```

This command will be processed as it was in our previous example. Because the command is isolated in parentheses, however, it will not change the current working directory.

Chapter 6

Introduction to Shell Programming

You have probably heard about something called "shell programming," or that the Unix shell "contains a programming language." The hoopla surrounding this subject would lead you to believe that this is very important to your learning to use Unix to operate your computer. On the other hand, the term *programming* makes it sound like something you would like to avoid if at all possible.

The average user will probably have little need for using the shell programming feature of the Unix system. Shell programming, while not necessarily a difficult subject, does begin to get into the intermediate-level operating features of the Unix system. We will cover it in any case, if for no other reason than the fact that it sounds terrifying makes it a ready challenge for demystification.

In Chapter 5 you were exposed to a very fundamental type of shell programming when we discussed putting more than one command (command, options, arguments) on a command line. It was demonstrated that a semicolon could be used between commands, thereby saving you the trouble of having to enter commands on separate command lines.

The semicolon in this instance is a command. Like any command in Unix, the semicolon provides a programmed service. It may seem strange to use a symbol like a semicolon as a command name, but it is certainly more convenient to use it to signify the separation of commands than some word like "end." For example,

```
cd /usr/joe; cat letter
```

versus

```
cd /usr/joe  end  cat letter
```

shows that metacharacters are merely a special group of commands, special in the sense that they are used to connect and/or control the operation of a command line.

Metacharacters are special also in that the instructions or software for metacharacters resides in the shell itself. Table 6-1 provides an overview of some of the metacharacters contained in the shell. As you look over the table, you might notice also a couple of other Unix features, like I/O redirection and the pipe.

Multi-tasking? Redirection of input and output? Pipes? All of these are "big" Unix features? Yes, many of the significant Unix features are merely controlled command line operators using metacharacters. Isn't demystification wonderful?

Okay, what about shell programming?

As you already saw, we can do a little "real-time" programming by using metacharacters in our command lines. The next step is to save the sequence of commands in a file, so that you do not have to re-enter repeatedly the same commands each time you want the computer to perform a job.

One of the differences between building programs by combining commands and few symbolic operators, and that of writing a program in a language such as BASIC or FORTRAN, is the relative simplicity. With shell programming, you can enter a command at your terminal and see something happen; a BASIC program seems abstract because you have to imagine what a series of commands will do.

A shell program, in very simple terms, is one or more command lines stored in a file. When creating shell programs, however, you must take into consideration the fact that you will not be able to intervene to adjust for changing parameters once the program is running. You will have to be careful to include the appropriate metacharacters to contend with this.

This last paragraph is beginning to sound foreboding. In simpler terms, if you create the simplest shell program, a file with a single command in it, there is little that is likely to go wrong with your shell program. However, the more complex your shell program, the greater the possibility for deviations to occur.

COMMAND FILES VS. SHELL SCRIPTS

There are two levels of shell programs. One or more Unix commands contained in a file—simply a file of commands—is referred to as a *command file*. More complex arrangements of commands are called *shell scripts*. There is little difference between a command file and a shell script; both are composed of Unix commands, shell command line operators.

The more complex shell programs begin to compare with programs written in computer languages such as FORTRAN, C, etc. Once you become familiar with Unix utilities and the features of the shell, you will be able to create some formidable programs. If the programs become too formidable, however, you may find that programming languages will be able to do the job better and faster.

RUNNING A SHELL PROGRAM

Programs created with the metacharacters contained in the shell will have the same status as one of the commercial Unix utilities. Commands made with the programs are also run just like a regular Unix command. You enter the name of the

Table 6-1. Shell Program Metacharacters.

Command connectors:

- `;` *Used to separate multiple commands on a single command line.* Waits for execution of preceding command to be completed.
- `&` *Run in background the previous command.* Similar to ; but does not wait for previous command to be completed.
- `:` Pipe. Similar to ; but additionally channels the output of a command into the next command.
- `(..)` Used to group commands together for processing, as you would parts of an algebraic expression.
- `&&` Similar to ; but will only execute the next command if the previous command returns a zero value.
- `||` see &&

Redirect input and output:

- `<` Redirect input to preceding command from a file.
- `>` Redirect output to a file.
- `>>` Redirect and append output to a specified file.
- `<<` Usually associated with a word, eg., << susan. Shell will read input file up to the word in the text following the <<, e.g., susan.

File name substitutions (wild cards)

- `*` Wild card for any and all characters. [1.5]
- `?` Single-character wild card.
- `[..]` Match to the enclosed (specified) character(s) or range of characters, e.g., 1-5 (any and all characters 1 through 5)

Remove special meaning of metacharacters in text

- `".."` Use the enclosed characters literally. Allows you to use the following symbols in a command statement by removing their special meaning: '< . # * ? & : ; ( ) [ ] ½ $ ' also: blank spaces, newlines, tabs
- `'..'` Use the enclosed characters literally. Same as " , but also removes special meaning of $
- `-` Use the next character literally. Removes the special meaning of the following character only, also $ '
- `'..'` Substitute command enclosed within. Allows commands to be intermixed with text statements.
- `{..}` Delineates reserved shell words when embedded in the command line.

Variables substitutions:

- `$(..)` Substitute a variable for the enclosed character string.

file containing the shell command, and the shell executes the program contained within the named file.

The simplest way to execute your new shell program is to run it as an argument to the shell command, as follows:

A more sophisticated method for executing your shell program is to make the file containing the shell program executable, by changing the permission or mode bits on the file. Your command file then may be executed directly, just like a regular Unix command. The shell will execute the commands contained in your shell program file in the order that they are entered on the command line(s).

The simplest shell program could be a single command with no options or arguments saved in a file. This is not very practical because you could just as easily execute the command itself. A more practical shell program would be a listing of commands, separated with semicolons and saved in a file. For example:

cat > sample	Make a file named **sample**.
cd; cat file1; mv file1 file2	New command.
^d	CTRL-d to end the command.

SHELL CONDITIONAL COMMANDS

Until now I have been purposely playing down the complexity of shell programming. I felt that if I came right out with the standard explanations and examples of the more advanced programs you can create with shell commands, you might not have bothered to read this part of the book.

The shell programming language offers a lot of capabilities for performing some advanced-level, imaginative programming. Besides the metacharacters, the shell also contains some reserved words or conditional commands, which can be used to create some very useful programs. Conditional commands provide decision-making facilities for shell programs. If you know a programming language, most of them will be as familiar as old friends; even if you don't, their functions will be apparent. Here are some examples of shell conditional commands:

if	case	until	for	elif	then
fi	in	do	while	done	else

I would like to remind you that command strings, pipes, pipe lines, filters, and the tee command discussed in this chapter also may be entered interactively. That is, their usage is not restricted to inclusion in a shell program.

SUMMARY

The foregoing examples have demonstrated the fundamentals of shell programming and other Unix features you may have read about. At this point the discussion on shell programming becomes more a demonstration on the use of various command sets and metacharacters. Shell programming continues in Chapter 15.

Chapter 7

Tutorial Summary

One proven formula for training is to "tell them what you are going to tell them, tell them, and then tell them what you told them." Now, in conclusion, I will tell you what I have told you.

First we examined the problem: Unix is considered to be a very large and complex computer operating system. Since it had been developed on and for minicomputers, the question has been raised whether Unix can or—maybe more important—*should* be ported over to the new breed of microcomputers.

I do not agree with the opponents of Unix, nor do I feel that they have a justifiable position. Frankly, a number of the anti-Unix-for-the-microcomputer articles I have read are not logical, in my opinion, nor do they even seem knowledgeable of Unix.

Anti-Unix articles have about as much impact as telling my twelve-year-olds that they should not attempt programming sprite graphics on their Commodore 64 computer because this is considered a college- or technical-school-level course.

The state of the art is advancing rapidly. A friend of mine commented that the Commodore 64 has the equivalent RAM of a mini-class computer of 12 years ago, one costing hundreds of thousands of dollars. Twelve years ago 64,000 bytes of RAM was a "real" computer. Today, Data General, Hewlett-Packard, and others offer 500,000 bytes in a machine that I can run on batteries and slip into my briefcase—display and all.

In this day and age, you need all of the encouragement you can get. I cannot see any reason that people should be convinced to remain behind. In some areas of programming my children know more about programming than I do. When some-

one writes that you should avoid Unix because it is too difficult to learn, just think of my twelve-year-old programming sprite graphics.

The biggest problem you will have in learning to use Unix is getting your arms around the subject. The logical structure of the system is elusive. However, as soon as you began to be able to organize the Unix system in your mind, you will have no problem figuring out what you need to learn to get started, and what you can shelve until some later date.

The theme throughout *Unix and Xenix Demystified* has been to show you how Unix is organized, so that you can get on with learning to use those parts of the system that you need to do your job. As I pointed out, for the most part you will be operating your system with application programs, and it really does not matter what operating system you are operating under—except that you should know that you will get better software in Unix-based programs than you will get with software that is based on less capable systems.

This last comment may be difficult for new computer users to understand, as some of the software operating on other systems is already better than some Unix-based programs. However, this is primarily because the Unix system has not (yet) had the large user base to attract companies to develop programs for it.

Until 1984, a software developer writing a program for the IBM PC using DOS could expect to sell twenty to fifty or more thousand copies of a program. By comparison, Unix-based application programs sell maybe a tenth of that number. I expect this will all be turned around with the introduction of the IBM Personal Computer AT using Xenix.

Introduction to Unix Systems

Chapter 1 put Unix into its environmental perspective. It explained that Unix is an operating system, the software that makes the computer run your application programs. It also pointed out that there are many Unix, Unix-based, and Unix-like operating systems on the market. It presented information about the nature of the differences between these systems and the systems within each category.

Most important, Chapter 1 set the stage for the following tutorial chapters by explaining that Unix systems can come in different sizes and shapes, and building the concept that maybe Unix is modular.

We carried this point to the next level, describing the components of the Unix system. This is where I had my difficulty conceptualizing Unix, as most new users do. It seemed as if Unix could be modularized, based on the fact that the utilities, in most cases, are designed to be stand-alone modules. It seemed logical to suspect that the 300-plus utilities should have some commonalities, and that, based on these commonalities, I should be able to organize the utilities into categories.

About the time that I had convinced myself that this must be so, *Understanding UNIX*, by James Groff and Paul Weinberg, was published. There, very nicely laid out in print, was confirmation that the system is modular. After feeling secure in this fact, I then understood the significance of the modular units that computer companies were selling.

I also began to understand that when I buy a Unix system I might not be getting a "full" Unix system—whatever *that* might be. Fortune Computer Company was

one of the first companies to unbundle the system to the extent that they sold the components of the system in something like six or eight packages. Their Unix system program (basic package) cost approximately $800, which at the time was not a bad price. However, when you purchased all of their modules, the price rose to around $2500.

The Structure of the Unix System

By this point in the book you should have had a sound overview of the structure of the Unix system. The next step was to break down the system based on the operation of the components of the system. In Chapter 2 we discussed the breakdown of the Unix operating system into the kernel, the library of system calls and the shell program.

After providing a brief description on the operation of the components of the operating system, we turned to examining the modules of Unix utilities. We presented a table of categories (modular groupings) of utilities and provided some examples of the types of services or functions that each category provided.

The Unix File System

Chapter 3 started to get into the "meat and potatoes" of Unix by explaining how software interfaces with the computer system. We showed you how disk memory is physically divided and addressed, so that the operating system can store and retrieve data and/or computer programs. This led to a discussion on files, types of files, and the fact that everything stored in the computer's memory is stored in a file.

The hierarchical file structure is one of the big features of the Unix system. That is not to say that hierarchical structures are unique to Unix. Quite the contrary. It is just that the authors of Unix could foresee the need for this type of filing and retrieval capability. Remember, Unix was originally designed to run much larger computer systems, which (if I can make another point for Unix) is far better than trying to retrofit the equipment required to enable a pickup truck to carry a ten-ton load.

We discussed building the hierarchical structure to fit your personal needs by making files and directories. We discussed file naming and pathnames, and their role in developing the hierarchical structure. We also pointed out that the Unix system has included a file security system which can be controlled by the individual owner of a file, and that the file owner could set up the file to be shared by other users, or by a selected group of users. They can even create alias names (links to) files.

Commands

Chapter 4 began to put together all of the information presented in earlier chapters. Once you had an understanding of the parts of the system, it was time to begin explaining how to use these parts to operate your computer.

We started our discussion on actually using Unix by describing the shell program, the user interface to Unix. We pointed out that, true to the style of Unix,

there is no single program for operating the system. Several such programs exist, and you can have several users logged on the computer, each using a different shell.

After familiarizing you with the operation of the shell, we next explained how to enter commands. We described a command as simply a computer program that provides a particular service or function stored on the computer's disk, and then we discussed how the shell goes about finding a command in the hierarchical structure.

Entering Unix commands can be a little tricky because there is no real standard command line format. We alluded to the fact that there might be but in reality the "standard" is only coincident with the fact that most of the commands in the system are entered in a similar format. (I am convinced that command line format standards were written after the fact, i.e., after authors started trying to find some commonalities between the entry formats for different commands.)

Be that as it may, we discussed the purpose and usage of the option and the argument on the command line. In general, the option is a command modifier and the argument is the thing upon which the command works. We pointed out that Unix can run several jobs for a single user, just as it can handle jobs from several users. This is the multi-tasking feature of Unix, also referred to as background processing.

Shells and Shell Programming

When I started learning to use Unix, I was very apprehensive about something called shell programming. First off, I did not know if shell programming, shell scripts, command programs, shell procedures—and I am sure I must have read other names—were the same thing or a whole bunch of new things that I would have to face. As I read about them in different books, I felt that they were the same thing, but it took a while before I felt confident in this assumption.

I also was apprehensive about having to learn to write them. It seemed that they could be useful, but. . . . The answer is they can definitely be useful, but certainly are not necessary to the operation of your computer. I have only included an introduction to the subject in order to demystify shell programming. To shell program, you only need to know how to connect commands with command line operators, how to put these command line(s) into a file, and how to set up the file so that it is executable.

I can remember the thrill of discovering that a file can be made executable simply by changing the permission bits. It came about when I was trying to use a command named **wall**. The **wall** command can be used to write a message to all of the users logged on the computer. It seemed as if I could use this command to write pre-prepared screens on my students' terminals by simply making a command continuing the instructions to display each screen.

I managed to figure out most of the problem, but I could not get my little program to execute like a command. I finally asked one of the Unix gurus, who explained how the **chmod** command is used to change the permission bits. Simple!

Shell programming is easy at the basic level, but when did a programmer ever let something remain simple? The shell program, as you have read, can provide some very extensive programming capabilities. I find the logic involved in using

commands to create programs easy for me to understand and work with. I much prefer this type of programming to using FORTRAN or BASIC.

Finally, there were those fearsome terms: tee, pipe, and filter. When I first came across them I wondered if I could learn to use them. They are useful, but again definitely not necessary to your learning to use Unix.

If I had to put down those things I felt were important to your learning to use Unix in an order of importance, I would create the following list:

1) Read *Unix and Xenix Demystified* in order to gain an understanding of the operation and function of the Unix system.
2) Learn to use the shell to enter commands.
3) Learn how to make directories and files, and how to maintain and locate files in the hierarchical structure.
4) Learn to use the application programs for which you purchased your computer.
5) Beyond this point, you have to have a need to get into the Unix system. You may have a problem, such as needing to perform some special system administration functions not available on your menu system, communicate with another Unix computer, or develop software. Maybe you just want to play some games.

Now it is time to begin using Unix. The following eight lab exercise sessions will provide some working examples you can use to become familiar with the Unix system first-hand. If you do not have access to a computer, you will find that each screen for each example has been included as it would appear on your terminal.

Part 2
Hands-On Unix

Chapter 8

Introduction to Lab Exercises

The objective of the following lab sessions is to culminate our discussion on Unix with examples of the use of Unix commands and to demonstrate the operation of some of the principal features of the Unix system. In the first seven chapters of this book you learned about:

- The Unix hierarchical file structure.
- The constituent parts of a Unix system.
- Interfacing Unix with the computer system.
- An overview on using the Unix system.

This *tutorial section* provided an *academic overview* of the operation of the Unix system and your computer. These chapters were comparable to the classroom part of a driver training course. Now it is time for you to get on the stool and get some first-hand experience behind the keyboard.

The lab exercise sessions present some everyday-type operations you might encounter while using Unix. The intent in these sessions is to provide some practical exercises in which you will learn to use some of the more common Unix utilities. The examples are also designed to help you to gain a feeling of confidence in your ability to learn to use these commands, as well as prepare you for learning to use other commands not presented in *Unix and Xenix Demystified.*

AN OVERVIEW

The exercises presented in the next eight chapters of *Unix and Xenix Demystified* will help you to learn how to:

- **Log On and Exit the Unix System (Chapter 9).** The first thing you will learn is how to gain access to the Unix system and sign off when finished. This session also includes a discussion of the Unix system's security features and how to enter your password to keep others from gaining access to your files.
- **Access Files (Chapter 10).** We will discuss using the information contained in directories to traverse the hierarchical file structure and to locate files.
- **Make Files and Directories (Chapter 11).** Some of the commonly used Unix commands will be presented to show you how to make directories and files in the hierarchical file structure, move them to different locations in the structure, make copies of them, display their contents on your terminal monitor, and print them out on the printer.
- **Modify File Security (Chapter 12).** We will discuss setting up file security for each of your files to keep other users from gaining access to your files.
- **Link Files (Chapter 13).** We will provide some practical examples of making alias names to existing files and/or commands.
- **Piping (Chapter 14).** We will present a brief introduction to the Unix pipe, which can be used to connect Unix commands in series to form a minicomputer program or routine.
- **Shell Programming (Chapter 15).** Chapter 6 described shell programming. In this lab session we will provide some working examples for using metacharacters and the **tee** command.
- **Remove Files and Directories (Chapter 16).** Obviously, after you clutter your system with the files you will create in these sessions, you will want to know how to get rid of them. The end of session 8 will lead you through the removal of everything that was created in the exercises.

There are slightly more than a dozen commands presented in these lab exercise sessions. The commands are used in multiple applications in order to demonstrate the versatility and power of the Unix system, as well as the fact that you do not need to know very many commands in order to learn to use Unix. I believe that you will be extremely impressed with the capabilities of the Unix system as you proceed with the lab exercise sessions.

These lab exercises were developed with the assistance of Mr. Jack Khaw. Mr. Khaw has many years of experience teaching Unix and computer languages at colleges, in industry, and at the Computer Training Centers in California. These exercises represent the hands-on training presented in our "Beginning Unix" class.

If you have a computer at your disposal it will be helpful to follow the exercises by entering the commands at your terminal keyboard. However, the graphics provided in conjunction with the exercises include every screen you would see on your terminal monitor. A computer is desirable, but not necessary.

SOME THINGS YOU SHOULD KNOW

Before starting the following lab exercise sessions, you should be aware of the

operation of the shell program and some standard operating instructions. With respect to its function in these sessions, the shell acts as an interface between you and the computer, telling you when to enter commands, interpreting your commands for the operating system, and making sure that the operating system processes them.

The Shell Prompt

The shell will indicate when it is ready to receive your command by displaying a prompt. Depending on the shell currently in operation on your system, the prompt could be:

$ AT&T Unix shell user prompt
% Berkeley Unix shell prompt
# "Super-user"

After logging on to Unix, you should have a **$** or **%** prompt. If you are operating under a menu-type shell (common on many microcomputer systems), you will need to exit from the menu to one of the system shells, generally by entering the character.

Command Interpretation

Commands are requests to the shell to make the computer perform a task. To the computer, commands are merely a series of characters. They have no meaning to the computer until the shell interprets the meaning to be a command, a text file, etc. This is why the shell is described as the system interface between you and the computer or the command interpreter.

Actually, the shell does not really interpret anything. What the shell does is try to find the set of characters in the form of a file name, one that already exists in the hierarchical file system, to match those you have entered at the terminal keyboard. When the shell finds a match, it stops at that file, reads the computer program or other data contained in the file, and performs the instructions that are contained therein.

Entering a Command

Correcting an Incorrectly Entered Command. If you make an error while entering a command, one of the following should eliminate or nullify the incorrectly entered character. Try them in the following order:

```
Backspace
^a (CTRL-a)
#
@
```

Most systems will eliminate incorrect characters from the terminal monitor display when these commands are entered at the terminal keyboard; others will not. Your system administrator or computer manual will be able to give you precise instructions regarding the operation of your system and which command should be used.

Acknowledgement of an Entered Command. Entering a command at the terminal keyboard sets into motion a complex series of events inside the computer.

You (fortunately) will not be aware of what is happening; you will only be aware of whether your command was or was not processed. In general, if the shell redisplays a prompt, e.g., the **$** or **%** prompts, chances are the command was processed.

If the command you enter requires that something be displayed on the terminal monitor, and it is displayed, you will obviously know that the command was processed.

If the shell was not able to process your command, because you spelled it wrong, forgot to enter a required part of the command line, entered the command line format incorrectly, or if the command you entered was not available on your system, etc., the shell may provide you a clue as to what you did incorrectly or why the command was not processed. Then again, it may not, depending on your specific system's operating characteristics.

The most common error is a misspelled command, which would be the same as a command that does not exist on your system. The shell may respond with:

not found or **bad directory**

If you enter a command line incorrectly, most shells will respond with a command line format statement, such as the one shown in Fig. 8-1. This is the response for an improperly entered **-p** option for the list contents of a directory (**ls**) command on the Xenix system. There is no **-p** option, as you can see in the error listing.

Command Line Formats. Command, command line formats, and command options for each of the commands used in these lab sessions are discussed in detail in Appendix A.

Halting a Command in Process

Most commands can be stopped or terminated by pressing one of the following keys on the terminal keyboard:

DEL	(Delete)
RUB	(Rubout)
ESC	(Escape)
Break	(Break)

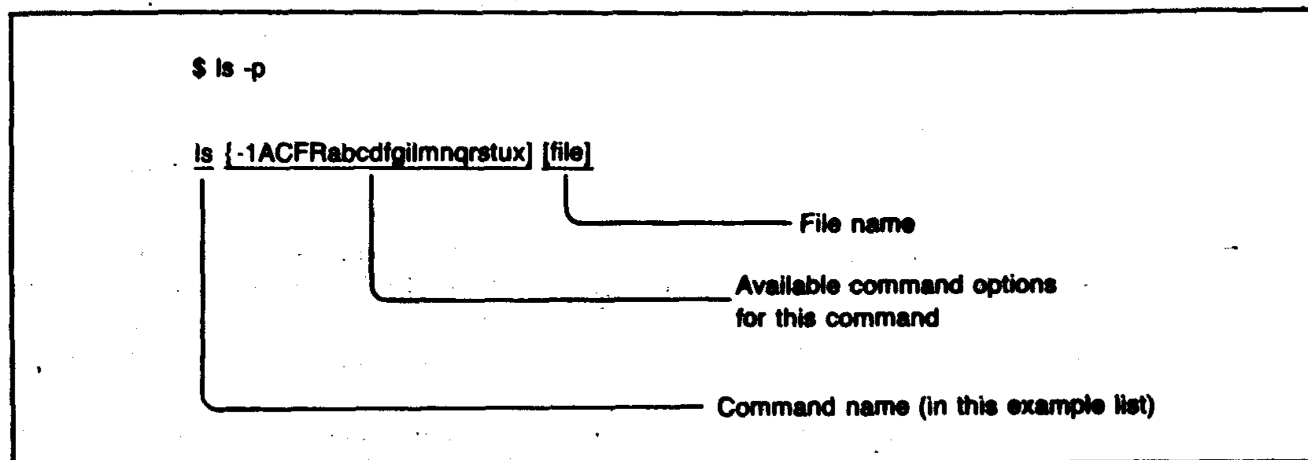


Fig. 8-1. Most shells provide helpful rather than cryptic error messages.

Control Keys

During the lab exercise sessions, we will use the functions CTRL-d, CTRL-s, CTRL-q, and others. Instead of adding more keys to the keyboard for the functions that these keys would perform, computer designers have opted to change the meaning of the existing keys by adding a control key.

The control key changes the meaning of any of the keys on the keyboard, just like the shift key can change the meaning of a character from lowercase to uppercase when you simultaneously press the shift and character keys. In fact, the control key probably is best thought of as a special kind of shift key.

A control key, in this and most other textbooks, is designated by CTRL and a character key name, or by preceding the character key with a ^ symbol, i.e., ^d. The control keys we will use in these sessions will be:

- ^d To exit from or log off the Unix system, and also to designate the end text entry routines.
- ^s To halt the display of data on the terminal monitor.
- ^q To resume the display of data on the terminal monitor.
- ^h Delete an improperly entered character.

When you enter a control key command, it will not be displayed on your terminal screen. Some routines, such as entering text, are ended with CTRL-d. In these cases, the shell will redisplay a shell prompt to acknowledge the command.

Graphic Aids

Throughout the exercises in the lab sessions, we will include graphic aids. The graphic aids will provide you with a model of the operations we are performing with the Unix commands. I believe you will find these models invaluable in understanding what is going on in the computer.

Chapter 9

Login and System-Level Security

The login name and password provide system-level security protection, keeping unauthorized individuals from gaining access to the system. Login names and passwords are used by Unix systems to identify, uniquely, individuals who are authorized to have access to the system.

When you enter a login name, Unix tries to match the login name that you enter with the names that have been registered with the system. If a match is successful, Unix will permit you to proceed to the next step, which is to enter your password. As with the login name, Unix tries to match the password you enter with a password that has been registered with the system in conjunction with the login name used.

Login names are entered into the Unix system's files by the system administrator or other person responsible for operating the computer. Many microcomputers have a user administration program added to their Unix system library of utilities to help nontechnical users and computer owners register new users on the system.

Passwords are entered by the individual users after they log on for the first time. If you should forget your password, the system administrator can clear your password, which will allow you to log on and enter another password. If you have not already been registered to have access to use the system at this point, I would suggest that you contact your system administrator or read your computer operator's manual and register yourself with the system.

In this session we will introduce:

login	How to gain access to the Unix system.
passwd	Setting a password (optional) to increase security measures against an individual gaining access to your files.

LOGIN NAME

A *login name* is any series of characters, usually a user's name, that the user chooses to use to identify himself or herself to the Unix system. The login name rules observed by most Unix systems are:

- Any series of letters and/or numbers is legal.
- Special characters such as /,.-(), etc., should be avoided.
- No spaces are allowed in the name.
- The first character should be a lowercase letter.
- Lowercase letters are usually used throughout a login name for convenient entry
- Upper- and lowercase letters are unique characters.
- The name must be different from all other login names.
- Though the name usually can be of unlimited length, most systems will only acknowledge the first eight characters.

Some computer systems, particularly those with terminals at remote locations, require that you go through a sign-on routine before you get to the point of logging on to the Unix system. Microcomputers require a special start-up routine (called *booting the system*) before you can log on.

When the system is ready for you to log on, it will display a login prompt on your terminal monitor. Refer to Fig 9-1.

- Screen A contains an example of a typical login prompt.
- Screen B is an example of an entered login name.
- Screen C displays the shell $ prompt. This indicates that you have gained access to the Unix system and the shell is ready to accept your command.

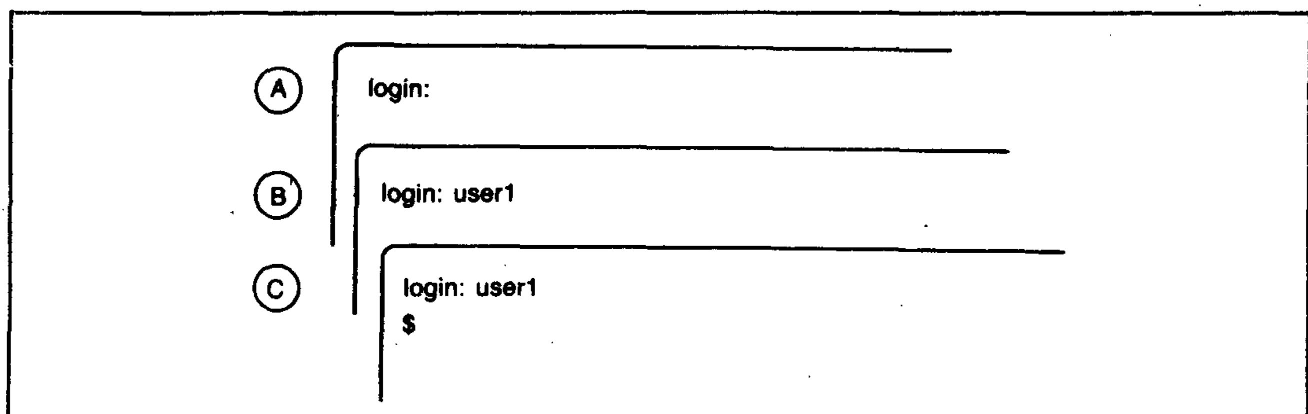


Fig. 9-1. Loggin on a Unix system.

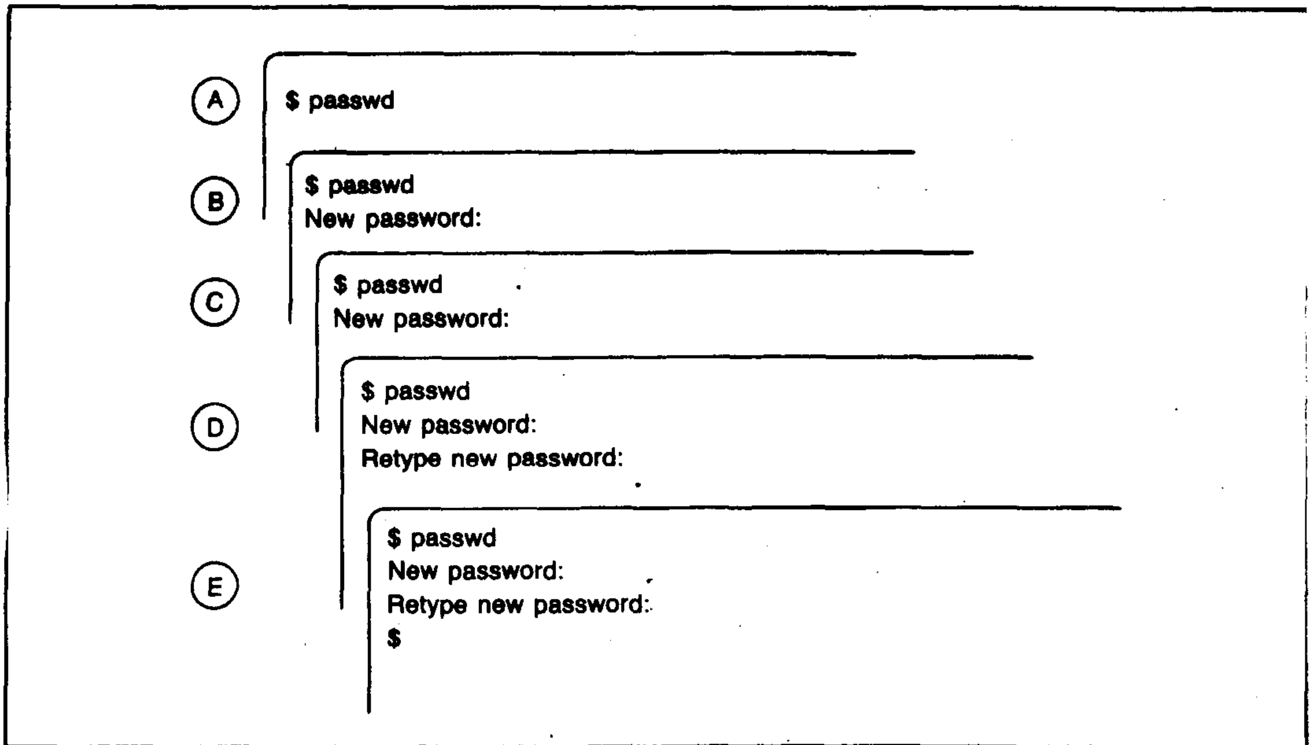


Fig. 9-2. Using the passwd command to establish or change your password.

PASSWORD

A password, like a login name, is a series of characters. The conventions observed for passwords for most Unix systems are:

- It may be any series of characters including upper- and lowercase letters, and numbers.
- Some special characters are allowed (e.g., / $ +, etc.), but avoid using them because they may be metacharacters used in the shell as commands.
- A minimum of 5 characters and a maximum of 13 are required for the password.

The first time you log on, a password prompt should not appear unless the system administrator has set a password at your request. You set the password with the **passwd** command. Our command sequence continues in Figure 9-2.

- Screen A shows the **passwd** command entered, which will start the program (utility) for setting up your password.
- Screen B displays a typical prompt in the password program, requesting you to enter your password.
- Screen C shows the results after entering your password. The password is not displayed when you enter it. This is a security feature that keeps others from learning your password by watching you enter it at the terminal keyboard.

- Screen D shows the prompt requesting you to enter your password again. Entering your password twice gives some assurance that you have entered the desired password.
- Screen E shows that the password is not displayed but the shell prompt is. This indicates that your password has been accepted.

The next time you log on, the system will display a password prompt after you enter a correct user name (Figure 9-3).

- Screen A shows the password prompt after the login name is accepted.
- Screen B is an example of an entered password. You will note that here again the password was not displayed when you entered it.
- Screen C indicates that Unix accepted your password and login name, displaying for you a shell prompt such as **$** or **%**.

LOGGED-ON ACKNOWLEDGMENT

As soon as your login name and password are accepted by Unix, the shell will acknowledge that you have gained access to the system with a **$** or **%** shell prompt (Figures 9-1C and 9-3C).

Prompts are used to inform you that the shell is ready for you to enter a command. (Sometimes a system message will be displayed.) The **$** prompt, used by most AT&T Unix systems, is also called the shell prompt, user prompt, command line prompt, primary prompt string, and other names. Berkeley Unix systems use the **%** as the shell prompt. Any symbols or words may be used, but we will not go into a discussion of the details for changing the prompt.

If you have accessed the system as a *super-user*, the system will display a **#** symbol, called the *super-user prompt*. This level of access to the system is reserved for the system administrator. When operating the system as a super-user, you can override any file securities and/or modify the system itself. It is also possible to wipe out inadvertently or otherwise damage the Unix system.

(A)
```
login: user1
password:
```

(B)
```
login: user1
password:
```

(C)
```
login: user1
password:
$
```

. 9-3. Logging on a Unix system with a password.

LOG OFF

The typical log-off procedures for many microcomputers using the Unix system is simply a ^ d (CTRL-d). You enter a CTRL-d by pressing simultaneously the control and d keys, similar to pressing simultaneously the shift and letter keys to enter an uppercase letter.

The system will acknowledge that you are logged off by displaying a login prompt on your terminal monitor. If this does not work on your system, contact your system administrator or read your computer user's manual.

Chapter 10

Locating Directories and Files

Now that you have gained access to the Unix system, the first thing you need to learn is how to locate directories and files in the hierarchical file structure or, as many users describe this, learn to "move about" in the structure. In this session we will introduce the following commands:

who am i	To determine the login name.
pwd	To determine the pathname of current or working directory.
cd	To change the working directory.
ls	To display a list of the subdirectories and files in a directory.

WHO AM I

To start this exercise, we will determine who I am logged on the system as, that is, the user name under which I have logged on the system.

The format for the **who am i** command is:

who am i (**am i** is an option to the command)

Figure 10-1 traces the use of the **who am i** command

- Screen A shows the shell **$** prompt, which means that the shell is ready to receive a command.
- Screen B shows the entered **who am i** command, requesting the shell to display the name of the logged-on user at this terminal.
- Screen C shows we are logged on the system as **user1**.

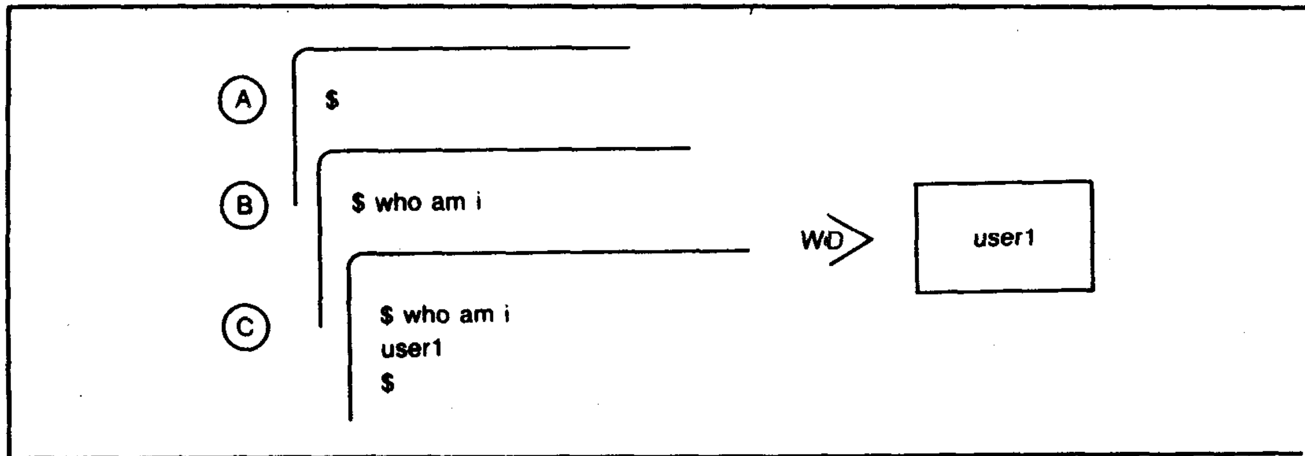


Fig. 10-1. Using the who am i command. The current working directory is shown at right.

The who am i command is useful when you want to determine who is working at an unattended terminal. It is also useful in determining what name you may have used to log on to the system when you have more than one registered user name and you have forgotten which name you used to log on.

PRINT WORKING DIRECTORY

The who am i command gives us a little information about user1, but it does not indicate the relative location of user1 in the hierarchical file structure. That is, it does not provide a pathname or indicate our current relative location in the file structure.

To obtain the pathname of our current relative location in the hierarchical file structure, we use the command pwd to instruct the shell to display the directory in which we are currently working or (in Unix terms) to "print the working directory."

The format for the pwd command is:

pwd *(no options)*

Figure 10-2 demonstrates the pwd command.

- Screen A shows the pwd command entered to determine the name of the current directory.
- Screen B displays the pathname of the current directory as /usr/user1.

Whenever you log on the system, Unix will always place you in the directory that you were assigned as part of the registration procedure. This is called your "home" directory.

We know that the top of the structure is the root directory, and the pathname of the working or current directory is /usr/user1. Thus, we can construct the start of a model of the hierarchical file structure of our system.

(A quick refresher on pathnames: All pathnames begin with a /, which is the symbol for the master directory, or in Unix, the root directory. Thereafter, the

names of all the directories along the path to a current or working directory are separated by slashes.)

The **WD>** in the graphic model indicates or points to the current relative location in the hierarchical file structure. In Unix vernacular, this is the current or working directory.

The **pwd** command can be used at any time to determine your relative location in the hierarchical file structure. However, you should understand that there is no physical hierarchical file structure, per se. That is, the structure that we use to demonstrate graphically the relative locations of directories does not physically exist in the computer. The hierarchical structure is only a conceptual diagram drafted to help you envision the relationships between files in the system. The pointers we use to show you our "location" are only a graphic aid to explain what we are doing with the commands in these exercises; they are a convenient way of discussing the changing of working directories.

CHANGE WORKING DIRECTORY

We use the **cd** command to move through the hierarchical file structure, or (in more precise terms) make another directory the working directory. The format for the **cd** command is:

cd directory name

Use of the **cd** command is demonstrated in Fig. 10-3.

- Screen A shows the **cd /** command entered, directing the shell to make the directory / the working directory. The root directory name, by convention, is always symbolicly entered as /.
- Screen B shows the shell **$** prompt again, indicating that the shell is ready for you to enter your next command.

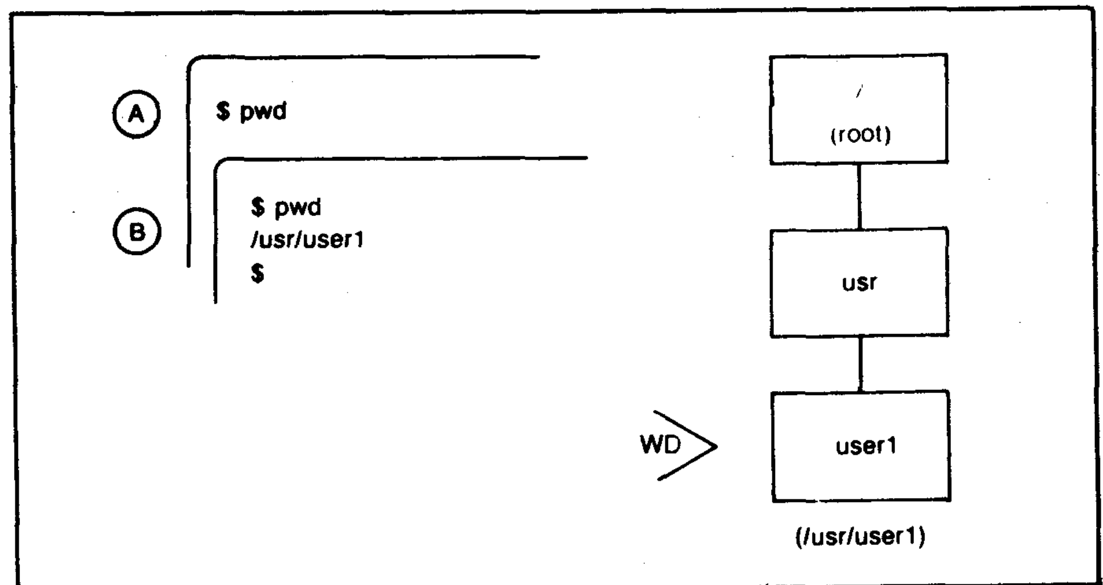


Fig. 10-2. The "print working directory" command sequence, illustrating its location in the hierarchical structure.

The **cd** command does not display any positive indication that it has completed its task, other than the display of the shell prompt. If the requested directory name had been the name of a file instead of a directory, the shell would have indicated with a diagnostic statement (such as **bad directory**) that it could not move to this directory.

To determine whether the shell changed the working directory, we will need to use the **pwd** command, the format for which is simply

pwd

Figure 10-3 continues.

- Screen C shows the **pwd** command entered to determine if the **cd** command changed the working directory to the root (/) directory.
- Screen D indicates the pathname of the working directory is /.

The **WD>** indicates the location of the current or working directory in the hierarchical file structure.

Now, for some practice and further familiarization with the **cd** and **pwd** commands, let's move back to our home directory (Fig. 10-4).

- Screen A shows the **cd usr/user1** command entered to change the working directory to our home directory.

You may have questioned why we entered the directory name part of the command line without the initial / for the root directory. The answer is we used the relative pathname and not the full pathname. The full pathname is **/usr/user1**, but

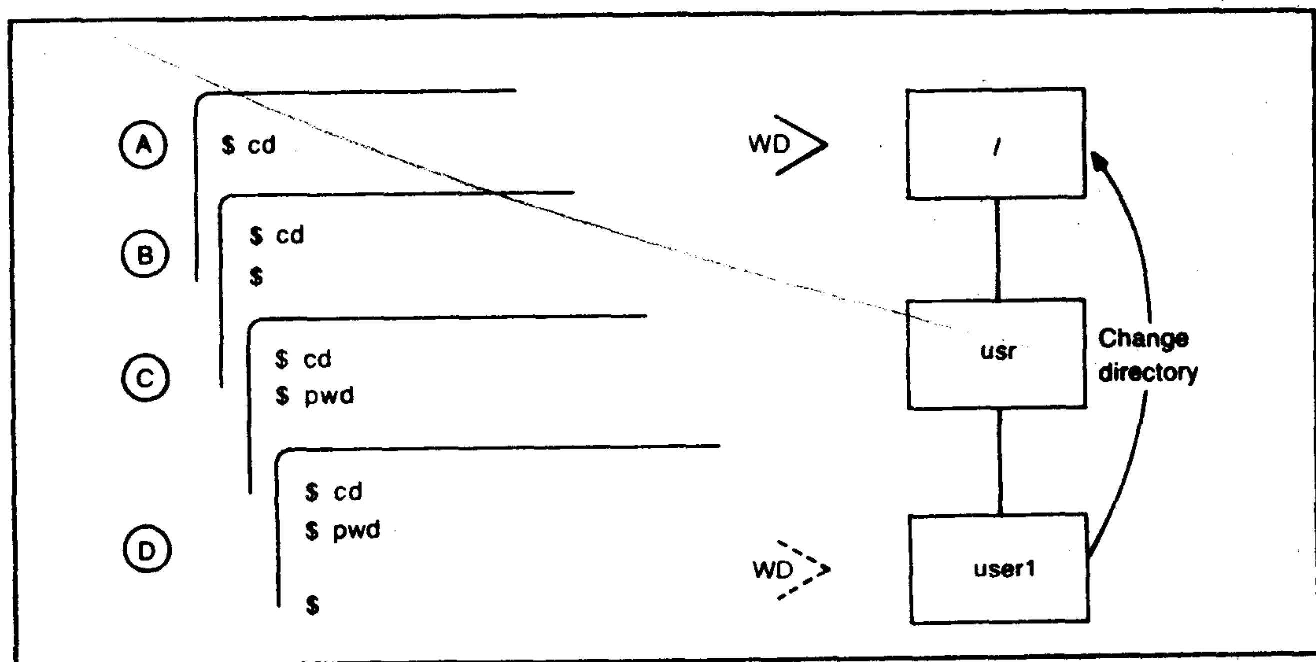


Fig. 10-3. Changing the working directory from user1 to root.

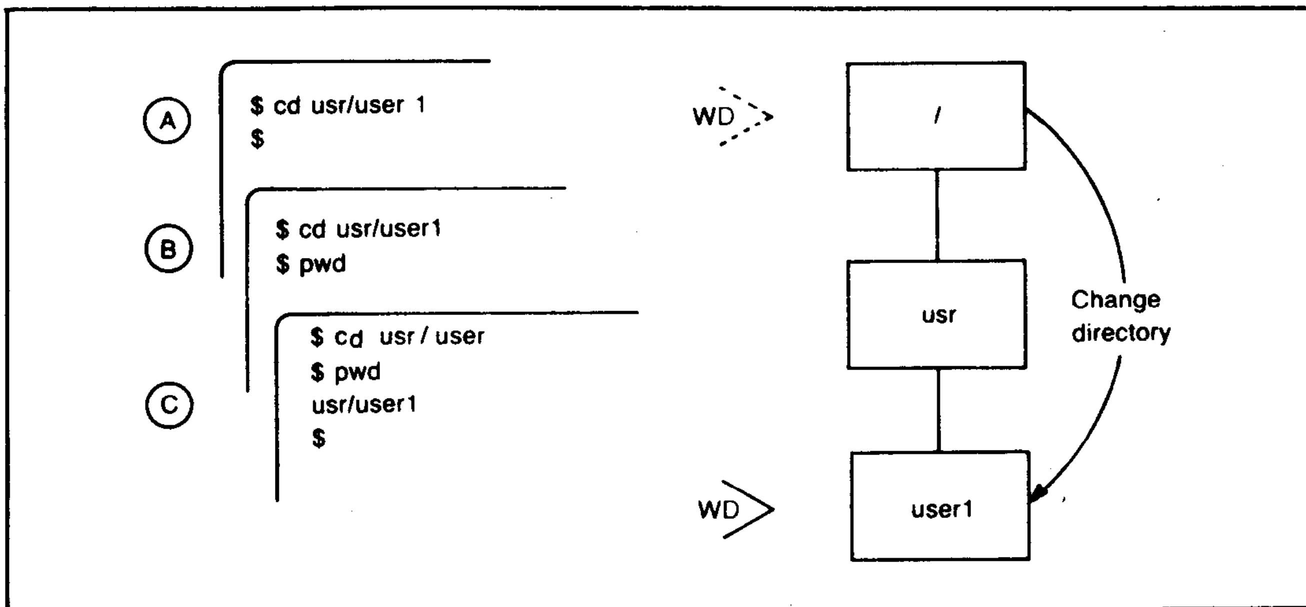


Fig. 10-4. Returning the user1 as the working directory.

we were already at the directory /. Entering the / would have been redundant, but not incorrect.

Once again, let's check to see if the current working directory is **/usr/user1**.

- Screen B shows the **pwd** command entered to determine the name of the current directory.
- Screen C shows the pathname **/usr/user1** .

If you should get lost while wandering through the hierarchical file structure, just enter the **cd** command without a directory name. The shell will automatically make your home directory the working directory. Try it! Move to / and then enter just the **cd** command.

LIST DIRECTORY CONTENTS

So far, you were able to move to the root directory and your home directory, because you knew that these directories existed. The next step is to find some other directories in the hierarchical structure to which we can move.

To ascertain the names of other directories, we can use the **ls** command. This command will list the names of the directories and files contained in a directory. We are currently in our home directory, so let's list the names of the directories and files here first.

The format for the **ls** command is:

ls command options name of a directory (optional)

Figure 10-5 shows the command sequence.

- Screen A shows the **ls** command entered to list the contents of the directory **user1** .

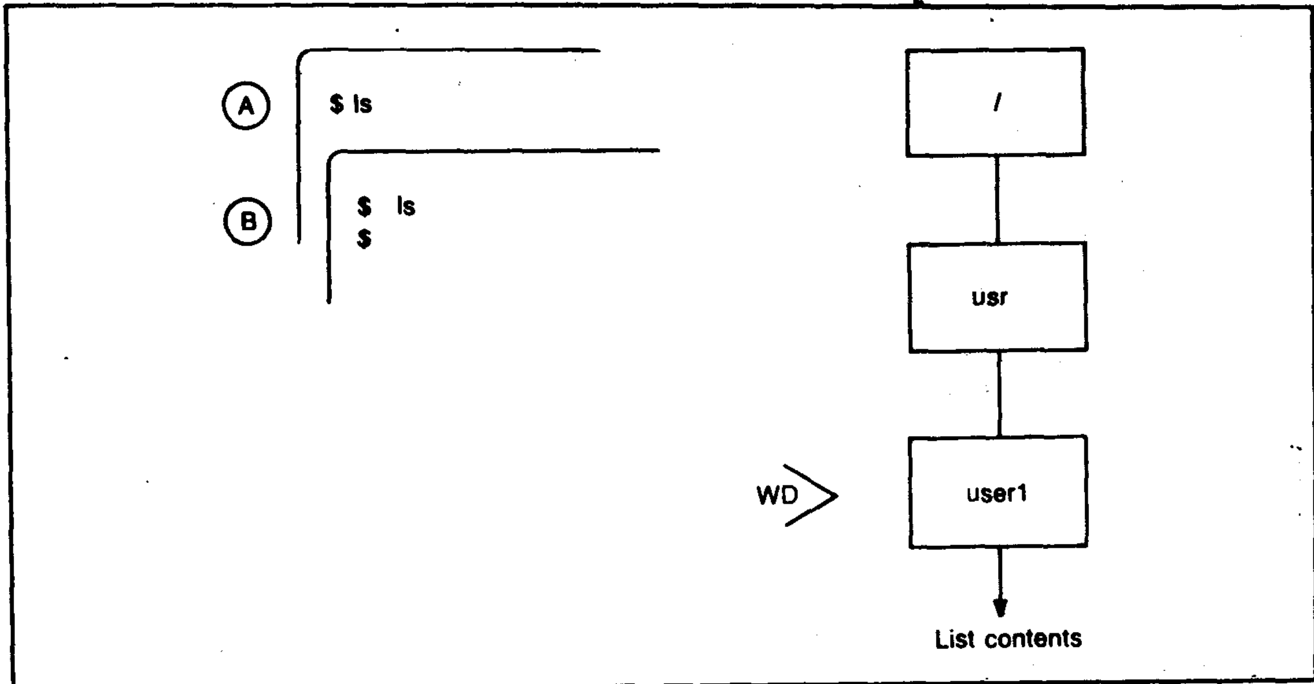


Fig. 10-5. The ls command displays the contents of the working directory.

- Screen B shows nothing?!

This indicates that there are no directories or files in our home directory. Keep this in mind, because in the next session we will be making some directories and files in our home directory. When we use the **ls** command later, you will be able to see the change. In the meantime, however, we can move to the / directory and try the **ls** command there (Fig. 10-6).

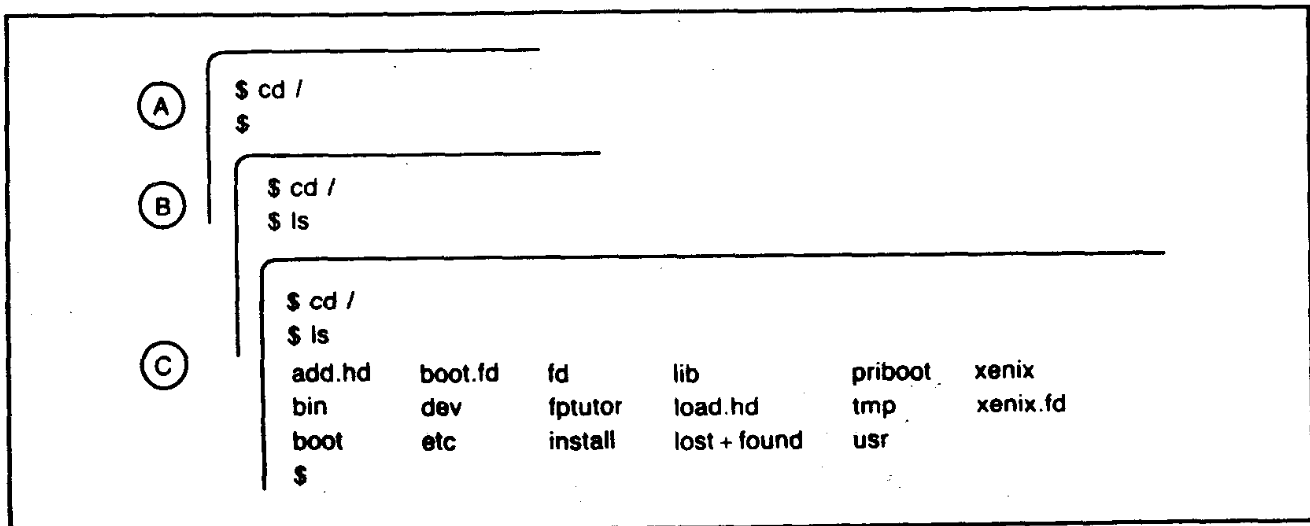


Fig. 10-6. Examining the contents of the root directory.

- Screen A shows the **cd** / command entered, to make the / (root) directory the working directory.
- Screen B shows the **ls** command entered to list the contents of the / directory.
- Screen C displays a list of the directories and files in the / directory.

This attempt to find some directories and files was a little more successful. Figure 10-6C indicates that there are 17 directories and files in my / directory. However, the display does not help us to determine if the named items are directories or files.

It just so happens that there is a remedy for this problem! The ls command has a -l option, which will direct the shell to list the contents of the / directory with details about the characteristics of the listed items (Fig. 10-7).

- Screen A shows the ls -l command entered. (The current directory is still /.)
- Screen B shows a listing of the directory and file names with some additional details about each of the files and directories in the directory /.

Ⓐ

```
$ ls -l
```

Ⓑ

```
$ ls -l
total 363
-rwxrwxr-x    1   root    2505  Apr    6   16:42  add.hd
drwxr-xr-x    2   bin     2544  Oct   12   18:44  bin
-rwxrwxr-x    1   root   11040  Apr   17   15:23  boot
-rwxrwxr-x    1   root   10448  Apr   17   15:23  boot.fd
drwxrwxrwx    3   root     864  May   29   21:43  dev
drwxr-xr-x    4   bin      768  Oct   12   17:45  etc
drwxrwxrwx    2   root      48  Jul   28   11:21  fd
drwxrwxrwx    2   root      32  May   25   22:52  fptutor
-rwxr-xr-x    1   root     287  Feb    1   00:00  install
drwxrwxrwx    2   root     320  Jul   28   11:22  lib
-rwxrwxr-x    1   root     709  Apr   17   15:23  load.hd
drwxrwxrwx    2   root     832  Apr   17   15:30  lost+foun-
-rw-r- -r- -  1   root    2656  Apr   17   15:23  priboot
drwxrwxrwx    2   root     656  Oct   25   16:54  tmp
drwxr-xr-x   26   bin      416  Oct   12   17:44  usr
-rwxrwxr-x    1   root   73408  Apr   17   15:23  xenix
-rwxrwxr-x    1   root   73168  Apr   17   15:23  xenix.fd
$
```

Fig. 10-7. The -l (long) option provides more information about files.

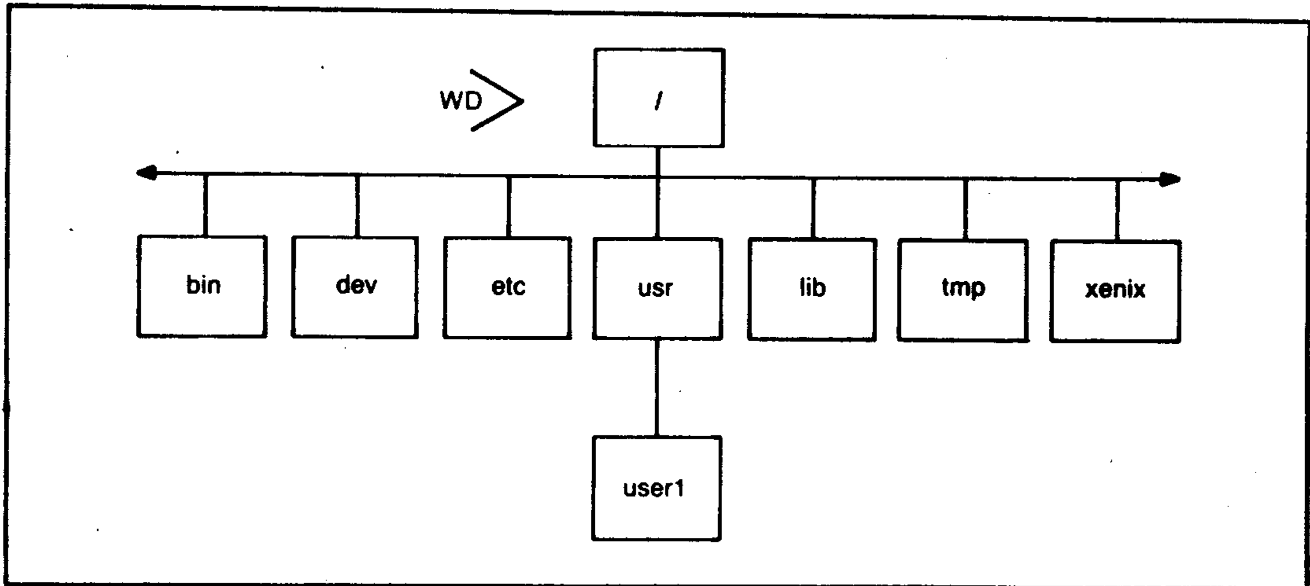


Fig. 10-8. The expanded hierarchical structure.

This listing is called the *long format*. It will be explained in further detail in Appendix A, but in the meantime we need to determine how to tell which items are directories and which are files.

If you look at the first column on each line, you will see a **d** or a **-**. If the line begins with a **d**, the line is a directory and if it begins with **-** the line is a file. The last word on the line is the name of the directory or file. (A note about entering command options: All options are preceded by a **-**. You can enter several options to a command on the same command line as long as they do not try to make the shell perform incompatible tasks, such as listing the items in alphabetic order and at the same time requesting that it should list the items in reverse alphabetic order.)

An expanded diagram of our hierarchical structure (Fig. 10-8) has been expanded to include some of the new directories from the listing in Fig. 10-7B.

Our next step will be to choose one of the directories listed in Screen B and use the **cd** command to move to this directory. One of the directories that is bound to contain number of directories and files is the **/bin** directory. This is the directory where most of the utilities (commands) are located. When we use the **ls** command to list the contents of the **/bin** directory (Fig. 10-9), you will have a chance to see some of the commands that we are using in these lab exercise sessions.

- Screen A shows the **cd bin** command entered to make the directory **/bin** the working directory.
- Screen B shows the **ls -l** command entered directing the shell to display the contents of the **bin** directory.
- Screen C is a partial display of the contents of the **bin** directory.

The **ls -l** command will list the contents of a directory, but it certainly is not programmed to make long lists readable. Try entering the **ls -l** command again, but this time when the shell starts listing the contents of **bin**, enter a **^s** (press the control and **s** keys on your terminal keyboard simultaneously).

A

```
$ cd bin
$
```

B

```
$ cd bin
$ ls -l
```

C

```
$ cd bin
$ ls -l
total 3655
-rwx- - x- -x   1  bin     11896  May  4  11:45  ac
-rwx- -x- -x    1  bin     31796  Feb  1  00:00  adb
-rwx- -x- -x    1  bin     10816  Feb  1  00:00  ar
-rwx- -x- -x    1  bin      4484  Feb  1  00:00  arcv
-rwx- -x- -x    1  bin     44356  Feb  1  00:00  as
-rwx- -x- -x    1  bin      9056  Feb  1  00:00  at
-rwx- -x- -x    1  bin     50288  Feb  1  00:00  awk
-rwx- -x- -x    1  bin      2020  Feb  1  00:00  ba
-rwx- -x- -x    1  bin     14244  Feb  1  00:00
-rwx- -x- -x    1  bin       612  Oct  1  00:0
-rwx- -x- -x    1  bin      4686  Feb  1         tr
-rwxr-xr-x      1  bin       329  Feb            troff
-rwx- -x- -x    1  bin      4830  Oct     00:00  true
-rwx- -x- -x    1  bin      6414          00:00  tsort
-rwx- -x- -x    1  bin       830       1  00:00  tty
-rwx- -x- -x    1  bin            eb   1  00:00  uniq
-rwx- -x- -x    1  bin         4  Feb  1  00:00  units
-rwx- -x- -x    1  root     1246  Feb  1  00:00  uucp
-rwx- -x- -x    1  bin     12092  Feb  1  00:00  uulog
-rwx- -x- -x    1  b       20698  Feb  1  00:00  uux
-rwx- -x- -x    1           6944  Feb  1  00:00  v7grep
-rwx- -x- -x       oot     10452  Feb  1  00:00  v7login
-rwx- -y           bin      8622  Feb  1  00:00  v7ls
-rwx            1  bin      8324  Feb  1  00:00  v7ps
     x          3  bin     67954  Feb  1  00:00  vi
   -x- -x       1  bin     13662  Feb  1  00:00  vplot
rwx- -x- -x     1  bin      8600  Feb  1  00:00  vpr
-rwx- -x- -x    1  bin      7228  Oct  1  00:00  who
-rwx- -x- -x    1  bin      6628  Feb  1  00:00  write
-rwx- -x- -x    1  bin     19586  Feb  1  00:00  xget
-rwx- -x- -x    1  bin     20196  Feb  1  00:00  xsend
-rwx- -x- -x    1  bin     25496  Feb  1  00:00  yacc
-rwx- -x- -x    1  bin      3530  Feb  1  00:00  yes
rwx- -x- -x     1  bin      8600  Feb  1  00:00  vpr
-rwx- -x- -     1  bin      7228  Oct  1  00:00  who
-rwx- -x- -x    1  bin      6628  Feb  1  00:00  write
-rwx- -x- -x    1  bin     19586  Feb  1  00:00  xget
-rwx- -x- -x    1  bin     20196  Feb  1  00:00  xsend
-rwx- -x- -x    1  bin     24496  Feb  1  00:00  yacc
-rwx- -x- -x    1  bin      3530  Feb  1  00:00  yes
$
```

Fig. 10-9. Examining the contents of the bin directory.

The ^s will halt the display on the terminal monitor. To resume displaying the listing of the bin directory, enter a ^q. The shell will begin listing the contents of the directory again. To halt the display again, enter a ^s. The ^s and ^q commands also can be used in conjunction with other commands that display data on the terminal monitor faster than you can read it.

(In a later session we will discuss a better way to stop the listing of a file, by piping the output to another command called more that has been programmed for this purpose.)

We will use the cd command again to change the working directory to the /usr directory, so that we can find our home directory in a listing of the contents of the directory usr (Fig. 10-10). This will help you to prove whether the diagram of the hierarchical structure is correct.

- Screen A shows the cd /usr command entered to make /usr the working directory.

This time we had to include the / for the root directory in order to move to the /usr directory. The rule is this: If you are not in the directory in which you wish to make a working directory, you must enter the full pathname of the new directory. That is, you may use relative pathnames only when you are in a "leg" of the hierarchical structure and "above" the directory which you wish to make the working directory.

Now let's use the ls -l command again to list the directories and files in the directory /usr.

- Screen B shows the ls -l command entered.
- Screen C shows a partial listing of the directories and files in the directory /usr.

You might note that all of the user files listed are directories, as denoted by the d in the first column of each line. You can find our user1 directory listed among the list of file names.

Now let's make our home directory the working directory. We can do this in one of two ways. We can simply enter a cd with no directory name, which will automatically move us to our home directory from any directory in the hierarchical structure, or we can enter a pathname (relative or full). The full pathname of our home directory is /usr/user1. We already are in the /usr directory, however, so our relative pathname is simply user1 (Fig. 10-11).

- Screen A shows the cd command entered to make our home directory the working directory.

The ls command can be used without a directory name, as we have already exhibited, or it can be used in conjunction with a directory name. Therefore you do not have to be located in the directory of the directory whose contents you wish to review. Let's try listing the contents of / from /usr.

- Screen B shows the **pwd** command entered.
- Screen C indicates that the current directory is **/usr**
- Screen D shows the **ls -l /** command entered to list the contents of the directory named **/**.
- Screen E lists the contents of the **/** directory.

Screen E should be compared with Fig. 10-7, Screen B, in order to prove that we actually have been able to list the contents of any directory from another directory.

(A)
```
$ cd / usr
$
```

(B)
```
$ cd / usr
$ ls -l
```

(C)
```
$ cd /usr
$ ls -l
total 28
drwxrwxrwx 2   bin      128  Jul  28  11:22  adm
drwxrwxrwx 2   altos     48  Apr  17  15:28  altos
drwxr-xr-x 2   bin     1408  Oct  12  18:43  bin
drwxrwxrwx 3   bin      128  Jul  28  11:40  dict
drwxrwxrwx 4   root     640  Jul  28  11:55
drwxr-xr-x 2   john      96  Oct  12
drwxr-xr-x 11  lee      400  Oc
drwxrwxrwx13   bin                                lib
drwxrwxrwx 2   root                       11:55  preserve
drwxrwxrwx 10  root                  28   11:55  spool
drwxrwxrwx 3              48  Jul  28   11:55  src
drwxrwx                   48  Jul  28   11:55  sys
                         272  Oct  25   16:06  tmp
               unix      128  Oct  24   04:18  unix
      xrwx 2   user       64  Apr  17   15:29  user
drwxrwxrwx 2   user1     112  Oct  24   04:07  user1
drwxrwxrwx 2   user2      80  Oct  25   13:57  user2
drwxrwxrwx 2   user3      48  May  25   22:52  user3
drwxrwxrwx 2   user4      48  May  25   22:52  user4
drwxrwxrwx 2   user5      48  May  25   22:52  user5
drwxrwxrwx 2   user6      48  May  25   22:52  user6
drwxrwxrwx 2   user7      48  May  25   22:52  user7
drwxrwxrwx 2   user8      48  May  25   22:53  user8
drwxr-xr-x 3   vin       112  Oct  18   02:36  vin
$
```

Fig. 10-10. Moving from the bin to the usr directory. Since they are in different "legs" of the structure, the *full pathname* must be used (see Fig. 10-8.)

```
(A)  $ cd
     $

(B)  $ cd
     $ pwd

(C)  $ cd
     $ pwd
     / usr / user1
     $

(D)  $ cd
     $ pwd
     / usr / user 1
     $ ls -l/

(E)  $ cd
     $ pwd
     /usr/user1
     $ ls -l/
     total 363
     -rwxrwxr-x    1   root    2505  Apr  6   16:42  add.hd
     drwxr-xr-x    2   bin     2544  Oct  12  18:44  bin
     -rwxrwxr-x    1   root   11040  Apr  17  15:23  boot
     -rwxrwxr-x    1   root   10448  Apr  17  15:23  boot.fd
     drwxrwxrwx    3   root     864  May  29  21:43  dev
     drwxr-xr-x    4   bin      768  Oct  12  17:45  etc
     drwxrwxrwx    2   root      48  Jul  28  11:21  fd
     drwxrwxrwx    2   root      32  May  25  22:52  fptutor
     -rwxr-xr-x    1   root     287  Feb  1   00:00  install
     drwxrwxrwx    2   root     320  Jul  28  11:22  lib
     -rwxrwxr-x    1   root     709  Apr  17  15:23  load.hd
     drwxrwxrwx    2   root     832  Apr  17  15:30  lost+found
     -rw-r- -r- -  1   root    2656  Apr  17  15:23  priboot
     -rw-r- -r- -  1   root    1024  Apr  17  15:23  pribootfd
     drwxrwxrwx    2   root     656  Oct  25  16:54  tmp
     drwxr-xr-x    26  bin      416  Oct  12  17:44  usr
     -rwxrwxr-x    1   root   73408  Apr  17  15:23  xenix
     -rwxrwxr-x    1   root   73168  Apr  17  15:23  xenix.fd
     $
```

Fig. 10-11. The ls command is not confined to the current working directory.

This last example makes a total of three ways we have demonstrated that you can use the **ls** command:

The command alone.
The command plus an option.
The command (option is optional) with an argument.

CHANGING DIRECTORIES WITH A METACHARACTER

So far you have learned that you can use the **cd** command in conjunction with a pathname of a directory to make another directory the working directory. You can make a directory a working directory with a full pathname or, if the directory that you are going to make a working directory is subordinate to the current directory, you may make it the working directory with a relative pathname.

Entering a full pathname or even some relative pathnames, however, is tedious over the long haul, particularly when you want only to move to another of your directories. Therefore the Unix system provides a shorthand method that will aid you to make a directory "up" the hierarchical structure a working directory. The double period (..) shorthand character is used to indicate to the shell that it is to move to the parent (one level up) directory (Fig. 10-12).

- Screen A shows the **cd** command entered without a directory name, instructing the shell to make sure our home directory is the working directory.
- Screen B shows the **cd** .. command entered.
- Screen C shows the **pwd** command entered to check the name of the working directory.
- Screen D indicates that the working directory is **/usr**, which is one level above **/usr/user1**.

It is also possible to use the .. to move up several levels of directories (Fig. 10-13).

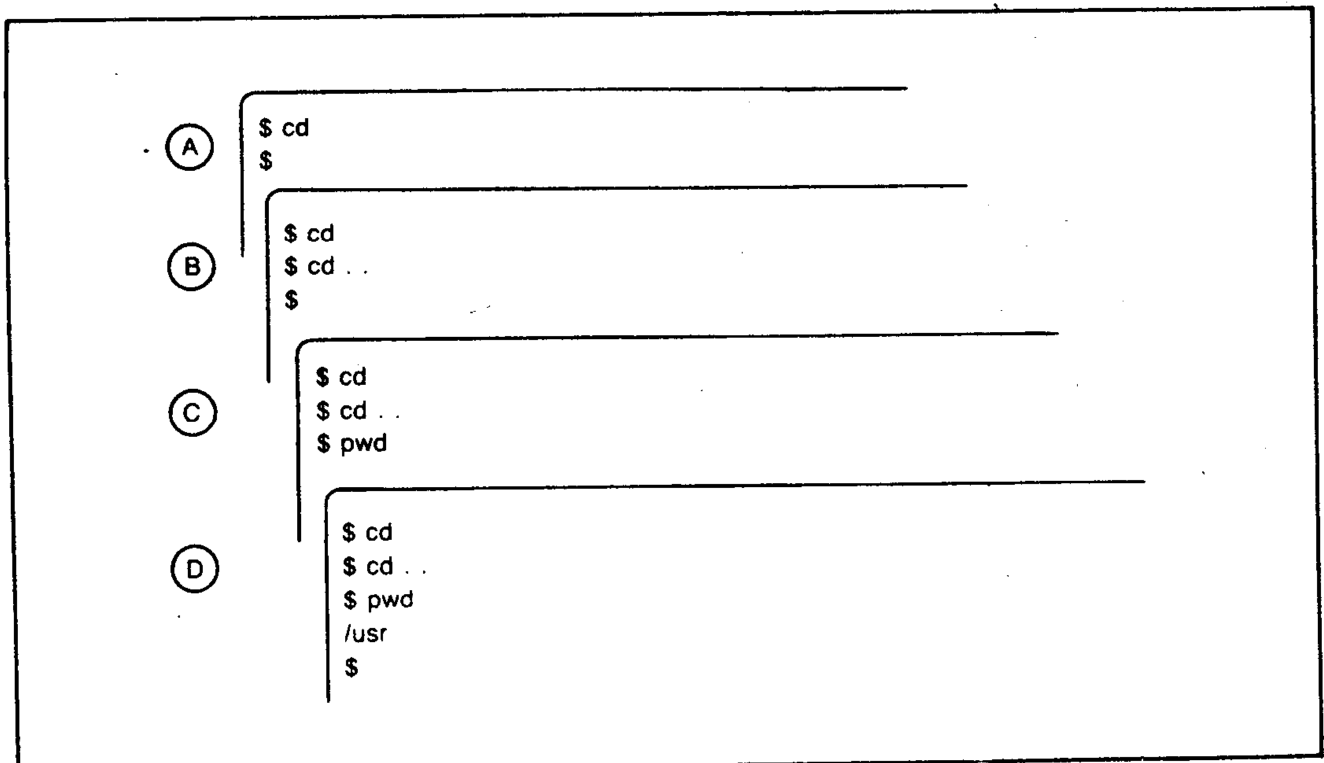


Fig. 10-12. The double dot is a shortcut to the parent of the working directory.

(A)
```
$ cd
$
```

(B)
```
$ cd
$ cd . ./. .
$
```

(C)
```
$ cd
$ cd . . / . .
$ pwd
```

(D)
```
$ cd
$ . ./. .
$ pwd
/
$
```

Fig. 10-13. The double dot can be used at more than one place in the pathname.

(A)
```
$ cd
$
```

(B)
```
$ cd
$ cd . ./user2
$
```

(C)
```
$ cd
$ cd . ./user2
$ pwd
```

(D)
```
$ cd
$ cd . ./user2
$ pwd
/usr/user2
$
```

Fig. 10-14. Another use of the double dot.

```
(A)
$ cd
$

(B)
$ cd
$ ls -l . .

(C)
$ cd
$ ls -1 . .
total 28
drwxrwxrwx 2  bin      128  Jul  28  11:22  adm
drwxrwxrwx 2  altos     48  Apr  17  15:28  altos
drwxr-xr-x  2  bin     1408  Oct  12  18:43  bin
drwxrwxrwx 3  bin.     128  Jul  28  11:40  dic
drwxrwxrwx 4  root     640  Jul  28  11:55
drwxr-xr-x  2  john      96  Oct  12
drwxr-xr-x 11  lee      400  Oct                lee
drwxrwxrwx13  bin      592             18:42  lib
drwxrwxrwx 2  root                28  11:55  preserve
drwxrwxrwx10  root           Jul  28  11:55  spool
drwxrwxrwx 3  sy          48  Jul  28  11:55  src
drwxrwxrwx                48  Jul  28  11:55  sys
drwxr           ot      272  Oct  25  16:06  tmp
d           3  unix     128  Oct  24  04:18  unix
      xrwx 2  user      64  Apr  17  15:29  user
drwxrwxrwx 2  user1    112  Oct  24  04:07  user1
drwxrwxrwx 2  user2     80  Oct  25  13:57  user2
drwxrwxrwx 2  user3     48  May  25  22:52  user3
drwxrwxrwx 2  user4     48  May  25  22:52  user4
drwxrwxrwx 2  user5     48  May  25  22:52  user5
drwxrwxrwx 2  user6     48  May  25  22:52  user6
drwxrwxrwx 2  user7     48  May  25  22:52  user7
drwxrwxrwx 2  user8     48  May  25  22:53  user8
drwxr-xr-x  3  vin      112  Oct  18  02:36  vin
$
```

Fig. 10-15. The ls command also will accept the double dot shorthand.

- Screen A shows the **cd** command entered to get us back to our home directory, so that we will have some room to move up the hierarchical structure.
- Screen B shows the **cd** ../.. command entered. We started at the **/usr/user1** level of the structure. This command line should have moved us up to the **usr** and then to the / level of the structure.
- Screen C shows the **pwd** command entered to check the name of the working directory.
- Screen D displays the pathname /, proving that the command moved us up two levels.

It is also possible to use the .. characters in conjunction with a pathname to move up the structure and down another leg (Fig. 10-14).

- Screen A shows the **cd** command entered to make our home directory the working directory.
- Screen B shows the command **cd ../user2** entered to make the directory, **user2**, the working directory.
- Screen C shows the **pwd** command entered to check to see if the working directory is now **/usr/user2**.
- Screen D indicates that the working directory has been correctly changed.

If there were additional directories in the **user2** directory, we could have tagged one of them on the end of this pathname. In any case, the .. characters have been demonstrated to save you the trouble of having to enter **/usr** in the pathname in order to make the directory **/usr/user2** the working directory.

The double period also may be used with the **ls** command to list the contents of a directory (Fig. 10-15), just as we entered a pathname (**/**) with the **ls** command in Fig. 10-11, Screen D.

- Screen A shows the **cd** command entered to make our home directory the working directory.
- Screen B shows the **ls -l ..** command entered, instructing the shell to display the contents of the directory at the parent directory level—which is the directory **/usr**.
- Screen C lists the contents of the directory **usr**.

These exercises have provided you with all the tools you will need to be well on your way to "navigating" the hierarchical structure on your own. As you can see, with just half a dozen commands you can find any file or directory in the structure, work in any directory, and more.

Chapter 11

Making New Files and Directories

In the first two exercise sessions you learned to log on to a Unix system, and to navigate through the Unix hierarchical file structure. Now it is time for you to learn to make some directories and files so that you can construct your own part of the hierarchical file structure.

There are several commands and methods for making a new file. The typical method for making a file is based on using an application program such as a word processor. When you make a new file with a word processor, it is effectively the same as if you were making it with Unix commands. If you doubt this, you can utilize the commands that you learned to use in our second session to move to your home directory. To confirm this statement, list the contents of the directory to check the names of your files.

Another method for making a file is to use one of the Unix editors such as **ed**, **ex**, **vi**, etc. However, in order to avoid including a considerable amount of text describing how to use these editors, we will create only some very simple files with the Unix shell redirect output (>) command. In this session we will introduce the following commands:

echo Displays the argument written on the command line. It is generally used in conjunction with other Unix commands.

> Used to direct the output of a command into a file. We will be using it in conjunction with **echo** and other commands to make new files in the hierarchical structure.

cat Used to display the contents of a file.

date Will display the current date and time (if the computer has an internal clock).

mv Used to move a file from one location in the hierarchical structure to another. Also used to rename a file and to make new files.

cp Used to make a copy of the contents of a file. Also used to make new files.

mkdir Used to make a new directory.

MAKING FILES WITH THE REDIRECTION COMMAND

We will use the **echo** command in conjunction with the redirection command to create some simple files with which we can perform some further exercises. The redirection command allows you to redirect the output from a Unix command into a file instead of to your terminal monitor.

The format for the **echo** command is:

echo "argument"

Figure 11-1 illustrates a typical sequence

- Screen A shows the command **echo** followed by the argument "**Welcome to Unix**".
- Screen B shows **Welcome to Unix** again.

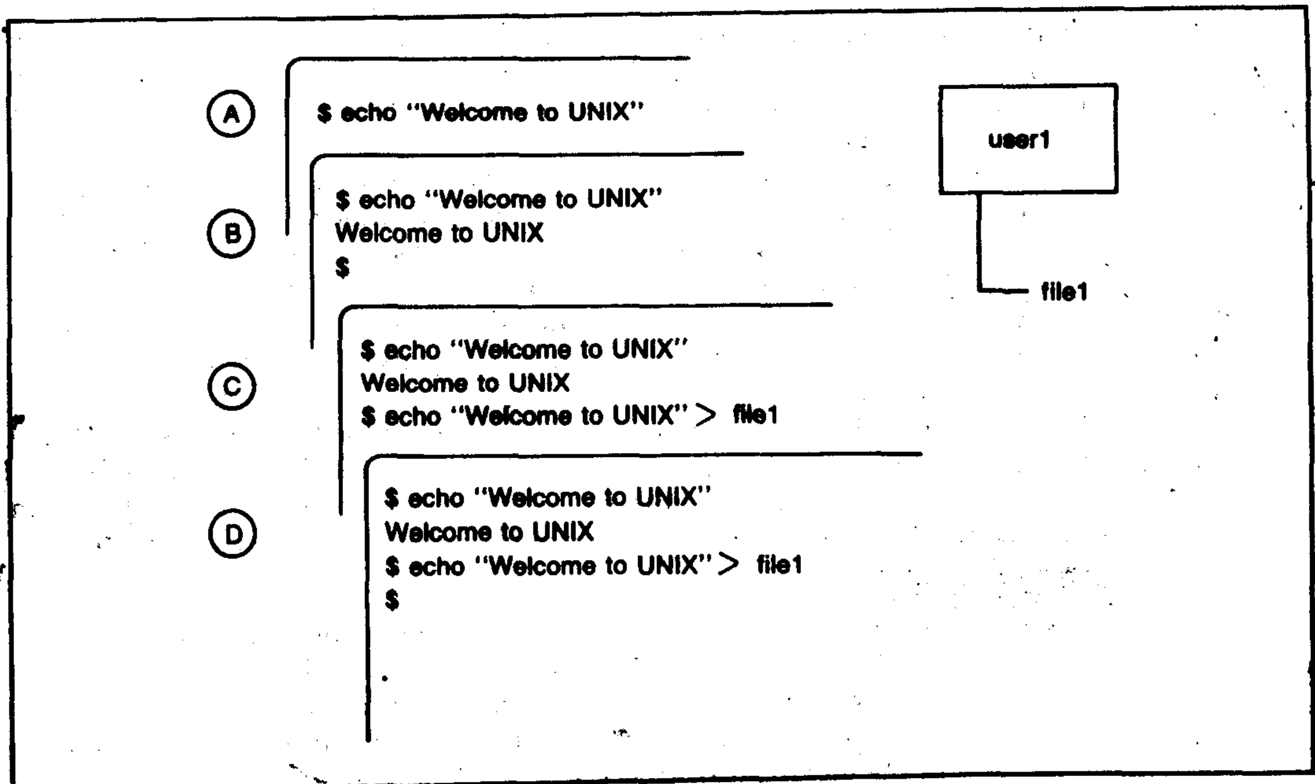


Fig. 11-1. A typical sequence of the echo command, which can be used to create a file.

The echo command has "echoed" the command line argument, which in our example is the statement, Welcome to Unix, that we entered in part A. This is not very impressive, but it demonstrates that the normal output of the echo command is to display the command line argument on your terminal monitor.

We will now show you how to redirect output of the echo command, the argument on the command line, to a file named file1. The file1 file does not yet exist in the hierarchical structure, as we saw when we used the ls command. The redirect will have to create file1 in order to be able to put the argument into it. The format for making a file with the echo command and redirection is:

echo "argument" > new file name

- Screen C shows the command line redirecting the argument Welcome to Unix to file1.
- Screen D displays only the shell $ prompt. The argument is not displayed as it was in Screen 1B.

Welcome to Unix was not displayed on your terminal monitor, because the output of the echo command was supposedly redirected to the file named file1. Let's see if it was.

First of all, is there a file named file1 in our home directory? To find out, we will use the ls -l command (Fig. 11-2).

- Screen A shows the ls -l command entered.
- Screen B indicates that there is a line entry (a file) in our directory with the file name file1.

We now know that there is a file named file1 in our directory, but do we know that it contains the argument that we entered? To find out if it does, we can use the cat command to look at the contents of file1.

The format for the cat command is:

cat file name

- Screen C shows the cat file1 command line entered. (The file name, file1, is the command line argument).
- Screen D displays our command argument as we entered in Fig. 11-1, Screen C. (The beginning and ending quotes are used to delineate the argument in the echo command line. They are not part of the argument and will not be included in file1.)

Let's make another file and check its contents (Fig. 11-3).

- Screen A shows echo "John Jones" > file2 entered.
- Screen B shows the cat file2 command entered to check the contents of file2.
- Screen C indicates that John Jones was redirected to and is now contained in the file named file2.

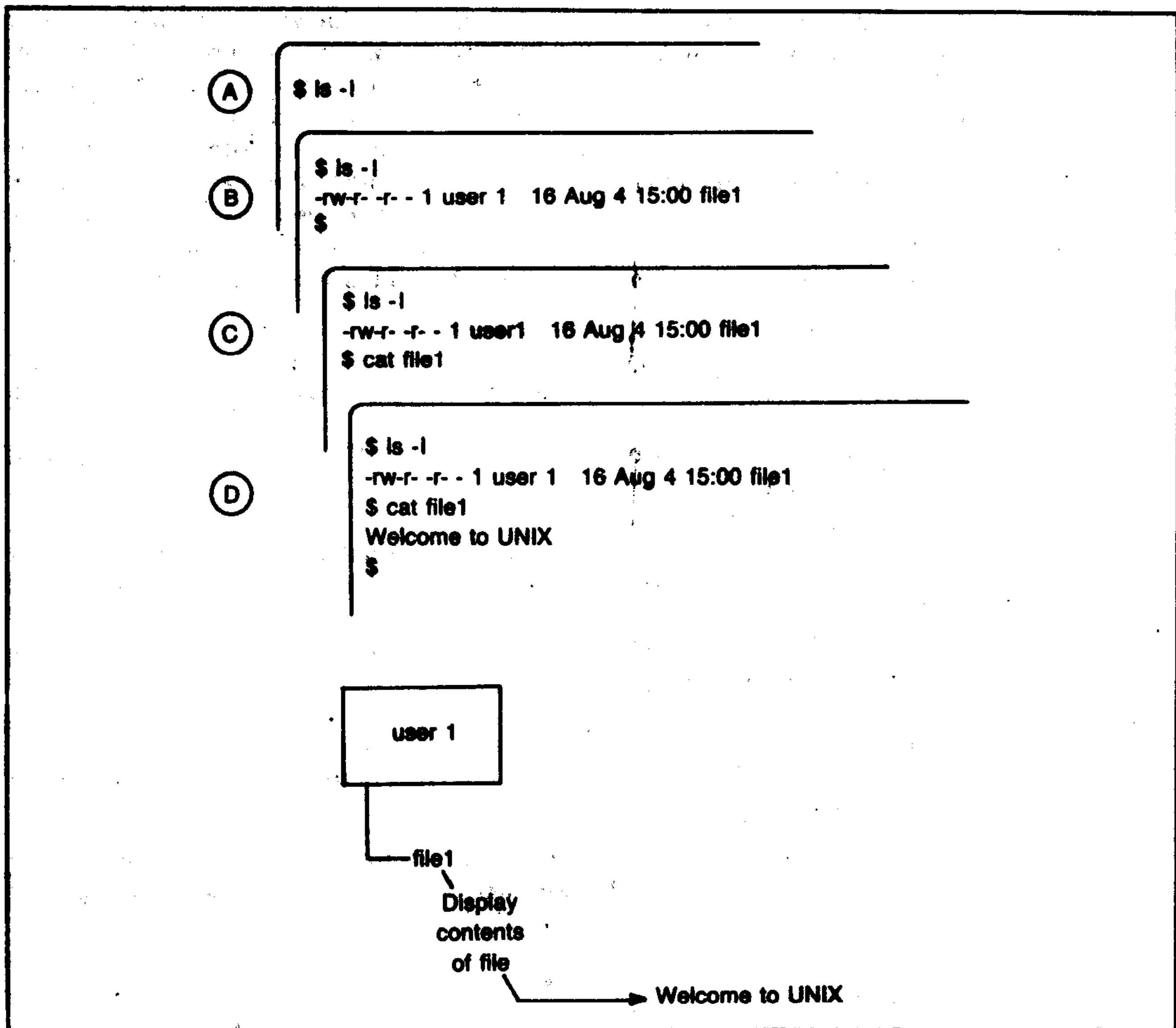


Fig. 11-2. Examining the contents of file1 with the cat command.

We have now confirmed that the redirection command can be used in conjunction with the **echo** command to make a file.

Next, let's use the redirect output command with another Unix command named **date**, to see if it will work the same way with another command. The standard output of the **date** command is to display the current date and time on your terminal monitor. The sequence appears in Fig. 11-4.

- Screen A shows the command **date** entered.
- Screen B displays the current date and time (Monday, August 4, 15:04:23).
- Screen C shows **date > file3** command entered.
- Screen D shows the **ls -l** command entered to check to see if a file3 was created.

- Screen E indicates that this has happened.
- Screen F shows the cat **file3** command entered to check the contents of **file3**.
- Screen G displays the date and time.

We have now proven that the redirect output command will work with at least two commands, whose output is normally displayed on the terminal monitor, to form new files.

The **echo** command with redirection is a very quick way to make files. More than one line can be created, but, since redirection provides you very limited correction and editing facilities, it is not used to any great extent for this purpose.

To create a second and more lines with the **echo** and redirect commands:

$ echo "Now is the time (press Return)
> to come to the aid (press Return)
> of your microcomputer" (press Return)

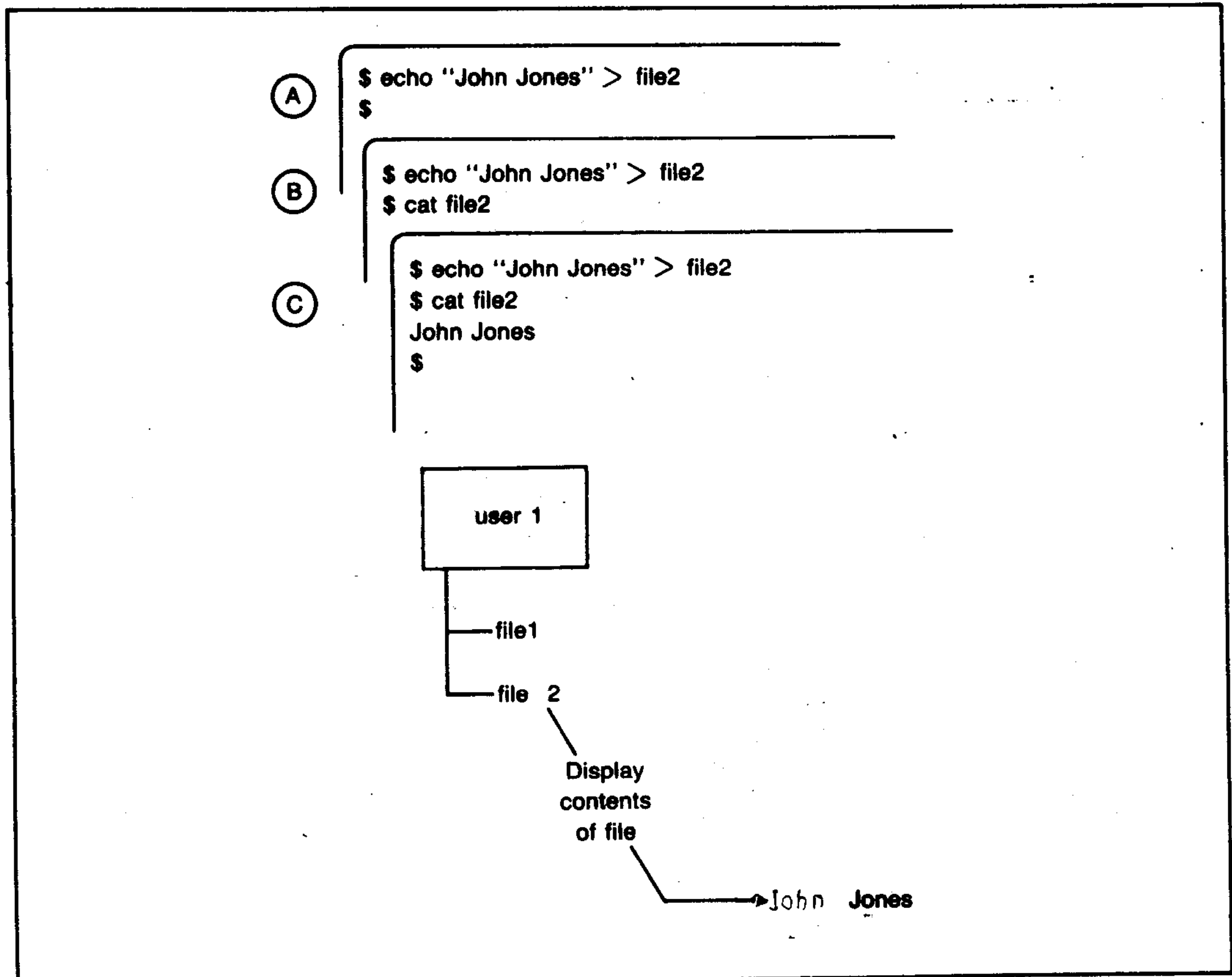


Fig. 11-3. Creating and verifying file2.

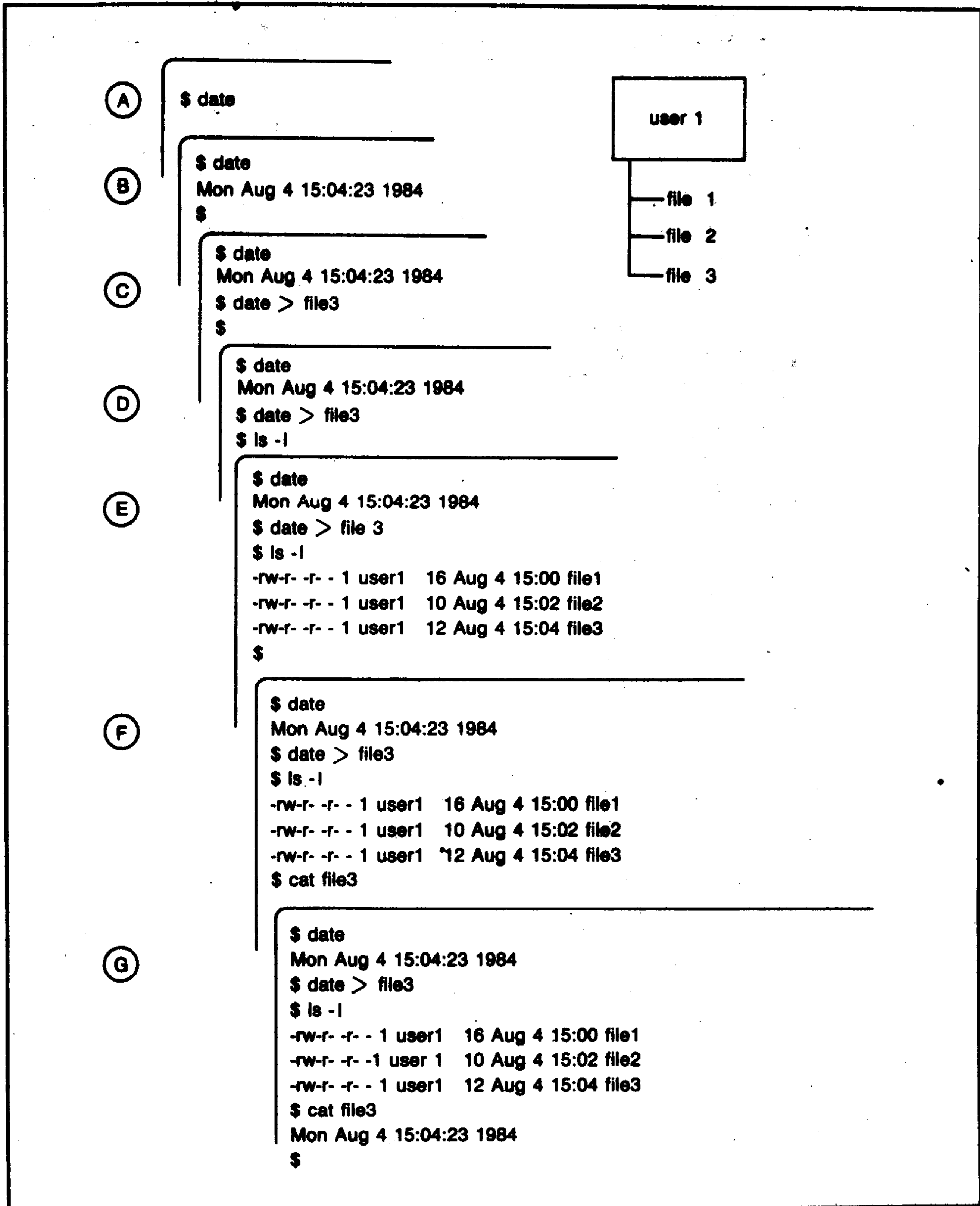


Fig. 11-4. Creating file3 with the date command.

After you enter the closing quote and press the Return key, the shell will execute the echo command and its argument. If you had added the redirect command and the name of a file to the command line, the shell would have put the argument into the specified file.

MAKING FILES WITH APPENDED REDIRECTION

Another command that can be used to make a file is the **cat** command. The difference between the two commands is that the **echo** command can generate a file with new information and the **cat** command can only use existing files to make new files. The format for using the **cat** command to make a file is:

cat file name(s) > new file name

Now examine Fig. 11-5.

- Screen A shows **cat file1 > file4** entered.
- Screen B shows the shell **$** prompt. The output from the cat command was redirected to **file4**.
- Screen C shows **ls -l** entered to check to see if **file4** was created.
- Screen D confirms the existence of four files in our home directory.
- Screen E shows **cat file4** entered to check the contents of **file4**.
- Screen F displays **Welcome to Unix**—which confirms the **cat** command copied the contents of **file1** into **file4**.

The **cat** command can also be used to concatenate (join) two or more files (Fig. 11-6).

- Screen A shows the **cat file1 file2 file3** command entered.
- Screen B displays the contents of our three files in the order that we requested them in part A.

We now know that we can redirect the output of some commands into a file, and that we can use the **cat** command to display the contents of several files with a single command line entry. The next question is: Can we redirect the output of the **cat** command to an existing file? The answer is yes, but *not* with the single > redirection.

If you use the single > to redirect the output of a command to an existing file, the redirected information will be written over the existing contents of the file to which the command is directed, replacing the information in the same way as when you record over an existing song on a cassette in your tape recorder. Overwriting is fine only if you do not need the existing contents of a file.

To add or append information to an existing file, you need to use the redirect-append command, a double >>, as shown in Fig. 11-7.

- Screen A shows the **cat file1 file2 file3 >> file4** command entered.
- Screen B shows the **cat file4** command entered to check the contents of **file4**.

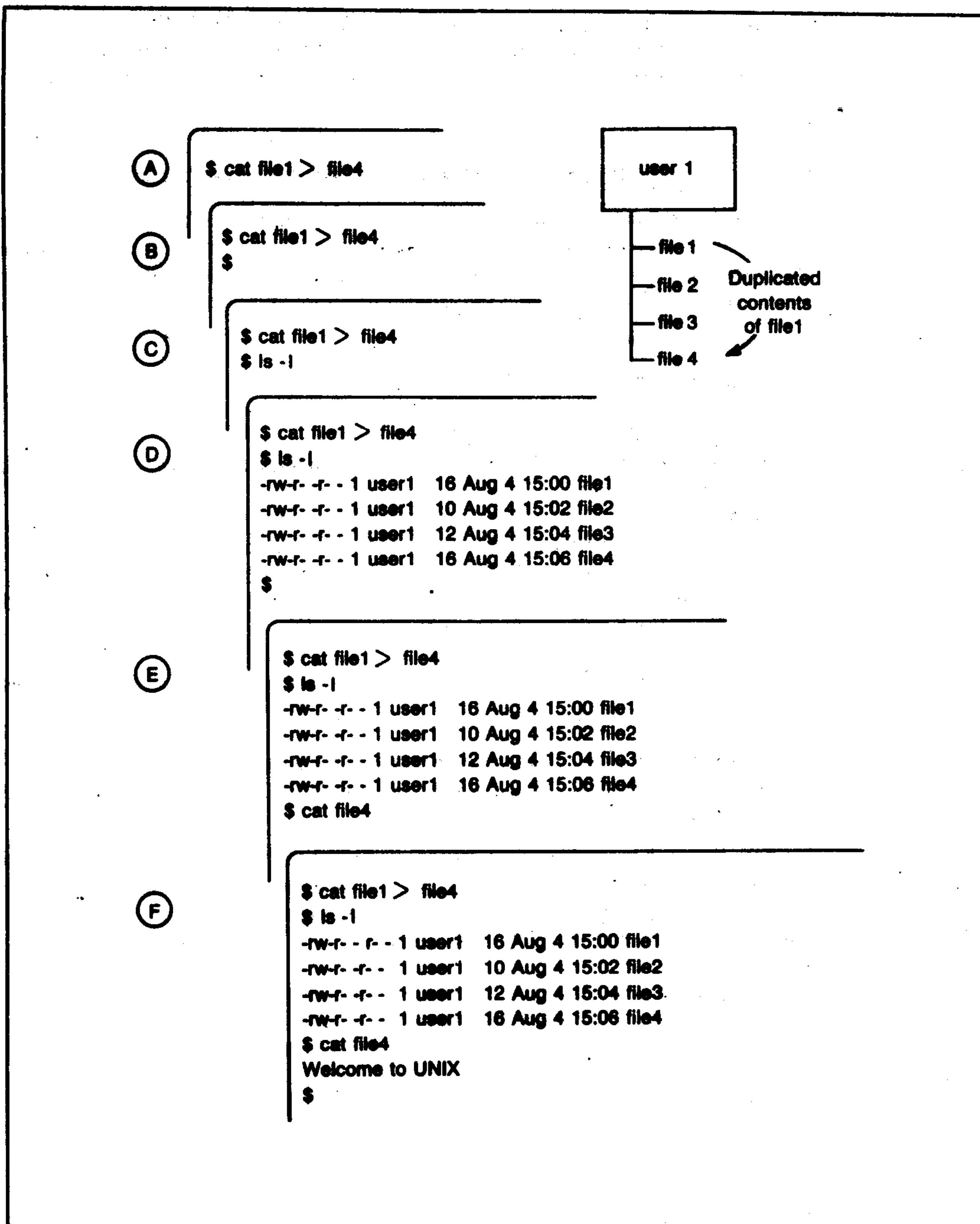


Fig. 11-5. The cat command can be used to copy a file.

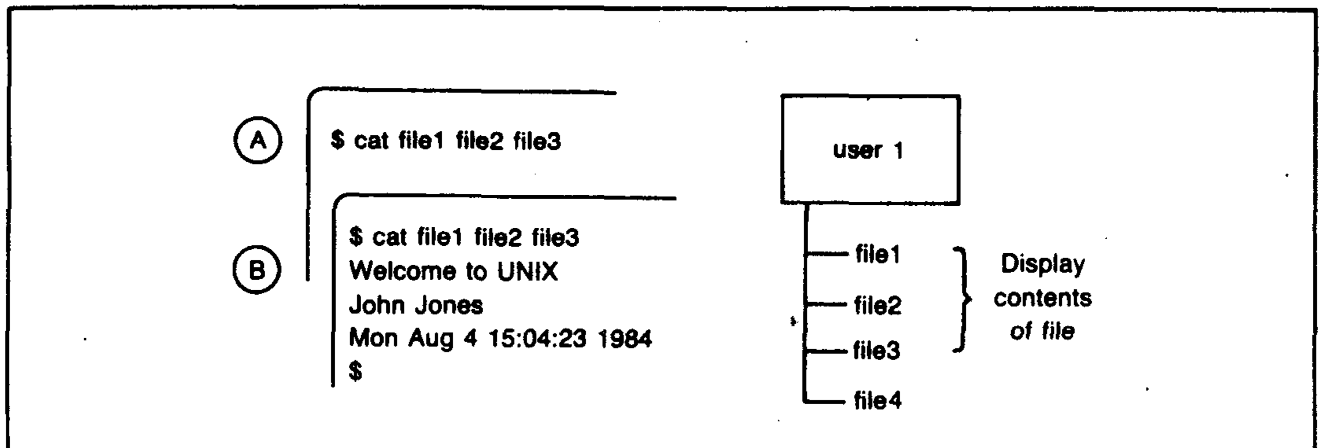


Fig. 11-6. The cat command also can be used to combine (concatenate) files.

- Screen C displays the previous contents of file4 (copy of file1), followed by the addition of the contents of file1, file2, and file3.

MAKING FILES WITH THE MOVE COMMANDS

The move command (mv) is another command that can be used to make a new file. The mv command is used to move a file from one hierarchical file structure location to a new location in the structure. Actually, the mv command does not physically move any files; it only renames a file's pathname, giving the illusion that the file is being moved. The only move actually made is on the graphic representation of the hierarchical file structure.

The command line format for the mv command is:

mv original file name > new file name

Use of the mv command is shown in Fig. 11-8.

- Screen A shows mv file1 dog entered, changing the name of file 4 to a file named dog.
- Screen B shows cat dog entered to display the contents of the file named dog.
- Screen C displays the contents of dog indicating the file1 has been renamed dog.
- Screen D shows the ls -l command entered to determine if there is still a file1 listed in the directory.
- Screen E displays the listing of files in our home directory, showing that file1 is no longer listed.

MAKING A FILE WITH THE COPY COMMAND

The copy command (cp) can be used to make a copy of an existing file and put the copy into a new file. Making a copy of a file will not affect the original file name or its contents. The cp command is particularly useful when you want to maintain a copy of a file and work on the original, or let another user make a copy.

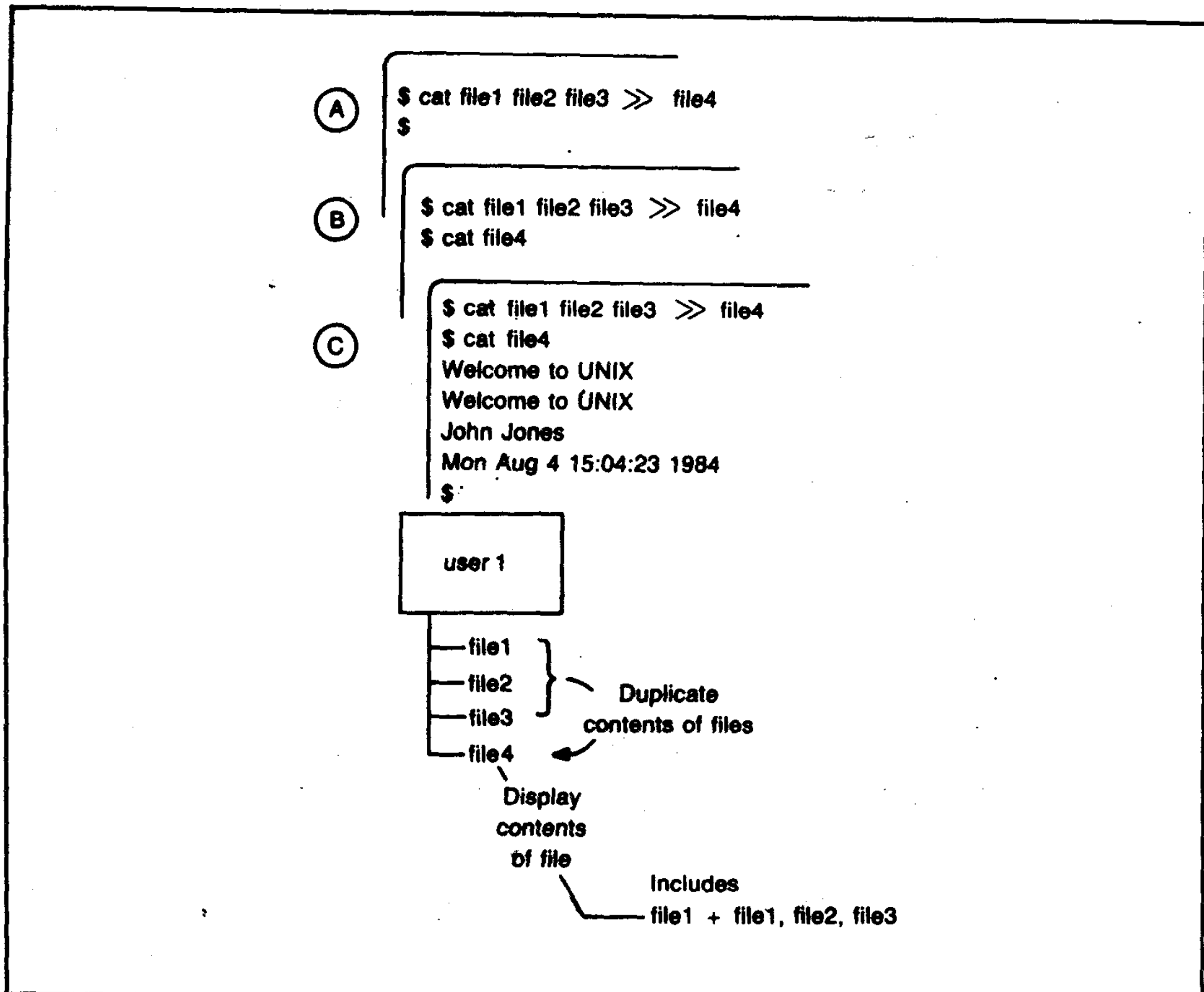


Fig. 11-7. Using the redirect-append metacharacter with cat.

The cp command format is:

cp file to be copied new file name

Figure 11-9 shows how to use it.

- Screen A shows the cp file2 file5 command entered to make a copy of file2 in the user1 directory.
- Screen B shows the ls -l command entered to determine whether a file5 has been generated. The resulting display indicates that there is now a file5 listed in the directory.

A copy of a file may be generated in another directory as well. The copy may use the original file name (in this case), or you may enter a new file name. However, since we have not made any other directories with which we can experiment at this

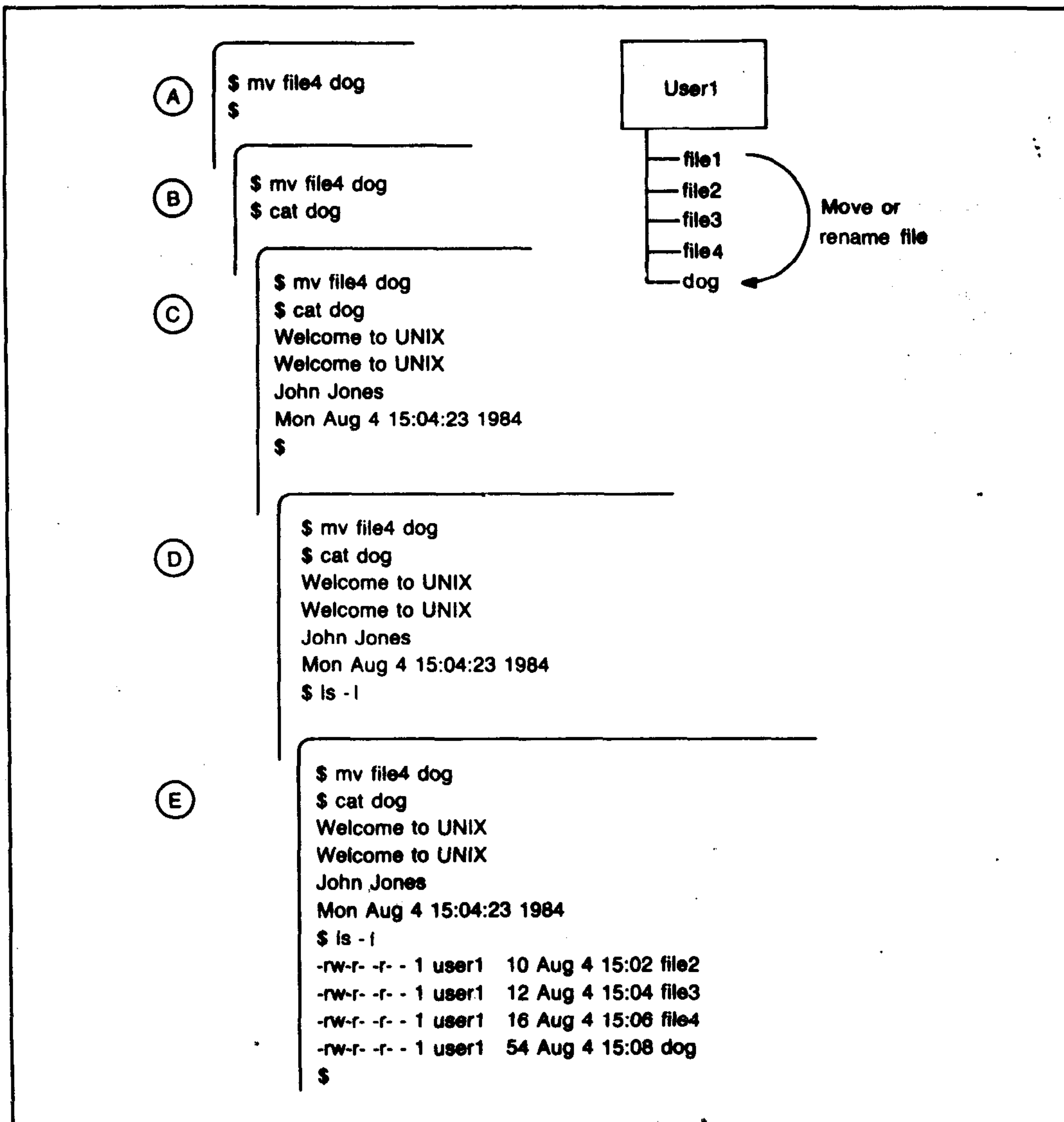


Fig. 11-8. Creating files with the move (mv) command.

time, we will have to wait to try this until later in the exercise, after we have learned to create directories.

The copy of a file may also be generated in another user's directory. However, to do this requires some working knowledge of file security. File security will be covered in Chapter 13. At that time I will demonstrate using the **cp** command to make a copy in one user's directory and put it in another user's directory.

MAKING A DIRECTORY

To make a directory, we move to the directory in which we wish to make the new directory and we use the **mkdir** command. The format for using the **mkdir** command is

mkdir name of new directory

and the sequence of screens in Fig. 11-10 shows how to use it.

- Screen A shows the **pwd** command entered to determine the name of the current directory.
- Screen B indicates that we are in our home directory **/usr/user1**.
- Screen C shows the **mkdir dir1** command, entered to make a new directory called **dir1**.

The full pathname of this directory, for reference, will be **/usr/user1/dir1**, but we need only enter the relative pathname, **dir1**, with reference to the working directory **user1**.

- Screen D shows the **ls -l** command entered to determine that the directory **dir1** was created.

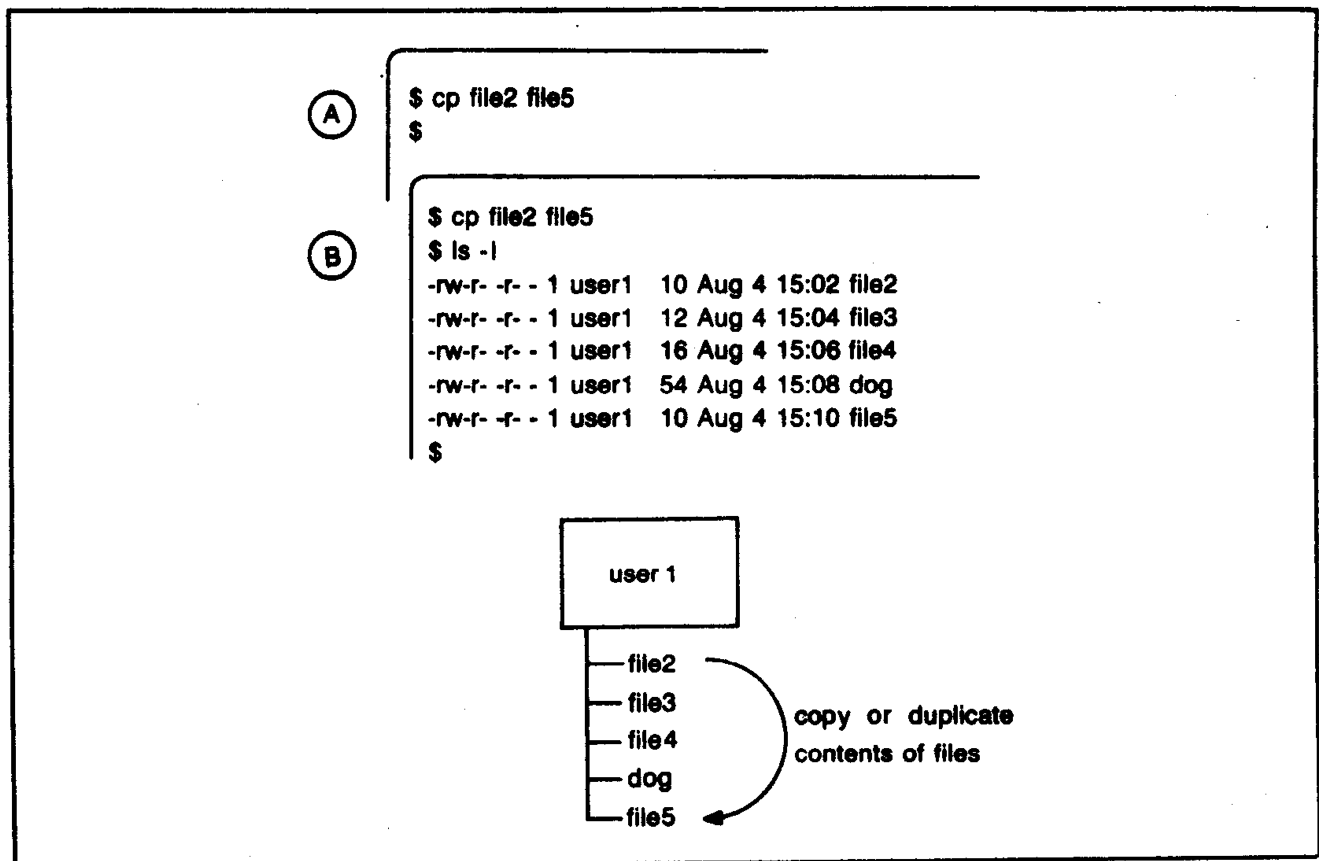


Fig. 11-9. Making a file with the copy (cp) command.

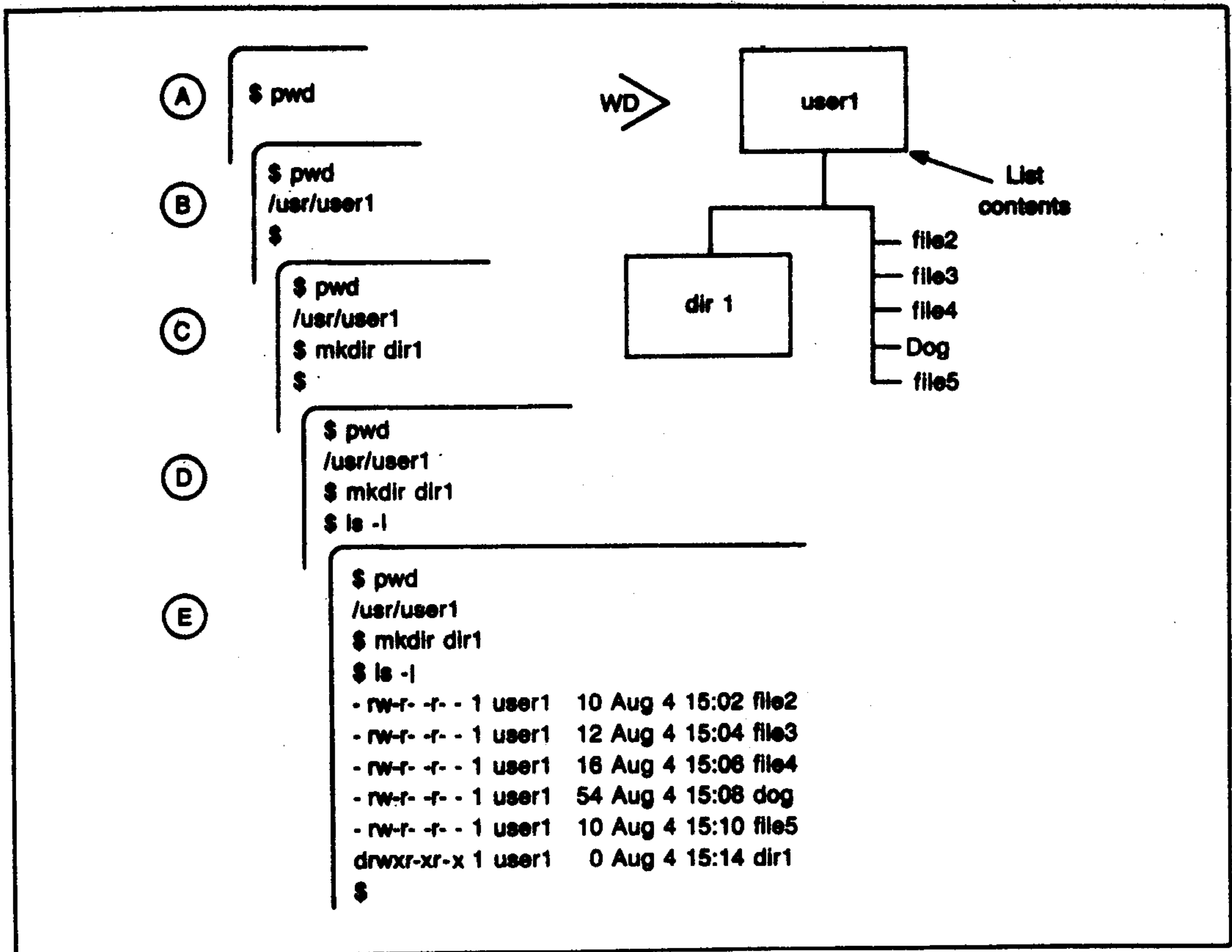


Fig. 11-10. Creating a directory.

- Screen E indicates that it was. The d at the beginning of the line indicates that it is a directory.

The next part of the exercise will show you two things. First, the directory in which you are creating a subdirectory does not have to be the parent directory. The second thing is that you can use a pathname, full or relative (depending on the relative location between the working directory and the directory to be created), to create a directory. Now examine Fig. 11-11.

- Screen A shows the pwd command entered to determine the current directory.
- Screen B indicates that the current directory is still our home directory, /usr/user1.
- Screen C shows the mkdir dir1/dirA command entered. For reference, the full pathname of this directory will be /usr/user1/dir1/dirA.

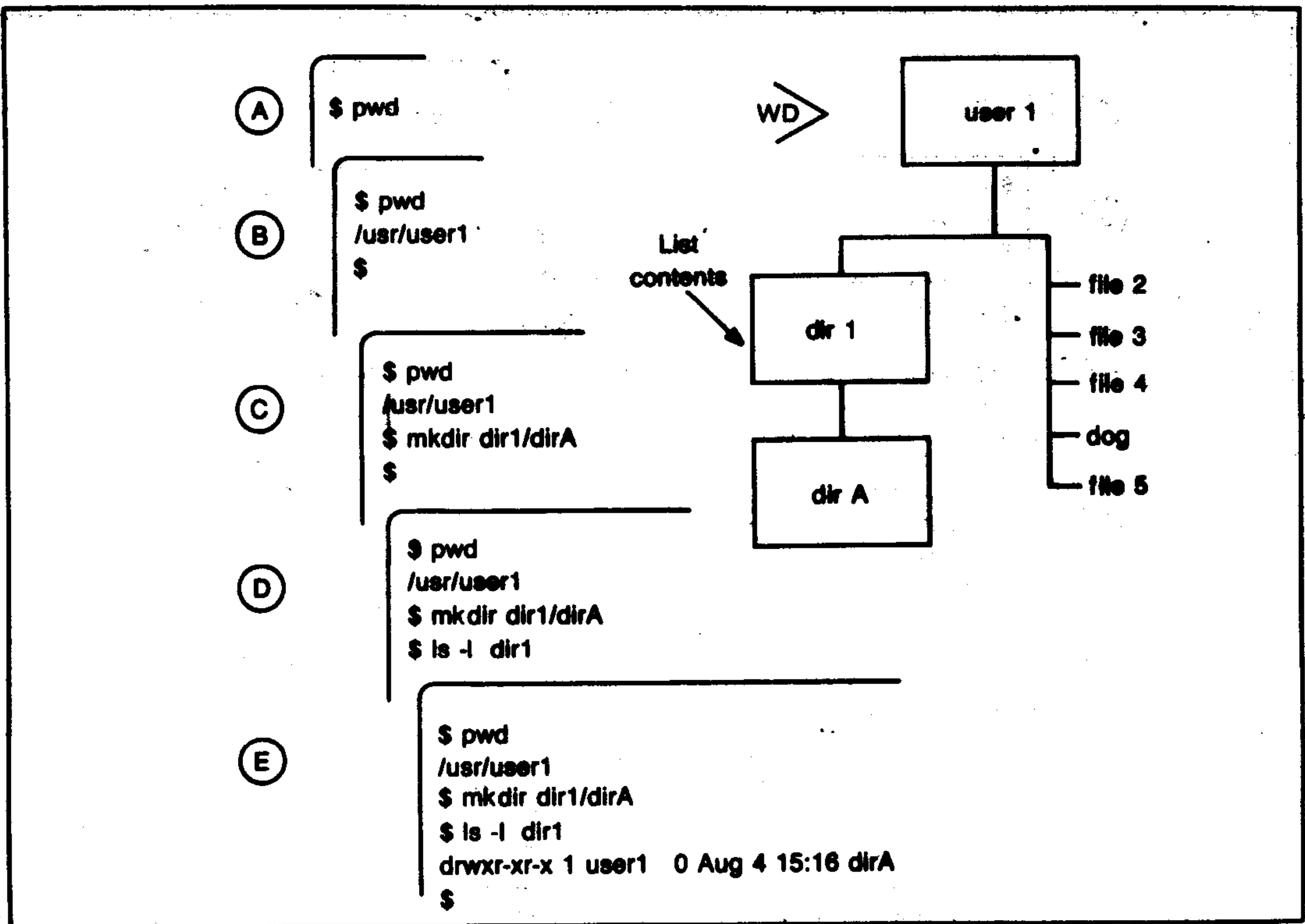


Fig. 11-11. A relative pathname can be used to create a directory outside the working directory.

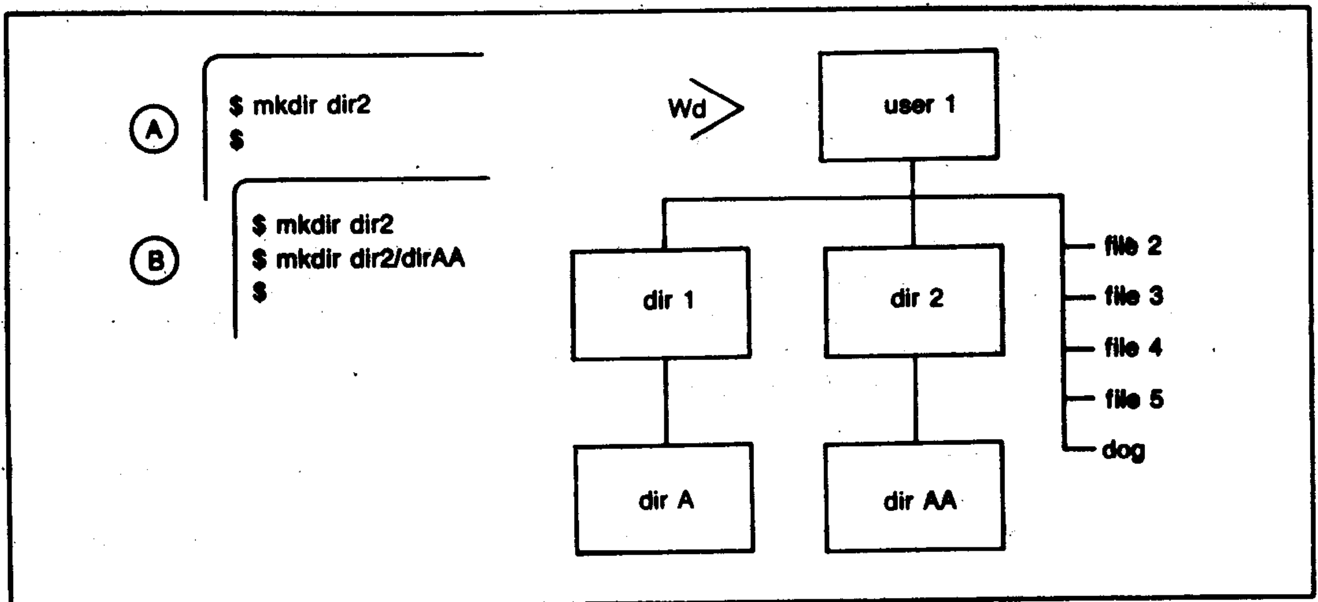


Fig. 11-12. Creating another directory with the mkdir command.

- Screen D shows the ls -l dir1 command entered to determine whether the directory dirA was created.
- Screen E proves that a relative pathname can be used successfully to create a new directory in a directory other than the working directory.

You might note that the ls -l command was activated from a directory level above the directory whose contents we wanted to investigate. The ls command, as you remember from the exercises in Chapter 10, can be used in conjunction with a full or relative pathname, depending on your relative location in the hierarchical structure.

For practice and to set up for the next exercise, let's create another directory (Fig. 11-12).

- Screen A shows the mkdir dir2 command line entered. The full pathname of this directory, for reference, is /usr/user1/dir2.

(A)
```
$ cd
$
```

(B)
```
$ cd
$ pwd
```

(C)
```
$ cd
$ pwd
/usr/user1
$
```

(D)
```
$ cd
$ pwd
/usr/user1
$ cp file2 dir1/dirA/file1
$
```

(E)
```
$ cd
$ pwd
/usr/user1
$ cp file2 dir1/dirA/file1
$ ls -l dir1/dirA
```

(F)
```
$ cd
$ pwd
/usr/user1
$ cp file2 dir1/dirA/file1
$ ls -1 dir1/dirA
-rw-r- -r- - 1 user1 10 Aug 5 13:12 file 1
$
```

Fig. 11-13. Creating files in other directories.

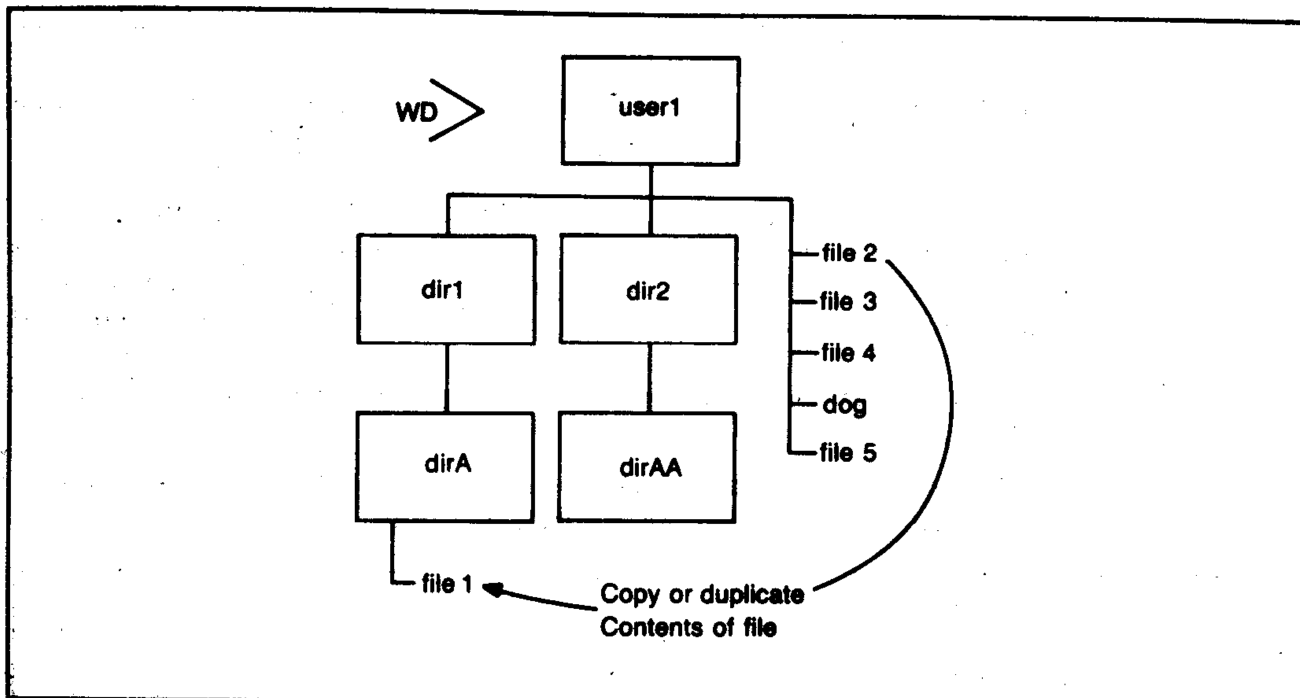


Fig. 11-14. The structure that results from the commands in Fig. 11-13.

- Screen B shows the mkdir dir2/dirAA command line entered. The full pathname of this directory, for reference, is /usr/user1/dir2/dirAA.

We could just as well have named this directory dirA instead of dirAA—without interfering with the directory name dirA under dir1. However, we will use dirAA in order to eliminate any confusion in these examples.

FILE CREATION USING PATHNAMES

In two of the earlier parts of this exercise session, we used the cp and mv commands to create files in the working directory. As promised, we will now learn that you can create files with these commands from a working directory other than the directory in which you are making a file.

The ability to list the contents of directories and files, and to make directories and files in a nonworking directory, is one of the features of the Unix system design. We will make a copy of file1 in our home directory and put the copy in our lower-level directory dir1/dirA. See Fig. 11-13.

- Screen A shows the cd command entered to return us to our home directory. That is, make our home directory the working directory.
- Screen B shows the pwd command entered to assure ourselves that our home directory is now the working directory.
- Screen C verifies that /usr/user1 is the current directory.
- Screen D shows the cp file2 dir1/dirA/file1 command entered to make a copy of file2 in directory dirA named file1.

- Screen E shows the ls -l dir1/dirA command entered to determine that the file1 was copied to dirA.
- Screen F indicates that file2 was copied to dir1/dirA/file1. There is a file called file1 listed in dirA. The resulting structure is shown in Fig. 11-14.

The next exercise (Fig. 11-15) will move file1 in dirA to dir2/dirAA to show you that you can move from one branch of the hierarchical structure to another, from our home directory, /usr/user1.

- Screen A shows the mv dir1/dirA/file1 dir2/dirAA/file1 command entered to move file1 to dirAA. (Both of these file names are relative to the current working directory, our home directory.)

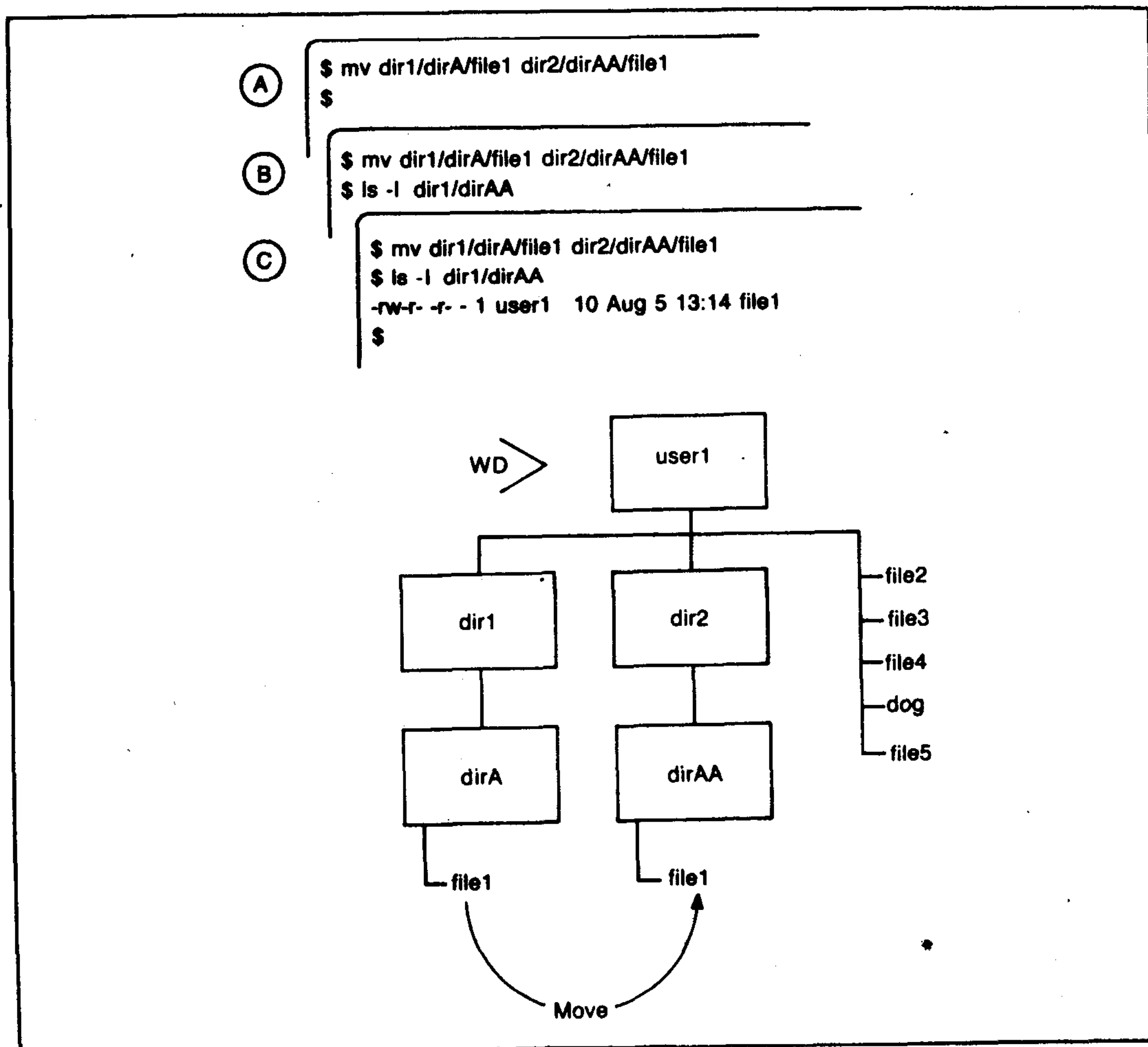


Fig. 11-15. Moving a file from one sub-directory to another.

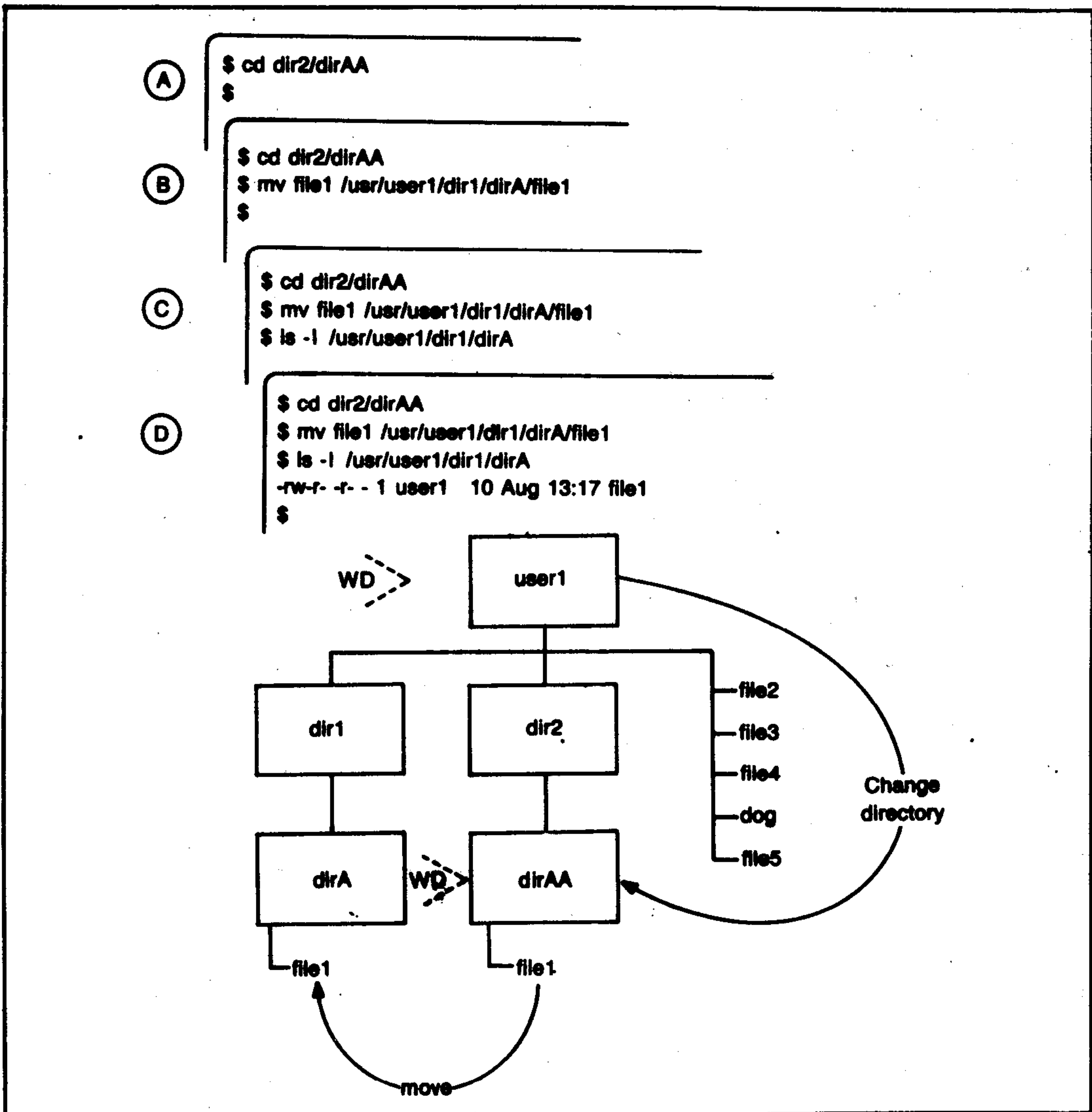


Fig. 11-16. Operating out of a sub-directory in your hierarchical file structure.

- Screen B shows the ls -l dir1/dirAA command entered to check that file1 was moved.
- Screen C lists the contents of dirAA.

We will now change the working directory to /usr/user1/dir2/dirAA and move file1 back to dirA, in order to show you that you can operate between directories in different branches of your hierarchical file structure (Fig. 11-16).

- Screen A shows the cd dir2/dirAA command line entered to change the working directory.
- Screen B shows the mv file1 /usr/user1/dir1/dirA/file1 command line entered to move file1 back to dirA. (The shorthand form is ../../dir1/dirA)
- Screen C shows the ls -l /usr/user1/dir1/dirA command line entered to check the contents of dirA.
- Screen D shows that file1 is again listed in dirA.

In the preceding exercises, we have always supplied the new file name in each example. If you do not indicate a new name, the shell will automatically retain the

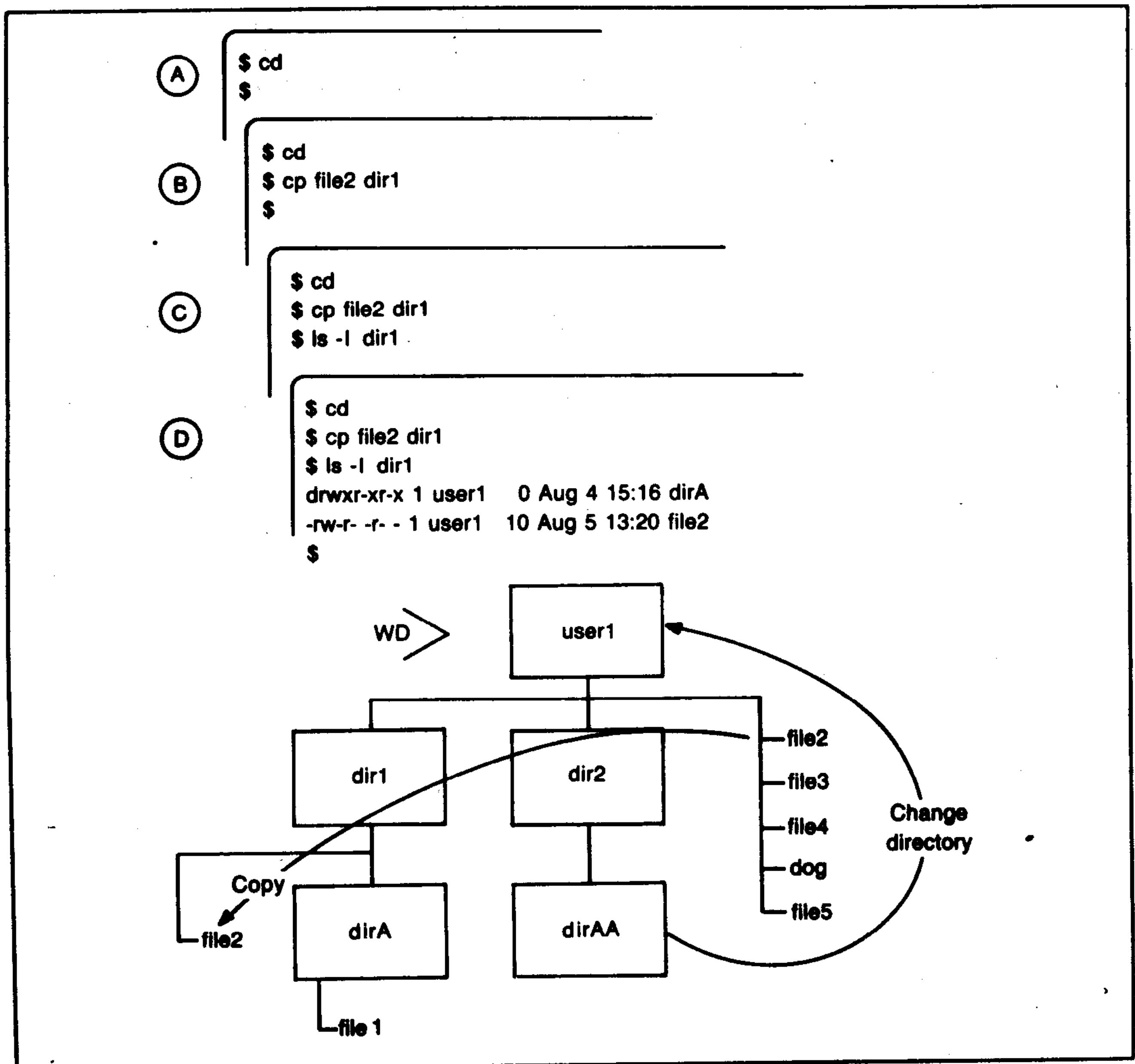


Fig. 11-17. If no new file name is specified, the shell assumes that the old one is to be retained.

old file name—*not* the file's pathname. Examine the sequence and structural diagram of Fig. 11-17.

- Screen A shows the command **cd** entered in order to make our home directory the working directory.
- Screen B shows command line **cp file2 dir1** entered (without a file name to indicate a new file name), to create a copy of **file2** in the **dir1** directory.
- Screen C shows the **ls -l dir1** command, entered to determine whether there is a **file2** now located in the **dir1** directory.
- Screen D indicates that a **file2** was created in **dir1**.

This session has been long, but we have accomplished a lot and a lot of information has been covered. Keep in mind, however, that we did it all with fewer than 10 commands. The basic operating principles of each of these commands and their varied applications are extremely simple, but very powerful.

Chapter 12

File Security

The Unix system provides for two types of security. *System-level security* keeps unauthorized individuals from gaining access to the system, and *file-level security* keeps authorized system users from gaining access to one another's files.

System-level security was discussed in the first lab session. We will now discuss file-level security. The commands we will cover in this session are:

chmod The "change permission mode bits" command is used to change the type of access allowed or disallowed to users in an assigned group and/or to all users.

chown The "change ownership of a file" command is used by the system administrator to change the name of the registered owner of a file.

CHANGING FILE SECURITY

File security is controlled by the permission or mode bit settings. As explained in Chapter 4, there are nine permission bits controlling read, write, and execute permissions. These bits can be turned on or off, allowing or disallowing access to a file by the owner of the file, a group member, and/or all users.

The permission bits are changed with the **chmod** command, the format for which is:

chmod security bit code file name

The permission bits are changed by entering a code for the desired security combinations, such as who should have access (owner, group individual, all users) and what access each will be allowed to have.

The security permission bit code is based on a binary number. Each group of three permission bits (owner, group, all users) is set individually on or off by the code number entered in conjunction with the chmod command for each type of permission. The coded number is based on the position of the individual bit in the string of permission bits, represented in the binary number system.

"Binary" is one of those words associated with computers that tends to frighten many people. The binary number system is based on a counting scheme that differs from our common, decimal-based system in one respect, but otherwise operates on exactly the same additive principle we learned in grade school. The only difference is that, with the binary system, we have fewer discrete digits we can use to represent numbers. Where the decimal system gives us ten different digits (0 through 9) to work with, the binary number system gives us two—0 and 1.

Now, stop and think for a moment about what the decimal representation of a number is actually telling us. For example, in the decimal representation

32767

we divide it mentally into "columns" or "places" that are assigned values (from right to left) in the sequence 1, 10, 100, 1000, etc. Having done so, we make the significance of any particular digit dependent upon the column where it falls.

In the above example, we have 3 ten-thousands (or 30,000), 2 thousands (2000), 7 hundreds (700), 6 tens (60), and 7 ones (7). To arrive at the actual value of the number we are trying to represent, we simply add them all together:

30000 + 2000 + 700 + 60 + 7 = 32767

We can add in either direction.

The binary number system works in exactly the same way, but since we have fewer different digits available, we must move to the left into another "place" much more frequently as the number we are trying to represent grows larger. Instead of the decimal 1-10-100-1000 sequence, the sequence of places in the binary number system (again from right to left) runs like this:

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, etc.

The mathematically inclined among you will recognize these as the powers of two. And, just as with the powers-of-ten decimal system, you can represent any number with additive combinations from the binary sequence. For example, the binary number

101101

is equivalent to the sum of 1 thirty-two, no sixteens, 1 eight, 1 four, no twos, and 1 one:

32 + 8 + 4 + 1 = 45.

To use the chmod command, however, we need not go even this deep into number theory, as long as we remember the following binary-decimal equivalences:

1 = binary 1
2 = binary 2

3 = binary (2+1)
4 = binary 4
5 = binary (4+1)
6 = binary (4+2)
7 = binary (4+2+1)

Now, let's marry binary counting with the permission bits;

Binary numbers	4	2	1
Permission bits	r	w	x

The permission bits are turned on or off by entering a value equal to the binary value with respect to the position of the permission bit. For example, if you want the execute (x) bit turned on, enter a 1 for the one binary position corresponding to the execute position for the individual, group member, or all users. If you want the execute and the write (w) bits turned on, the total value of their binary positions is 2 plus 1, or 3. To turn just the write bit on, its value with respect to its binary position is 2.

There is no reason that the originator of the permissions organized the code rwx. It could have just as easily been set up wxr. The only significance of the order of the permission bits and the binary numbers are their relative positions.

Each group of three permission bits forms a separate numeric value or column—like the standard hundreds, tens, and ones columns. For example, if you wanted to turn all of the permission bits on, you would enter 777 with the chmod command:

4	2	1	4	2	1	4	2	1
r	w	x	r	w	x	r	w	x
	7			7			7	
	(hundreds)			(tens)			(ones)	

Figure 12-1 graphically illustrates the chmod command.

- Screen A shows the chmod 777 file2 command line entered, which will turn all of the permission bits on. Every user on the system will be able to read the file, change the file (write), and execute the file (assuming that file1 contains a program).
- Screen B shows the ls -1 command entered to check the status of the permission bits for file2.
- Screen C indicates that the three sets of read, write, and execute permission bits have been turned on.
- Screen D shows the chmod 666 file2 command line entered, which will set only the read and write permission bits on. The execute bits will be turned off.
- Screen E shows the ls -l command entered to check the status of the permission bits for file2.
- Screen F indicates that only the three sets of read and write bits have been turned on.

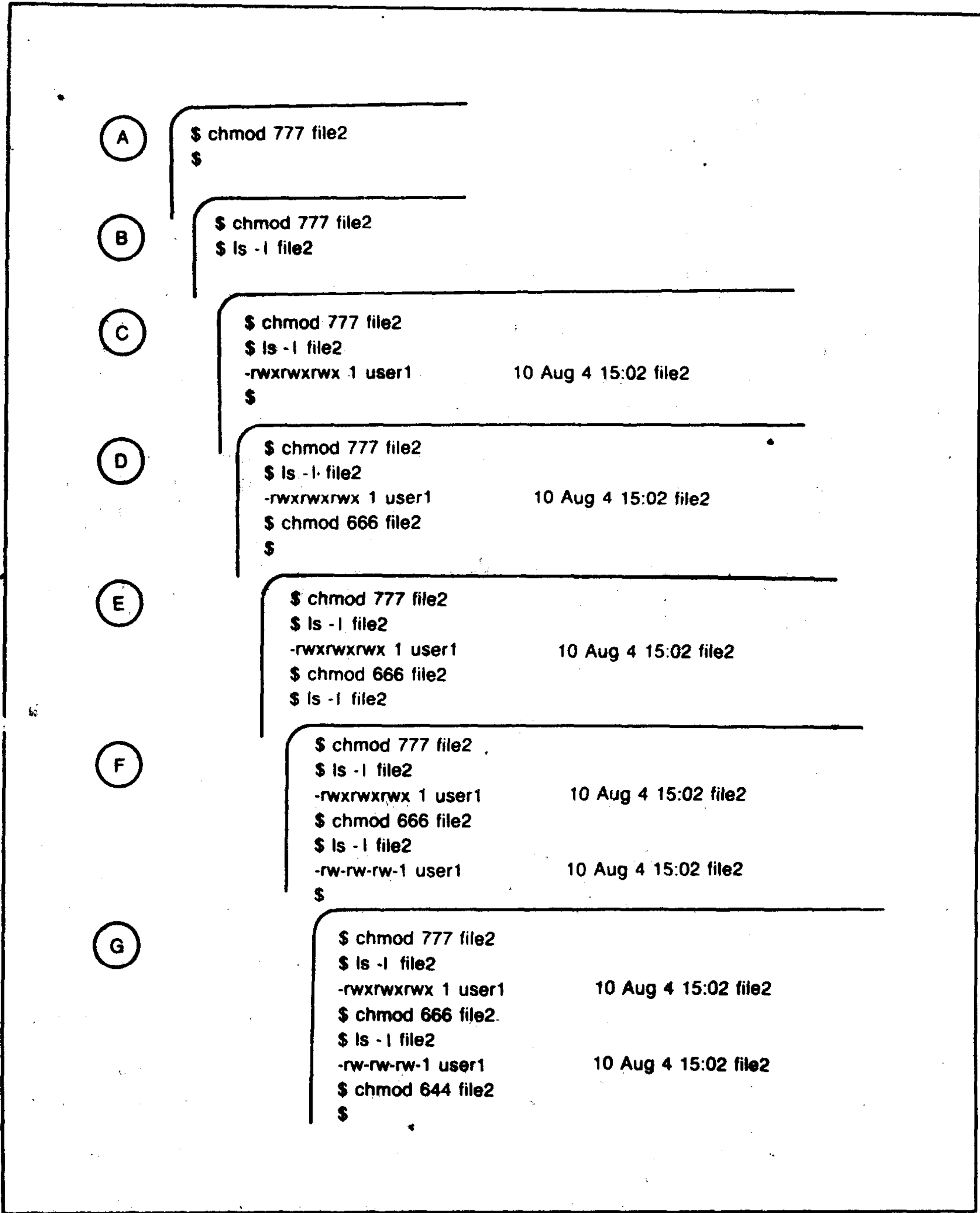


Fig. 12-1. Changing permission bits with the chmod command.

- Screen G shows the **chmod 644 file2** command line entered, which will turn on the read and write permission bits for the owner of the **file2**, and the read bits for the group and other system users.

The **chmod** sequence continues in Fig. 12-2.

- Screen A shows the **ls -l** command entered to check the status of the permission bits for **file2**.
- Screen B indicates that the appropriate permission bits have been turned on.

FILE COPYING PERMISSION

In Chapter 11 we explained how to copy and move files. You saw that it was possible to copy and move files between directories belonging to the same owner. Based on these exercises, you may have surmised that it also must be possible to move files between different users' directories.

The only thing that is different about moving files between different users' directories, or that will affect your moving files in this way, is the status of the permission bits. For example, user1 cannot copy one of user2's files if the read permission bit for all users (or group members if user1 and user2 are in the same group) on the file is not turned on. User2 cannot send a copy of one of his files to user1 if the write permission bit for user1's directory is turned off to all users (or group individuals, if user1 and user2 are members of the same group).

After changing the appropriate read, write, and/or execute permission bits for the files and directories of **user1** and **user2**, we will move a file from **user1** to **user2**.

(A)

```
$ chmod 777 file2
$ ls -l file2
-rwxrwxrwx 1 user1          10 Aug 4 15:02 file2
$ chmod 666 file2
$ ls -l file2
-rw-rw-rw- 1 user1          10 Aug 4 15:02 file2
$ chmod 644 file2
$ ls -l file2
```

(B)

```
$ chmod 777 file2
$ ls -l file 2
-rwxrwxrwx 1 user1          10 Aug 4 15:02 file2
$ chmod 666 file2
$ ls -l file2
-rw-rw-rw- 1 user1          10 Aug 4 15:02 file2
$ chmod 644 file2
$ ls -l file2
-rw-r- - r- - 1 user1       10 Aug 4 15:02 file2
$
```

Fig. 12-2. Continuation of the chmod sequence begun in Fig. 12-1.

User2 copies (pulls) a file from user1. The permission bit settings required for user2 to copy a file from user1, should be a **444** on user1's file to be copied. (All read bits are on; the setting on the owner bits will not affect user2 copying the file.) A **440** could be used if user2 and user1 are members of the same group. A **404** could be used if user2 was not a member of the same group as user1. If they were members of the same group, a **404** setting would deny user2 read access to user1's files, even though they would be open to the public.

User1 sends (pushes) a copy of a file to user2. The permission bit settings required for user1 to send a file to user2's directory should be at least a **222** on user2's directory. (All write bits are on; the setting on the owner bits will not affect user1's ability to write in user2's directory.) This would allow user1 to have write permission into user2's directory. A **202** would be acceptable if user2 is not in the same group as user1. If user2 is in the same group, permission to access the directory would be denied.

Copying a File From Another User's Directory. User1 will copy user2's file1 and name it file6 in his directory (Fig. 12-3).

- Screen A shows the cp /usr/user2/file1 file6 command entered to copy file1 from user2's directory.
- Screen B shows the ls -1 command entered to check to see if the file was transferred.
- Screen C indicates that file6 appears in user1's directory.

Sending a File to Another User's Directory. User1 will send a copy of his file2 to user2, and it will remain named file2 in user2's directory (Fig. 12-4).

- Screen A shows the cd command entered to make sure that the working directory is our home directory /usr/user1.
- Screen B shows the cp file2 /usr/user2/file2 command line entered transferring a copy of file2 to user2. We could also have entered the command, simply:

```
cp file2 /usr/user2
```

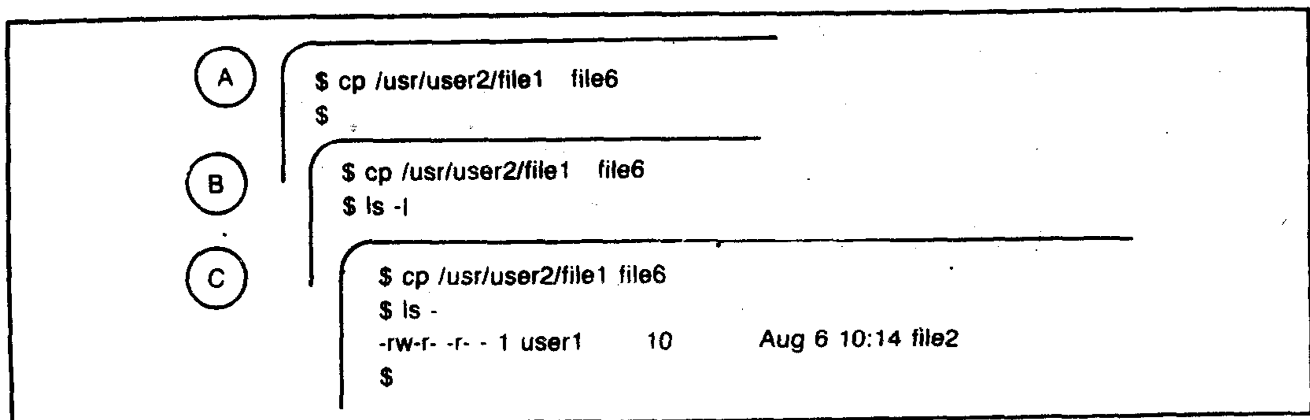


Fig. 12-3. If the permission bits allow, a file may be copied from another user's directory.

```
(A)  $ cd
     $

(B)  $ cd
     $ cp file2 /usr/user2/file2
     $

(C)  $ cd
     $ cp file2 /usr/user2/file2
     $ ls -l / usr/user2

(D)  $ cd
     $ ls -l /usr/user2
     -rw-r- -r- -    1 user2    10   Aug  6   10:14   file1
     -rw-r- -r- -    1 user1    10   Aug  7   10:18   file2
     $
```

Fig. 12-4. A file also may be sent to another user's directory.

and Unix would have created a **file2** as part of the copy command file transfer transaction. If user2 already had a **file2** in his or her directory, an error message would have been displayed indicating that Unix could not make the transfer.

- Screen C shows the **ls -l /usr/user2** command line entered to check if **file2** was copied to the directory **user2**.
- Screen D indicates that **file2** is indeed listed in the directory **user2**.

CHANGING FILE OWNERSHIP

Another form of file security is the ownership of a file. Only the owner of a file (or the system administrator) may change the permission bits on a file, remove the file, etc. When you transfer files between users, the ownership of a file has to be changed to that of the new owner if the new owner is to be able to have control of the file (copy). Ownership may be changed as part of the transfer, or it may be changed by the system administrator.

In our example (Fig. 12-3), when you "pull" a file from another user into your file, part of this transaction automatically changes the ownership of the file copy. In Fig. 12-4, when you send a file to another user, this transaction does not automatically change the ownership to the new user. In this case, you would have to have the system administrator change the ownership of the file with the **chown** command. Now examine Fig. 12-5.

A

```
$ ls -l
```

B

```
$ ls -l
-rw-r- -r- -  1 user1     10 Aug  4  15:02  file2
-rw-r- -r- -  1 user1     12 Aug  4  15:04  file3
-rw-r- -r- -  1 user1     16 Aug  4  15:06  file4
-rw-r- -r- -  1 user1     54 Aug  4  15:08  dog
-rw-r- -r- -  1 user1     10 Aug  4  15:10  file5
drwxr-xr-x 1 user1        20 Aug  4  15:14  dir1
drwxr-xr-x 1 user1        25 Aug  4  15:20  dir2
```

C

```
$ ls -l
-rw-r- -r- -  1 user1     10 Aug  4  15:02  file2
-rw-r- -r- -  1 user1     12 Aug  4  15:04  file3
-rw-r- -r- -  1 user1     16 Aug  4  15:06  file4
-rw-r- -r- -  1 user1     54 Aug  4  15:08  dog
-rw-r- -r- -  1 user1     10 Aug  4  15:10  file5
drwxr-xr-x 1 user1        20 Aug  4  15:14  dir1
drwxr-xr-x 1 user1        25 Aug  4  15:20  dir2
$ ls -l /usr/user2
```

D

```
$ ls -l
-rw-r- -r- - 1 user1      10 Aug  4  15:02 file2
-rw-r- -r- - 1 user1      12 Aug  4  15:04 file3
-rw-r- -r- - 1 user1      16 Aug  4  15:06 file4
-rw-r- -r- - 1 user1      54 Aug  4  15:08 dog
-rw-r- -r- - 1 user1      10 Aug  4  15:10 file5
drwxr-xr-x 1 user1       20 Aug  4  15:14 dir1
drwxr-xr-x 1 user1       25 Aug  4  15:20 dir2
$ ls -l /usr/user2
-rw-r- -r- - 1 user2      10 Aug  6  10:14 file1
-rw-r- -r- - 1 user1      10 Aug  7  10:18 file2
$
```

Fig. 12-5. Determining the ownership of a file sent to another user's directory.

- Screen A shows the **ls -l** command entered so that we can list the contents of our home directory and look at the ownership associated with our files.
- Screen B shows the ownership of all of user1's files belonging to user1.
- Screen C shows the **ls -l /usr/user2** command entered to list the contents of user2's directory (to which we sent a copy of **file2**).
- Screen D indicates that the **file2** that we created in user2's directory is still owned by user1.

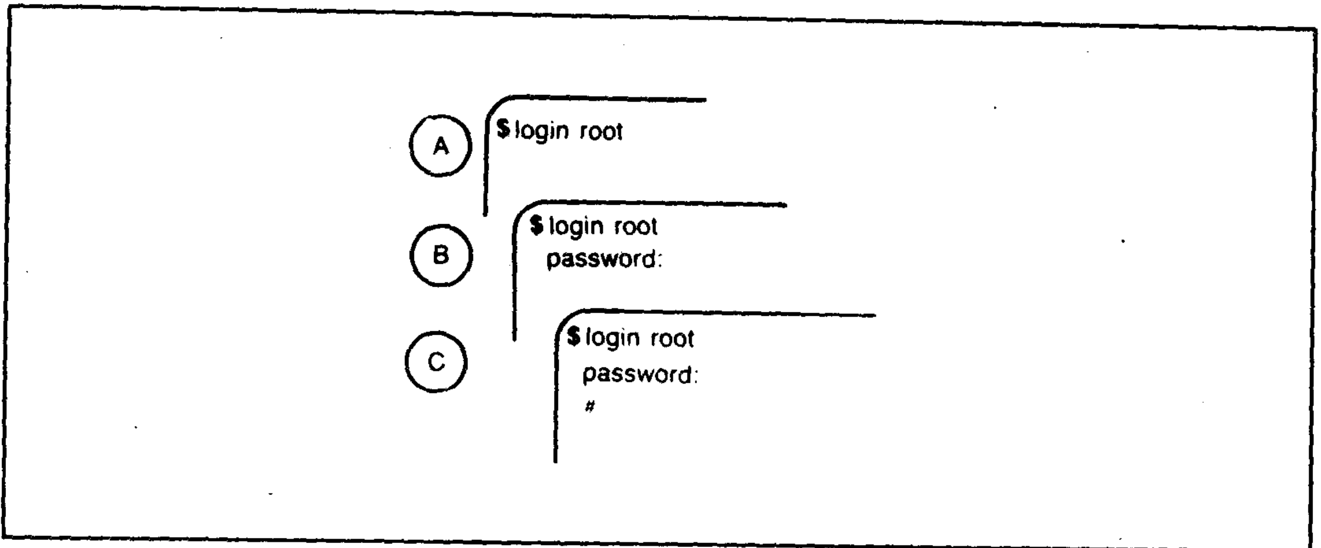


Fig. 12-6. Logging on as a super-user, the system administrator.

```
A   # chown user2 /usr/user2/file2
    #

B   # chown user2 /usr/user2/file2
    # ls -l

C   # chown user2 /usr/user2/file2
    # ls -l
    -rw-r- -r- -   1   user2   10   Aug 6   10:14   file1
    -rw-r- -r- -   1   user2   10   Aug 7   10:18   file2
    #

D   # chown user2 /usr/user2/file2
    # ls -l
    -rw-r- -r- -   1   user2   10   Aug 6   10:14   file1
    -rw-r- -r- -   1   user2   10   Aug 7   10:18   file2
    # login user1
    password:
    $
```

Fig. 12-7. Changing the ownership of a file with the chown *command*.

If you are the system administrator (probably the microcomputer owner) you must log on to the system as "root" or as the system administrator in order to change the ownership of user2's file2 (Fig. 12-6).

- Screen A shows the login root command entered to log on the system as root.
- Screen B displays the password: prompt. (When you enter the password, it will not be displayed.) As the system administrator, we have access to the password.
- Screen C displays the shell # prompt, indicating that we are logged on as the root user (super-user).

The format for the chown command is:

chown   new owner's login name   file name

Figure 12-7 illustrates super-user use of the chown command.

- Screen A shows the chown user2 /usr/user2/file2 command entered by the system administrator.
- Screen B shows the ls -l /usr/user2 command entered to check if the ownership of file2 has been changed.
- Screen C indicates that the ownership of user2's file2 has been changed from user1 to user2. (Compare with Fig. 12-4, Screen D.)
- Screen D shows the login user1 command entered to log on as user1 again.

To read another user's file you must have the appropriate read permission. To write in or modify another user's file, the file must have the appropriate write permission. Another user's directory, in which there is a file you wish to copy, need not have the appropriate permission for you to access that directory, in order for you to gain access to any file in the directory that has the appropriate permission settings.

In a similar manner, you may send a file to any directory or subdirectory belonging to another user, as long as *that* directory has the appropriate access permission.

Each file (and directory) must be individually secured. Securing the directory above a file or a subdirectory does not secure the subordinate files and directories. When the operating system creates a new file the usual permission status is:

Directory	755	drwxr-xr-x
File	644	-rw-r--r--

Chapter 13

Linking Files

In the preceding lab exercises we have discussed making directories and files, changing the working directory, listing the directories and files in a directory, examining the contents of files, manipulating the files in the hierarchical file structure.

In this lab exercise we will learn how to link one or more file names to a single file, in essence making an alias name or names for a file. This Unix system feature is useful from two standpoints:

- You can give any of the Unix commands an alias name.
- You can give another user access to any of your files without requiring them to enter a pathname to your file, and without having to make a copy of your file (which would consume additional disk storage space.

Figure 3-13 presented a model of file linking. You will note that many file names can be linked to a single file. We are telling the operating system to link a new file name to a block on the disk that already contains a file, instead of creating an actual file at another block location on the disk.

The format for the link command (**ln**) is:

command existing file name alias or linked file name

(A)

```
$ cd
$
```

(B)

```
$ cd
$ ln  /usr/user2/file1 file6
$
```

(C)

```
$ cd
$ ln  /usr/user2/file1 file6
$ ls -l
```

(D)

```
$ cd
$ ln  /usr/user2/file1 file6
$ ls -l
-rw-r- -r- -   1  user1      10 Aug  4  15:02 file2
-rw-r- -r- -   1  user1      12 Aug  4  15:04 file3
-rw-r- -r- -   1  user1      16 Aug  4  15:06 file4
-rw-r- -r- -   1  user1      54 Aug  4  15:08 dog
-rw-r- -r- -   1  user1      10 Aug  4  15:10 file5
drwxr-xr-x   1  user1      20 Aug  4  15:14 dir1
drwxr-xr-x   1  user1      25 Aug  4  15:20 dir2
-rw-r- -r- -   2  user2      10 Aug  6  10:18 file6
$
```

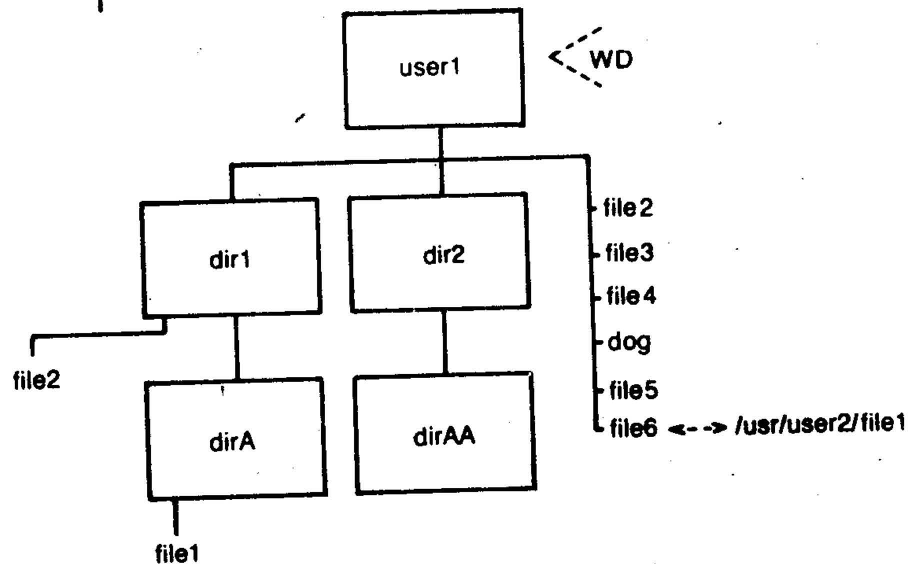


Fig. 13-1. An example of linking files with the link (ln) command.

The first exercise we will perform with the link command is to create a new file name in our home directory that is an alias for an existing file. Examine Fig. 13-1.

- Screen A shows the cd command entered to make sure our home directory is the working directory.
- Screen B shows the ln file /usr/user2/file1 file6 command entered to make a file named file6, which will have the same disk block address as user2's named file6, which will have the same disk block address as user2's file1.
- Screen C shows the ls -l command entered to check if there is a file6 in our directory.
- Screen D indicates that there is a file6 now listed.

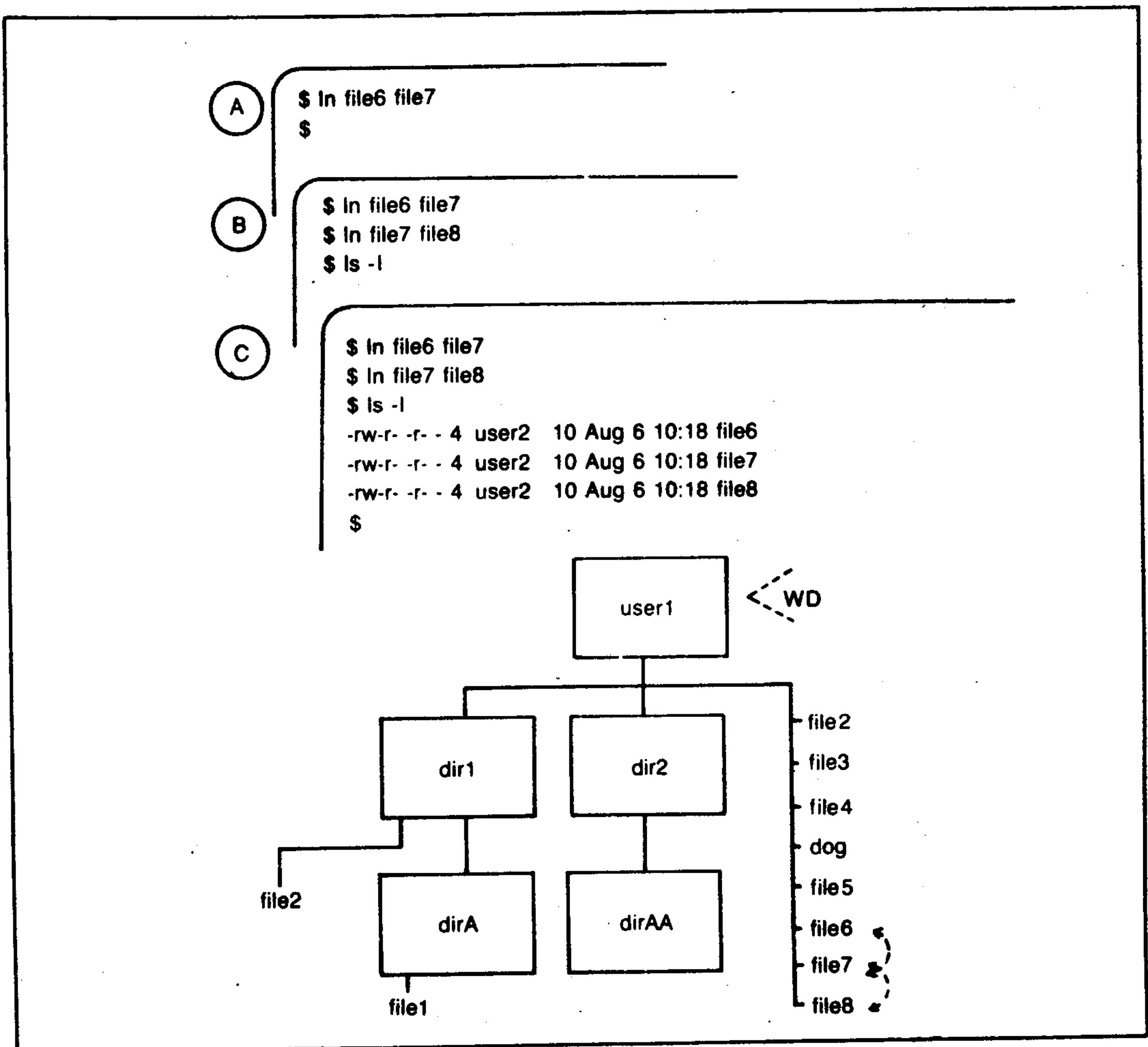


Fig. 13-2. Creating multiple links.

You might note that the owner of file6 is user2; if we perform an ls -l listings of user2's directory, you will see that both lines are exactly the same, with the exception that the file name in user1's directory is called file6. Also note that the number of links is now two instead of one. Now look at Fig. 13-2.

- Screen A shows the ln file6 file7 command line entered to see if we can create another link to user2's file through the link we just made.
- Screen B While we are making links, let's also make a link from file7 to file8.
- Screen C confirms that the files file7 and file8 were created. (In the interest of conserving text pages, we will not continue to list the complete listing of files and directories.)

Actually, the results above should not have been too surprising. The data describing the statistics on file1 were merely copied over into user1 directory from user2's directory (disregarding the file name, which is superfluous). If we made links all day long, we would still be copying the same statistical data about user2's file1.

There are a couple of items to take note of at this point. First, all of the lines are exactly the same, the only exceptions being the difference in the file names, themselves. Second, notice that the number of links has now gone from two to four.

The question that now comes to mind is, "What happens if we alter one of the linked file names?" Change permission bits of /usr/user2/file1 to 666 first, and then examine Fig. 13-3.

- Screen A shows the echo command and appended redirection entered to add Hi there to file8.
- Screen B shows the ls -l command entered to look at the status of the files now.
- Screen C indicates that the number of bytes contained in all of the files has grown from 10 to 19, and the date on the files has changed from Aug 6 10:18 to Aug 7 11:15.

(A)
```
$ echo "Hi there" >>  file8
$
```

(B)
```
$ echo "Hi there" >>  file8
$ ls -l
```

(C)
```
$ echo "Hi there" >>  file8
$ ls -l
-rw-rw- -rw- - 4 user2  19 Aug 7 11:15 file6
-rw-rw- -rw- - 4 user2  19 Aug 7 11:15 file7
-rw-rw- -rw- - 4 user2  19 Aug 7 11:15 file8
$
```

Fig. 13-3. Can we alter the file name of a linked file?

```
(Operator logged on as the system administrator)

(A)  # chown user1 file7
     #

(B)  # chown user 1 file7
     # ls -l usr/user1

(C)  # chown user1 file7
     # ls -l usr/user1
     -rw-rw- -rw- - 4  user1      9 Aug  7  11:15 file6
     -rw-rw- -rw- - 4  user1     19 Aug  7  11:15 file7
     -rw-rw- -rw- - 4  user1     19 Aug  7  11:15 file8
     #
```

Fig. 13-4. The file owner's permission bit settings override any links to the file that may already exist.

If you are skeptical about the actual contents of the file, use the command:

cat file6 file7 file8 /usr/user2/file1

to display the contents of all of these files. We must use the full pathname to access user2's **file1** from user1's directory. (Technically, this command is displaying the contents of **file1** four times.)

File1 has read permission for group and others; user1 therefore can access **file1**. However, user1 is in no way a partner in the ownership of **file1**. If user2 chooses to reset the permission bits, he could deny user1 read and/or write (and execute if it were a program) access at any time. The fact that links had been made before user2 changed the permission bits has no bearing on the matter.

For our next exercise (Fig. 13-4), let's ask the system administrator to change the ownership of the file from user2 to user1.

- Screen A shows the **chown user1 file7** command entered to change the ownership of the file.
- Screen B shows the **ls -l** command entered to check the ownership of the file. (Remember, we are operating as the administrator, so we have to enter a full pathname in order to list the contents of user1's directory.)
- Screen C indicates that the ownership of all of the files has been changed.

Note that if you do not have permission to read—i.e., the permission bits are not set to allow you to read, write, and/or execute a file—having a link to another user's file in your directory will still not afford you any additional access privileges than if you had tried to access the file with **cat** or another command.

Chapter 14

Using Shell Metacharacters

In Chapter 11 we discussed making files with the redirect output command. The redirect output > and appended redirect output >> command were demonstrated as the means to direct (or redirect) the output of a Unix command to a file instead of displaying the output on your terminal screen. (We just as easily could have used the redirect output command to redirect the output of one of your computer programs to a file). These examples were designed to provide you with a basic demonstration of the capabilities available in the Unix system shell's metacharacters for controlling or manipulating the flow of data on a command line.

In this session we will discuss using some additional metacharacters:

- The redirect input command (<) is used for directing input to a command.
- The semicolon command separator (;) is used for entering several commands on a command line.
- The pipe (|) command is used for channeling the output from one command or program directly into another.
- The **tee** command is used for channeling the output from one command directly into another and into a file or display it on the terminal monitor

REDIRECT INPUT

The < is called the *redirect input command* because it redirects the input from a data file to a command, instead of to the terminal keyboard. For simplicity, however, let's think of this command in a positive sense. It is a command for

performing a service, rather than a command that alters the normal operation of another command, i.e., displaying the output of the command on the terminal display.

The redirect input command < works in an identical fashion to the redirect output > command, but in reverse. Instead of directing the output to a file, < is used to direct the data already in a file into a utility or a computer program for processing, where normally the input to a command or a program is obtained from the terminal keyboard. This is particularly useful when the information to be processed is a large data file that would be time-consuming to enter, data that is already in a file in memory or which can be compiled with another command, and/or data that is to be used (reused) in several applications.

The format for the direct input command is

command < Data file name

and its use is shown in Fig. 14-1.

- Screen A shows the command **cat < file2** entered.
- Screen B displays the contents of **file2**.

This command line will display the contents of **file2** on the terminal screen. In practice, we would enter the command line simply as **cat file2** to display the contents of a file. This exaggerated format, using redirected input, demonstrates that the **cat** command would also take input from the redirect input command as well. The redirect input command is more often used in conjunction with other commands in more complex command lines.

MULTIPLE-COMMAND COMMAND LINES

In all of our examples thus far, we have limited our command lines to a single command statement, i.e., a single command with its necessary options and arguments. Now we will discuss entering more than one command on a line.

The semicolon (;) is used to delineate the end of each in a series of command statements on a command line. Thus, with the semicolon you can string together as many command statements on a command line as you desire, before beginning the processing of the command line by pressing the Return key. See Fig. 14-2.

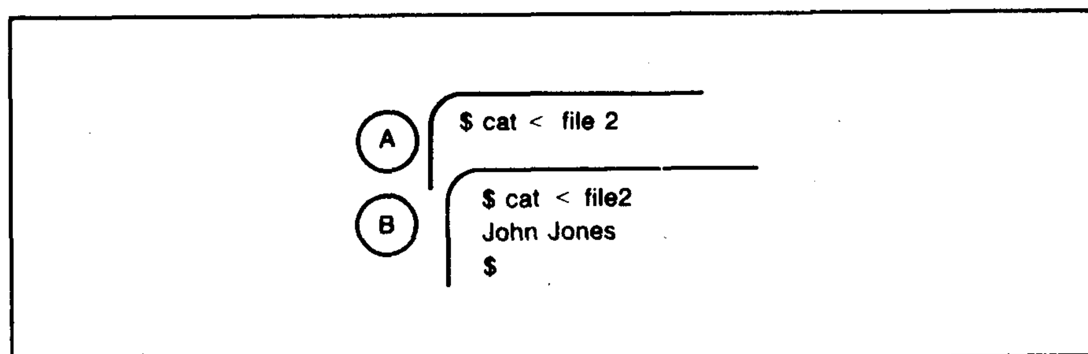


Fig. 14-1. Accessing the data in a file with the redirect-input metacharacter.

```
(A) $ pwd; cd /; pwd; ls; cd; pwd

(B) $ pwd; cd /; pwd; ls; cd; pwd
/usr/user1
/
add.hd    boot.fd   fd        lib           priboot   xenix
bin       dev       fptutor   load.hd       tmp       xenix.fd
boot      etc       install   lost+found    usr       usr
/usr/user1
$
```

Fig. 14-2. Entering a multiple-command command line.

- Screen A shows a series of commands entered:

  pwd; cd /; pwd; ls; cd; pwd

- Screen B displays the sequence of data concurrent with the commands that we entered in 4A.

This command printed the name of the current directory, changed the current working directory to the root / directory, printed the name of the current working directory, listed the contents of the current working directory, changed the current directory back to our home directory, and printed the name of the current working directory to confirm that we are back to our home directory.

PIPES

The pipe is used in a similar fashion as the redirect output command, with the exception that it is used to direct (channel) data from (or generated by) one command into the following command or program.

Several pipes can be used in a single command line to channel data through several commands. This is called a *pipe line*. When the types of commands through which data is being passed and processed are of the type that sort or in some other way manipulates the data in the file, the command line is referred to as a *filter*.

Some examples of the command line formats using the pipe are:

command | device	**ls | lpr** Send the output of **ls** to the printer.
command | command	**cat file2 | wc** Send the output of file2 to another utility called "word count" (**wc**), which will provide a tabulation of the number of words in the file.

command | computer program	ls | your.program If, for example, you had a special sort program, you could channel the output of the ls command to your program the same way as you would to a utility.
computer program | computer program	your.prog1 | your.prog2 Just as the pipe can channel data between two Unix commands, it can also channel data between two programs.

Now examine Fig. 14-3.

- Screen A shows the command ls -l /usr entered. This command will list the users in the system. (The /usr directory contains the names of the home directories of all of the registered users.)
- Screen B shows the display.

The problem here, which we can't reproduce on the printed page, is that the display went by so fast that you could not see the first files listed. Figure 14-4 provides a possible answer.

- Screen A shows that the cure for this problem is the **more** command mentioned earlier, used here in a slightly different way. When it was introduced I said that you could use it to display the contents of a file instead of using the **cat** command—which, if you remember, gave us the same problem of displaying large files too fast. The only problem is that the **more** command will not list the files in a directory. However, we can use the **ls -l** command to list the files, and then channel the output of the **ls -l** command to the **more** command:

  ls -l /usr | more

- Screen B indicates that the display of the ls -l /usr command is now under control. The display halted when the screen was filled. To continue, press the space bar. This will instruct more to display another screen full of data.

Figure 14-5 illustrates an alternate output for data.

- Screen A shows that the pipe can also be used to channel output from a command to the printer spooler (name **lpr**) for a printout on your printer. Enter **ls -l /usr | lpr**.
- Screen B shows the terminal display—nothing but the **$** prompt! The output went to the printer for a printout.
- Printout C simulates the hard copy.

(A)

```
$ ls -l /usr
```

(B)

```
total 28
drwxrwxrwx  2  bin     128  Jul  28  11:22  adm
drwxrwxrwx  2  altos    48  Apr  17  15:28  altos
drwxr-xr-x  2  bin    1408  Oct  12  18:43  bin
drwxrwxrwx  3  bin     128  Jul  28  11:40  dict
drwxrwxrwx  4  root    640  Jul  28  11:55  include
drwxr-xr-x  2  john     96  Oct  12  17:44  john
drwxr-xr-x 11  lee     400  Oct  24  09:35  lee
drwxrwxrwx 13  bin     592  Oct  12  18:42  lib
drwxrwxrwx  2  root     48  Jul  28  11:5...
drwxrwxrwx 10  root    160  Jul  ...             spool
drwxrwxrwx  3  sys     ...              11:55  src
drwxrwxrwx  2  sys     ...          28  11:55  sys
drwxrwxrwx  2  ...     272  Oct  24  10:36  tmp
drwxrwx...             128  Oct  24  04:18  unix
dr...          user     64  Apr  17  15:29  user
...xrwx     2  user1   112  Oct  24  04:07  user1
drwxrwxrwx  2  user2    80  Oct  25  13:57  user2
drwxrwxrwx  2  user3    48  May  25  22:52  user3
drwxrwxrwx  2  user4    48  May  25  22:52  user4
drwxrwxrwx  2  user5    48  May  25  22:52  user5
drwxrwxrwx  2  user6    48  May  25  22:52  user6
drwxrwxrwx  2  user7    48  May  25  22:52  user7
drwxrwxrwx  2  user8    48  May  25  22:53  user8
drwxr-xr-x  3  vin     112  Oct  18  02:36  vin
$
```

Fig. 14-3. Examining the contents of the /usr directory.

Now, let's look at a pipe used in a different context (Fig. 14-6).

- Screen A shows the command **ls /usr | wc -l** entered. This combination of commands will figure out how many users are registered on the system.
- Screen B indicates that there are 24 users. The way we got this answer is based on the fact that each user is represented by a single line in the **/usr** directory. The command **wc -l** is used to count the number of lines generated by the **ls -l** command. There are 24 lines, so there must be 24 users.

Pipe Lines

As we discussed earlier, a pipe line is a series of commands connected with pipes. The following screens (Fig. 14-7) demonstrate some examples of pipe lines.

Z89.6.54

(A)

```
$ ls -l /usr | more
```

(B)

```
$ ls -l /usr | more
total 28
drwxrwxrwx 2  bin    128  Jul 28 11:22 adm
drwxrwxrwx 2  altos   48  Apr 17 15:28 altos
drwxr-xr-x 2  bin   1408  Oct 12 18:43 bin
drwxrwxrwx 3  bin    128  Jul 28 11:40 dict
drwxrwxrwx 4  root   640  Jul 28 11:55 include
drwxr-xr-x 2  john    96  Oct 12 17:44 john
drwxr-xr-x 11 lee    400  Oct 24 09:35 lee
drwxrwxrwx13  bin    592  Oct 12 18:42 lib
drwxrwxrwx 2  root    48  Jul 28 11:55 preserve
drwxrwxrwx10  root   160  Jul 28 11:55 spool
drwxrwxrwx 3  sys     48  Jul 28 11:55 src
drwxrwxrwx 2  sys     48  Jul 28 11:55 sys
drwxrwxrwx 2  root   272  Oct 24 10:36 tmp
drwxrwxrwx 3  unix   128  Oct 24 04:18 unix
drwxrwxrwx 2  user    64  Apr 17 15:29 user
drwxrwxrwx 2  user1  112  Oct 24 04:07 user1
drwxrwxrwx 2  user2   80  Oct 25 13:57 user2
drwxrwxrwx 2  user3   48  May 25 22:52 user3
drwxrwxrwx 2  user4   48  May 25 22:52 user4
drwxrwxrwx 2  user5   48  May 25 22:52 user5
- -MORE- -
```

```
drwxr-xr-x 2  bin   1408  Oct 12 18:43 bin
drwxrwxrwx 3  bin    128  Jul 28 11:40 dict
drwxrwxrwx 4  root   640  Jul 28 11:55 include
drwxr-xr-x 2  john    96  Oct 12 17:44 john
drwxr-xr-x 11 lee    400  Oct 24 09:35 lee
drwxrwxrwx13  bin    592  Oct 12 18:42 lib
drwxrwxrwx 2  root    48  Jul 28 11:55 preserve
drwxrwxrwx10  root   160  Jul 28 11:55 spool
drwxrwxrwx 3  sys     48  Jul 28 11:55 src
drwxrwxrwx 2  sys     48  Jul 28 11:55 sys
drwxrwxrwx 2  root   272  Oct 24 10:36 tmp
drwxrwxrwx 3  unix   128  Oct 24 04:18 unix
drwxrwxrwx 2  user    64  Apr 17 15:29 user
drwxrwxrwx 2  user1  112  Oct 24 04:07 user1
drwxrwxrwx 2  user2   80  Oct 25 13:57 user2
drwxrwxrwx 2  user3   48  May 25 22:52 user3
drwxrwxrwx 2  user4   48  May 25 22:52 user4
drwxrwxrwx 2  user5   48  May 25 22:52 user5
drwxrwxrwx 2  user6   48  May 25 22:52 user6
drwxrwxrwx 2  user7   48  May 25 22:52 user7
drwxrwxrwx 2  user8   48  May 25 22:53 user8
drwxr-xr-x 3  vin    112  Oct 18 02:38 vin
$
```

Fig. 14-4. Piping the output of ls -l through the more command to control the screen display.

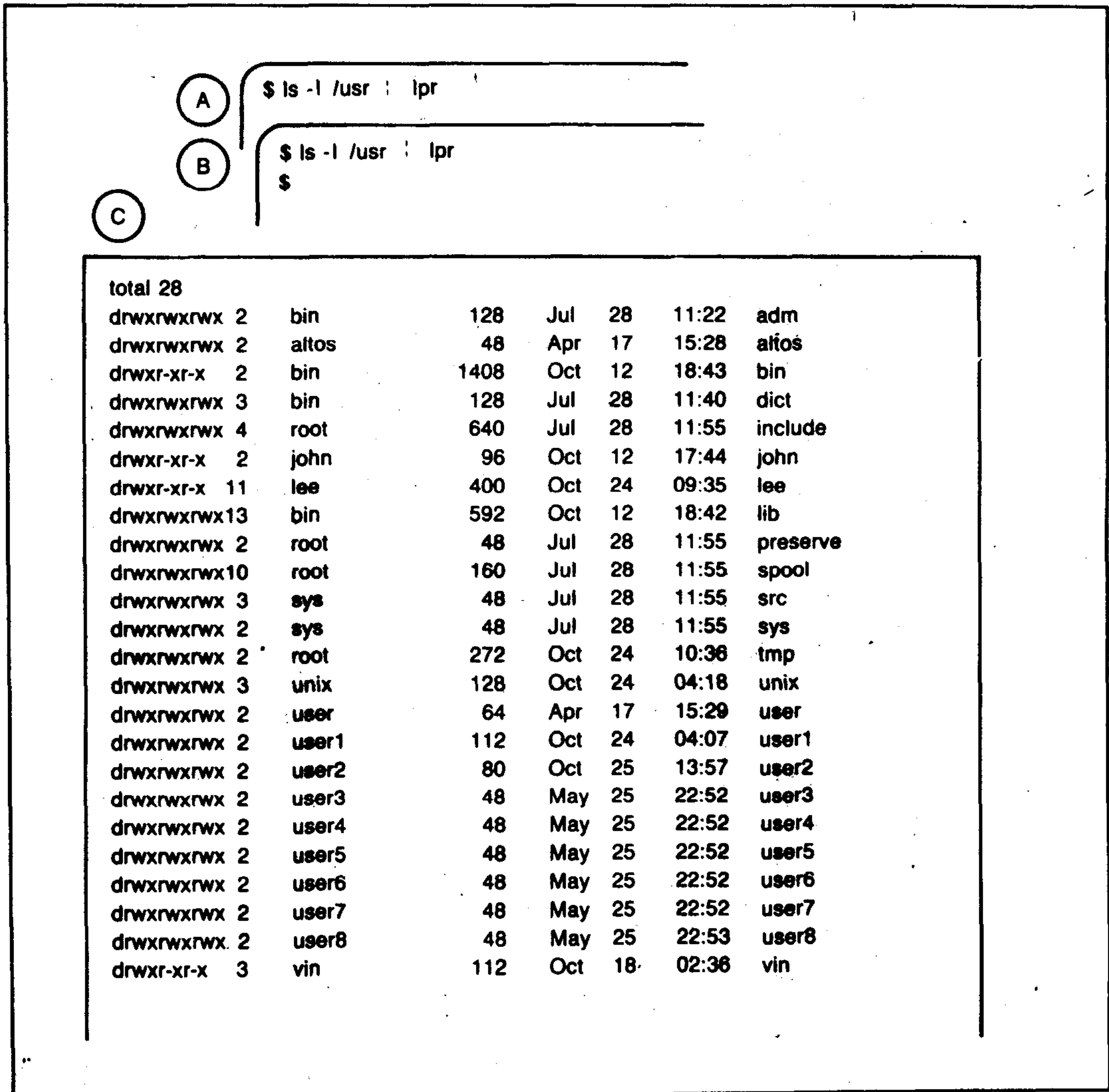


Fig. 14-5. Command output may be piped to a peripheral device, in this case the line printer.

- Screen A shows how we can make the command, which in Fig. 14-5 Screen B told us how many users were listed in the /usr directory, and print out the results by using the pipe to channel the output to the printer. Enter:

  ls -l /usr | wc -l | lpr

- Printout B simulates a hard copy of the information. The number is again 24.

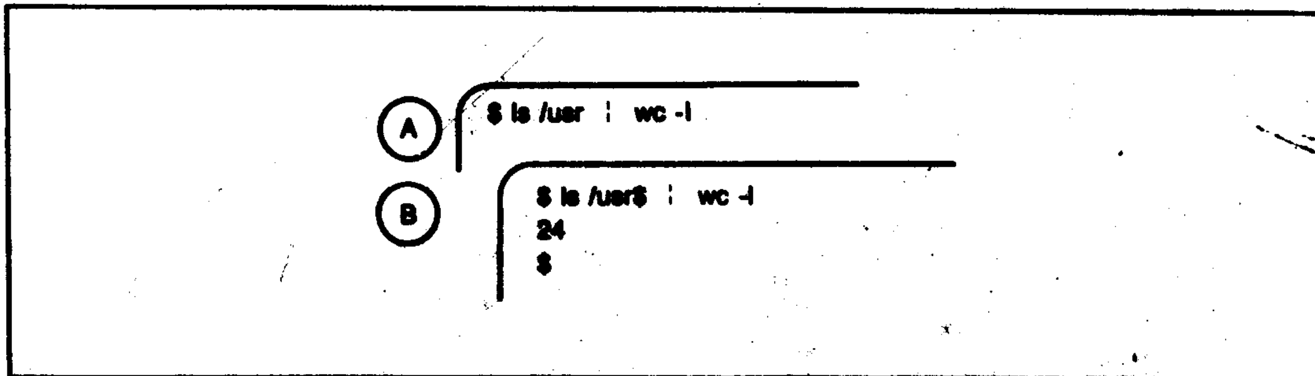


Fig. 14-6. Piping the output of ls -l through the word count utility to determine the number of users.

Filters

The preceding command/output sequences lead naturally to a discussion of filters. Examine Fig. 14-8.

- Screen A shows the ls -l /usr | wc -l command entered to determine the number of users registered. Unfortunately, that 24 also includes some system files, so we do not really have 24 users. To resolve this problem we can use a command that will "filter" out the names of files that are not users' files.

I listed all of the students files numerically: user1, user2, etc. Therefore, all I have to do is tell the shell to look for only those files that have user in the name. The command for this is:

```
ls /usr | grep "user*"
```

grep is a command that will act like a filter. The asterisk after user is a *wild card* character, meaning that user plus any and all characters following will be acceptable.

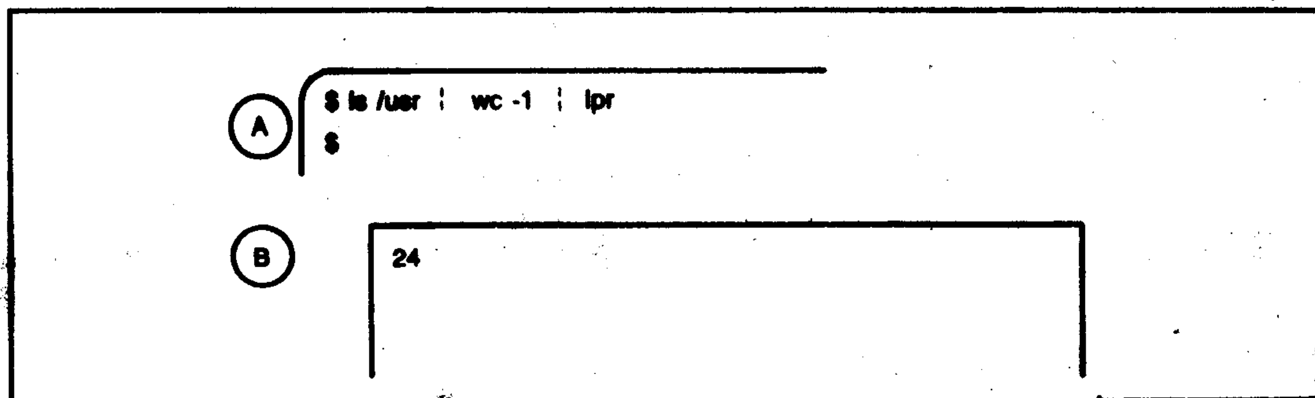


Fig. 14-7. The output of the sequence in Fig. 14-6 can then be piped to the printer. Multiple pipes in a command sequence is called a pipe line.

A

```
$ ls -l /usr | wc -l
24
$ ls -l /usr | grep "user*"
```

B

```
$ ls -l /usr | wx -l
24
$ ls -l /usr | grep "user*"
drwxrwxrwx 2 user     64  Apr 17  15:29  user
drwxrwxrwx 2 user1   112  Oct 24  04:07  user1
drwxrwxrwx 2 user2    80  Oct 25  13:57  user2
drwxrwxrwx 2 user3    48  May 25  22:52  user3
drwxrwxrwx 2 user4    48  May 25  22:52  user4
drwxrwxrwx 2 user5    48  May 25  22:52  user5
drwxrwxrwx 2 user6    48  May 25  22:52  user6
drwxrwxrwx 2 user7    48  May 25  22:52  user7
drwxrwxrwx 2 user8    48  May 25  22:53  user8
$
```

C

```
$ ls -l /usr | wc -l
24
$ ls -l /usr | grep "user*"
drwxrwxrwx 2 user     64  Apr 17  15:29  user
drwxrwxrwx 2 user1   112  Oct 24  04:07  user1
drwxrwxrwx 2 user2    80  Oct 25  13:57  user2
drwxrwxrwx 2 user3    48  May 25  22:52  user3
drwxrwxrwx 2 user4    48  May 25  22:52  user4
drwxrwxrwx 2 user5    48  May 25  22:52  user5
drwxrwxrwx 2 user6    48  May 25  22:52  user6
drwxrwxrwx 2 user7    48  May 25  22:52  user7
drwxrwxrwx 2 user8    48  May 25  22:53  user8
$ ls -l /usr   grep "user*"   wc -l
```

D

```
$ ls -l /usr | wc -l
24
$ ls -l /usr | grep "user*"
drwxrwxrwx 2 user     64  Apr 17  15:29  user
drwxrwxrwx 2 user1   112  Oct 24  04:07  user1
drwxrwxrwx 2 user2    80  Oct 25  13:57  user2
drwxrwxrwx 2 user3    48  May 25  22:52  user3
drwxrwxrwx 2 user4    48  May 25  22:52  user4
drwxrwxrwx 2 user5    48  May 25  22:52  user5
drwxrwxrwx 2 user6    48  May 25  22:52  user6
drwxrwxrwx 2 user7    48  May 25  22:52  user7
drwxrwxrwx 2 user8    48  May 25  22:53  user8
$ ls -l /usr   grep "user*"   wc -l
9
$
```

Fig. 14-8. Piping to a command that can be used to filter the output.

- Screen B is a listing of the files that meet the conditions of the search.
- Screen C shows the means I used to avoid counting the lines. I channel the listing to **wc -l** to count them for me, as follows:

```
ls -l /usr | grep "user*" | wc -l
```

- Screen D now indicates that there are only nine users.

Now, just for practice, let's run the **ls -l /usr** through another filter called the **sort** command (Fig. 14-9), and use it to display the list of files in reverse order.

- Screen A shows the **ls -l /usr | sort -r | more** command line entered.
- Screen B shows the same listing of files as before, only in reverse order.

The Tee Command

The **tee** command is similar to the pipe, except that it is used to direct the output of a command simultaneously to a file as well as to your terminal monitor.

The command line format for the **tee** command is:

```
command | tee file name
```

The pipe is used to connect the **sort** and **tee** commands, and a file name is required as one of the standard outputs of the tee command. The other standard output is your terminal screen. The output to your terminal can also be redirected with another pipe. Now examine Fig. 14-10.

- Screen A shows the command:

```
ls /usr | sort -r | tee file8
```

- Screen B lists the files in /**usr** is reverse order.
- Screen C shows the command **cat file8** entered to check the file to see if the **tee** command sent the data to the file as well as displayed it on the terminal.

Let's do one more thing in this sequence of operations (Fig. 14-11).

- Screen A shows the command:

```
ls /usr | sort -r | tee file7 | wc -l
```

- Screen B displays a 24, but no listing of files. The reason is that the normal output of the **tee** command to the terminal was channeled to the **wc -l** command. The **24** represents the output of the **wc -l** command.

(A)

```
$ ls -l /usr  ¦  sort -r ¦ more
```

(B)

```
$ ls -l /usr  ¦  sort -r ¦ more
total 28
drwxrwxrwx13    bin      592   Oct   12   18:42   lib
drwxrwxrwx10    root     160   Jul   28   11:55   spool
drwxrwxrwx 4    root     640   Jul   28   11:55   include
drwxrwxrwx 3    unix     128   Oct   24   04:18   unix
drwxrwxrwx 3    sys       48   Jul   28   11:55   src
drwxrwxrwx 3    bin      128   Jul   28   11:40   dict
drwxrwxrwx 2    user8     48   May   25   22:53   user8
drwxrwxrwx 2    user7     48   May   25   22:52   user7
drwxrwxrwx 2    user6     48   May   25   22:52   user6
drwxrwxrwx 2    iser5     48   May   25   22:52   user5
drwxrwxrwx 2    user4     48   May   25   22:52   user4
drwxrwxrwx 2    user3     48   May   25   22:52   user3
drwxrwxrwx 2    user2     80   Oct   25   13:57   user2
drwxrwxrwx 2    user1    112   Oct   24   04:07   user1
drwxrwxrwx 2    user      64   Apr   17   15:29   user
drwxrwxrwx 2    sys       48   Jul   28   11:55   sys
drwxrwxrwx 2    root     272   Oct   25   15:56   tmp
drwxrwxrwx 2    root      48   Jul   28   11:55   preserve
drwxrwxrwx 2    bin      128   Jul   28   11:22   adm
drwxrwxrwx 2    altos     48   Apr   17   15:28   altos
 - -MORE- -
```

```
drwxrwxrwx 4    root     640   Jul   28   11:55   include
drwxrwxrwx 3    unix     128   Oct   24   04:18   unix
drwxrwxrwx 3    sys       48   Jul   28   11:55   src
drwxrwxrwx 3    bin      128   Jul   28   11:40   dict
drwxrwxrwx 2    user8     48   May   25   22:53   user8
drwxrwxrwx 2    user7     48   May   25   22:52   user7
drwxrwxrwx 2    user6     48   May   25   22:52   user6
drwxrwxrwx 2    user5     48   May   25   22:52   user5
drwxrwxrwx 2    user4     48   May   25   22:52   user4
drwxrwxrwx 2    user3     48   May   25   22:52   user3
drwxrwxrwx 2    user2     80   Oct   25   13:57   user2
drwxrwxrwx 2    user1    112   Oct   24   04:07   user1
drwxrwxrwx 2    user      64   Apr   17   15:29   user
drwxrwxrwx 2    sys       48   Jul   28   11:55   sys
drwxrwxrwx 2    root     272   Oct   25   15:57   tmp
drwxrwxrwx 2    root      48   Jul   28   11:55   preserve
drwxrwxrwx 2    bin      128   Jul   28   11:22   adm
drwxrwxrwx 2    altos     48   Apr   17   15:28   altos
drwxr-xr-x 11   lee      400   Oct   24   09:35   lee
drwxr-xr-x  3   vin      112   Oct   18   02:36   vin
drwxr-xr-x  2   john      96   Oct   12   17:44   john
drwxr-xr-x  2   bin     1408   Oct   12   18:43   bin
$
```

Fig. 14-9. Piping through a filter (sort) to the display-control utility more.

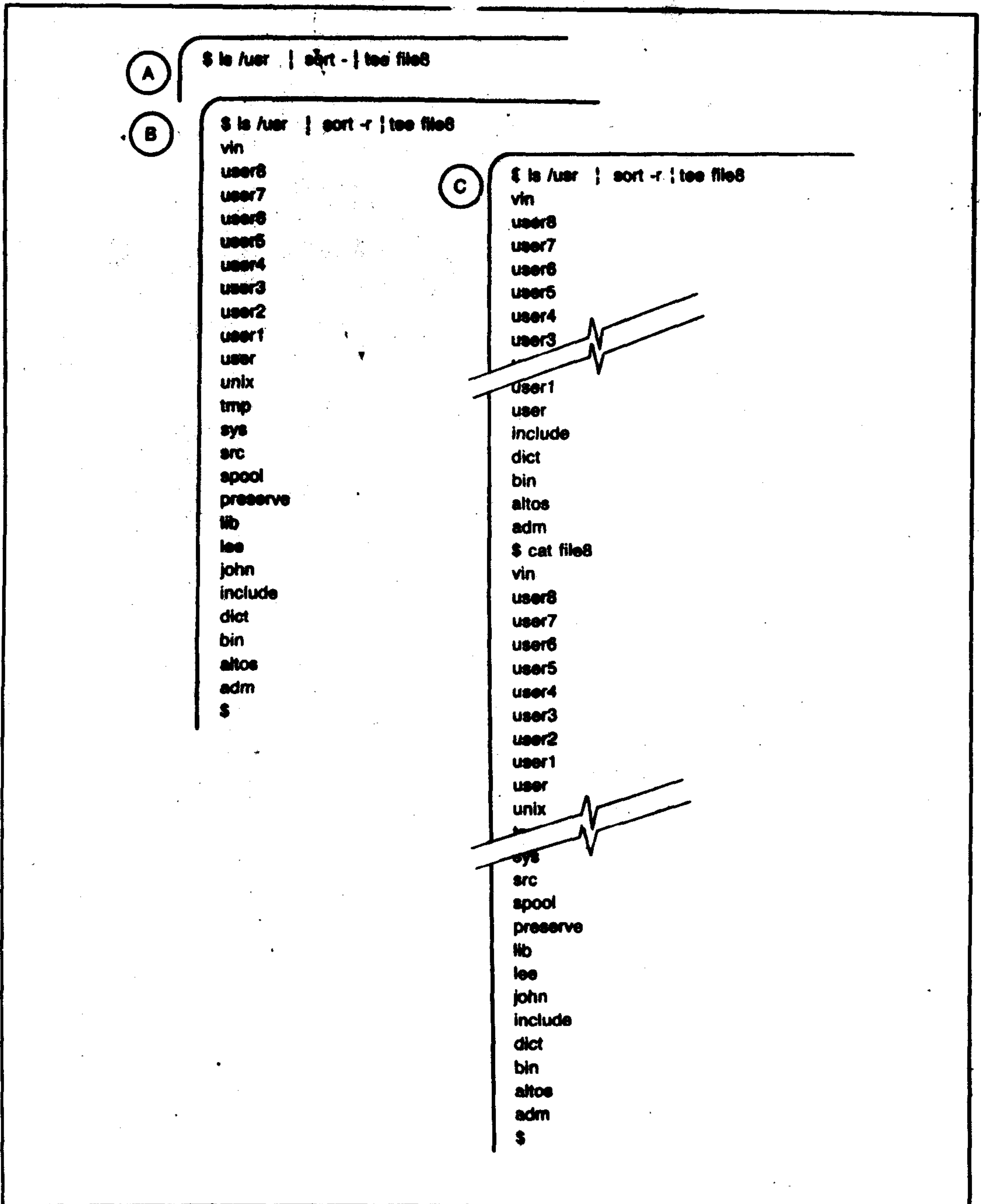


Fig. 14-10. The tee may be used to route output to multiple destinations, in this case to the terminal and file8.

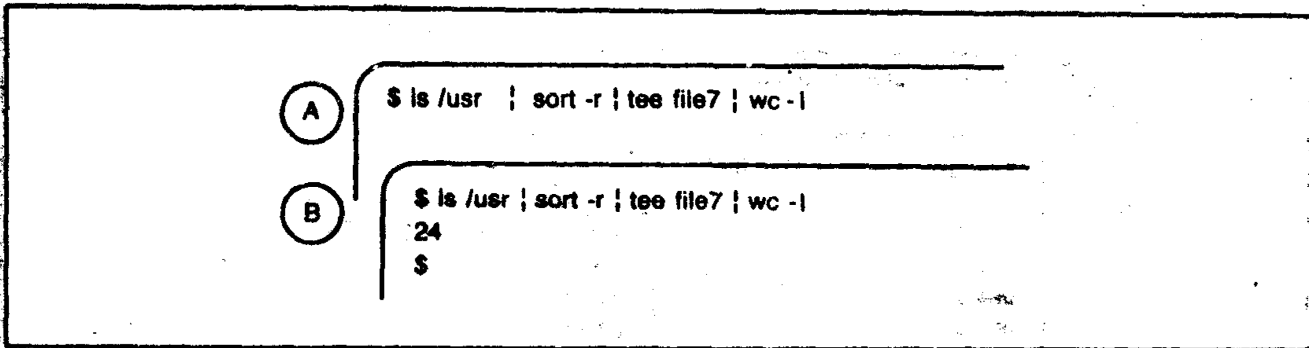


Fig. 14-11. The output of a tee can be piped.

If you wanted to, you could include several **tee** commands in a command line. The following is an example of a workable command:

ls /usr ¦ tee save1 ¦ sort -r ¦ tee save2 ¦ wc -l ¦ tee save3

The number 24 is displayed, and three new files were formed: **save1**, **save2**, and **save3**. **Save1** should contain the output of the **ls** command, **save2** should be the output of the **sort** command, and save3 should be the number 24, the output of the **tee** command.

SUMMARY

The purpose of the exercises presented in this lab session is to show you that you can use some of the commands contained in the shell program to simulate a computer program, i.e., a series of instruction combined to perform some desired task. In Chapter 15 we will go one step further and save the sequence of commands in a file, enabling us to run this sequence of commands over and over again without having to re-enter the sequence of commands. Saving a sequence of simple commands in a file will serve as an introduction to creating simple shell programs.

Chapter 15

Making and Using Shell Programs

In Chapter 14 we showed you how to join commands together, in essence creating a small, very simple computer program. In this lab session we will show you how to:

- Save programs in a file to make shell programs.
- Change the permission mode bits on the file to make the file containing the shell program executable.
- Run a shell program as if it were a command.

To start, let's make a command that will demonstrate how you can make a short program to combine the output of two Unix commands and give you a custom version of their services. The two commands we will use will be the **pwd** and **ls -l** commands. What we want our command (let's name it simply **li**) to do is print out the name of a directory and then do an **ls -l** listing of its contents. Fig. 15-1 shows the sequence.

- Screen A shows another usage of the **cat** command for creating a new file:

```
cat > li
```

- Screen B shows our **cat > li** command line, but the shell has not redisplayed a prompt. What is actually happening here is that the shell is waiting for you to create some text to put into our new file **li**.

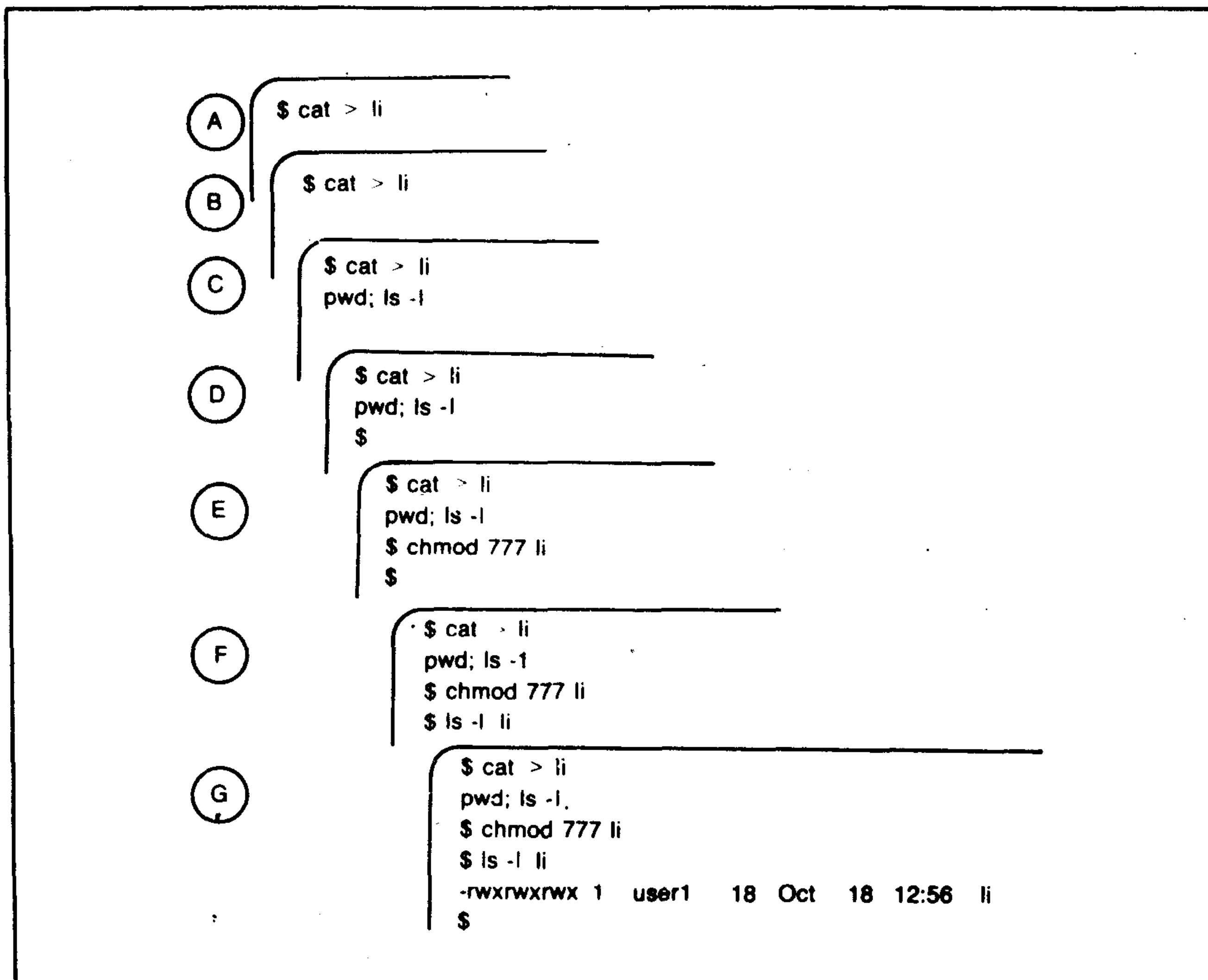


Fig. 15-1. User-created utilities are examples of shell programs.

- Screen C shows the command line we are entering into our file **li**:

  pwd; ls -l

  To end the text input to our file, enter a ^d. (There is no display for the ^d input.)
- Screen D shows the shell displaying the shell prompt.
- Screen E shows the command chmod 777 li entered. The 777 will make our file li an executable file, i.e., a command. (We could just as easily have turned on only the owner's execute mode bit.)
- Screen F shows the **ls -l li** command entered to check if the permission mode bits have been properly changed.
- Screen G shows that all of the x (execute) permission mode bits have been turned on. Our command li is now ready to be used by the owner, a user in your group, and anyone in the system.

USING SHELL VARIABLES

Simple shell programs can be as simple as our opening example. Complex shell programs can be as complex as if we were using one of the high-level programming languages to program some task. The nice thing about shell programs is that users not familiar with the logic involved in writing programs can learn to write shell programs, since shell programs can be simply a series of commands.

In our opening example we used our shell program to perform its function for only a single directory. We could make our li command more flexible by using something called *shell variables.* Using shell variables will take you back to your high school algebra days. You may remember something like:

Solve this problem for c:

$a = 1$
$b = 2$
$c = a + b$

The answer is $1 + 2 = 3$; $c = 3$.

First, let's add the change directory command to our previous li command example, as follows:

```
cat  >  li
cd;  pwd;  ls  -l
^d
```

The **cat** > li will cause the new program to "overwrite" (replace) the present contents of the li command with this new command. Examine Fig. 15-2.

- Screen A shows the appropriate entries.
- Screen B shows our li command entered. Now, every time that you enter li you will be moved to your home directory, have it confirmed, and print out the contents of its directory.
- Screen C shows a way to make our li command a more general application command. We need to be able somehow to make the command more flexible. To do this we can add a shell variable ($1) as follows:

```
cat > li
cd $1; pwd; ls -l
^d
```

- Screen D shows the li /usr command line entered.

When you execute our li command and include the name of the directory to which you wish to move (just as you would provide an argument to the **cd** command), the **$1** shell variable in our li command will read the first argument on the command line (**/usr**), replace the **$1** with **/usr**, and execute the command. The command will make the specified directory the working directory, print the name of the working directory, and list the contents of this directory.

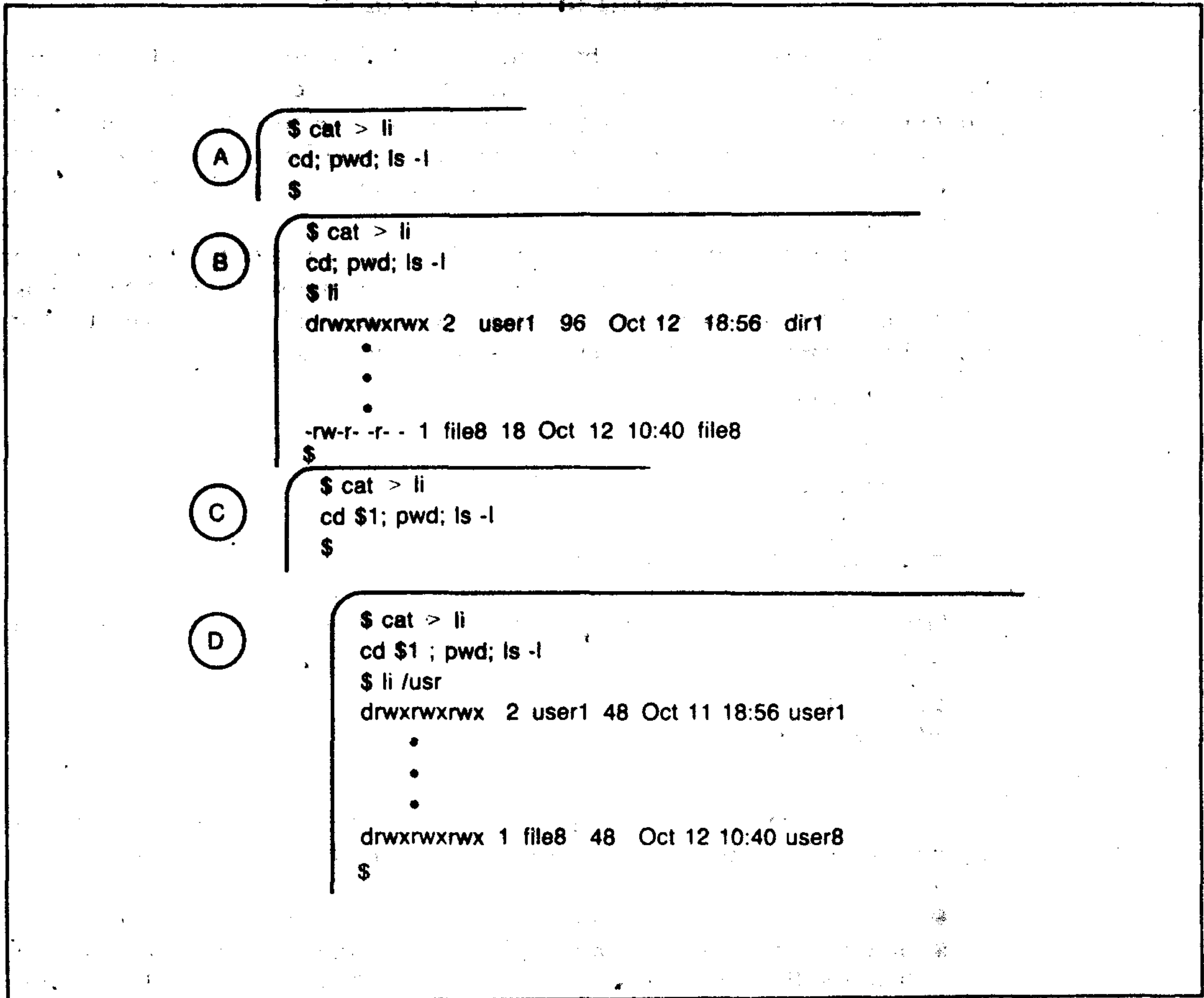


Fig. 15-2. Using a shell variable to make the li utility more general.

Our next example (Fig. 15-3) will employ several shell variables in a shell program.

- Screen A shows the entry of the shell program:

```
cat > li
echo This is the first variable $1
echo This is the fourth variable $4
echo This is the second variable $2
echo This is the 6890th variable $6890
^d
```

- Screen B shows the **li 1 5 2356 129809** command line entered. (The command was already made executable.)

- Screen C shows the output:

```
This is the first variable $1 1
This is the fourth variable $4 129809
This is the second variable $2 5
```

(nothing in the fourth line)

This presents several points of interest about our use of shell variables:

- $1 will look for the first argument after the command, $2 will look for the second, etc.
- If you have an entry and no shell variable in your program for that position, the program will ignore the missing position(s). To get the 6890th variable to be displayed here, you would have to enter 6890 variables after the command.
- If you have a shell variable number and nothing in that position, it will enter a nothing value for this shell variable number.
- You may have many shell variables in a command line or shell program.

(A)

```
$ cat > li
echo This is the first variable $1
echo This is the fourth variable $4
echo This is the second variable $2
echo This is the 6890th variable $6890
$
```

(B)

```
$ cat > li
echo This is the first variable $1
echo This is the fourth variable $4
echo This is the second variable $2
echo This is the 6890th variable $6890
$ li 1 5 2356 129809
```

(C)

```
$ cat > li
echo This is the first variable $1
echo This is the fourth variable $4
echo This is the second variable $2
echo This is the 6890th variable $6890
$ li 1 5 2356 129809
This is the first variable $1 1
This is the fourth variable $4 129809
This is the second variable $2 5

$
```

Fig. 15-3. Using several variables in a shell program.

The next thing that you might ask is, "Can I substitute values other than numeric values for a shell variable?" The answer is yes, as shown in Fig. 15-4.

- Screen A shows the entry of a new shell program:

```
cat > li
echo $1
echo $2
echo $3
^d
```

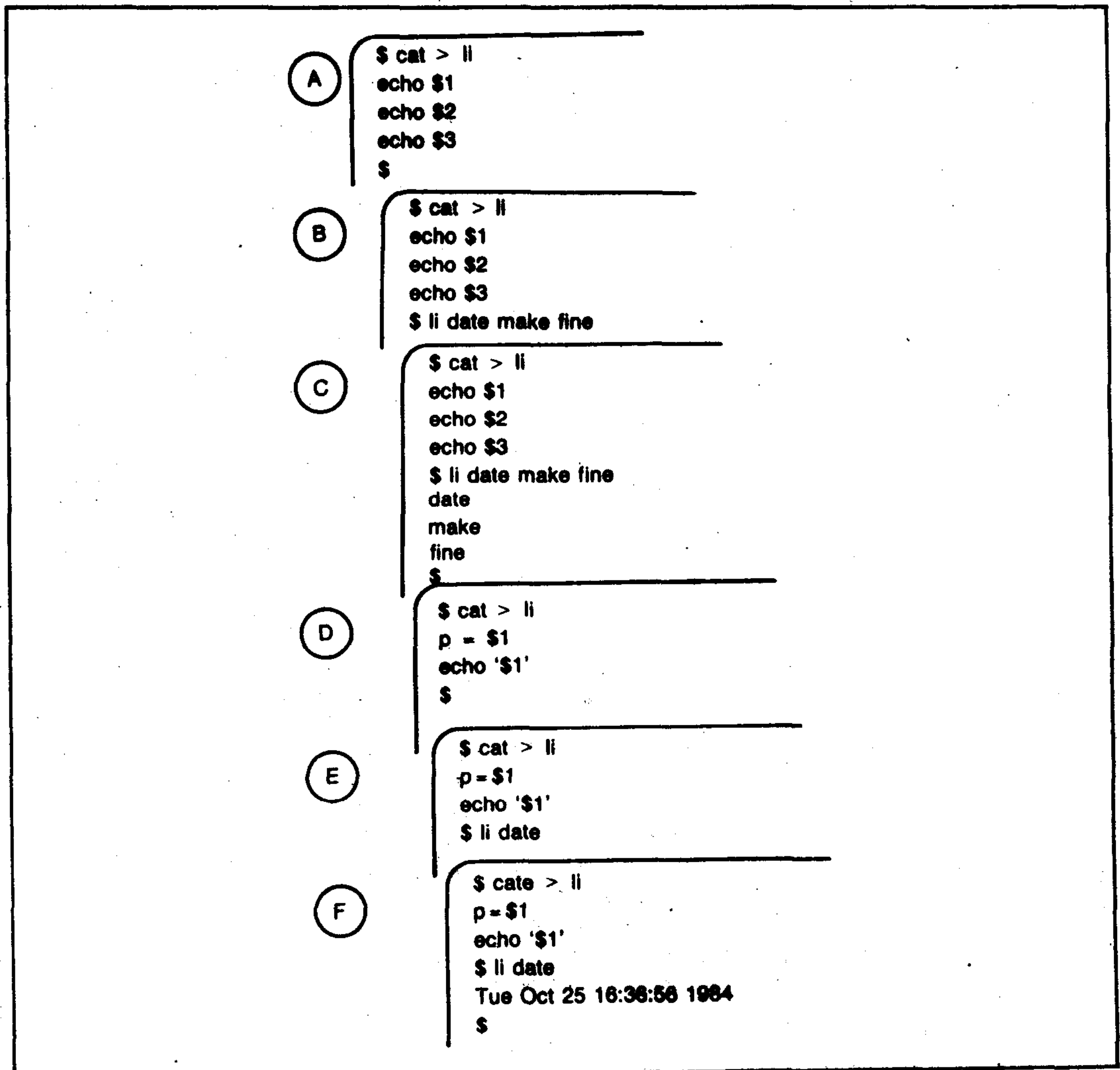


Fig. 15-4. Shell variables are not restricted to numeric data types.

- Screen B shows the command li date make fine entered.
- Screen C shows the response:

```
date
make
file
```

- Screen D shows the entry of another shell program to test whether we can substitute command names as variables and have the shell program execute them:

```
cat > li
p=$1
echo `$p`
^d
```

- Screen E shows the command li date entered.
- Screen F shows the response Tue Oct 25 16:36:51 1984.

The response in Screen F indicates that you can read in a value for a shell variable (the name of a command), set this value equal to another value, and use the value in a program statement, indicating to the shell with the (`) metacharacter that the enclosed is to be replaced with the value, in this case the output of the date command.

Kochan and Wood have an interesting shell program providing an example of the use of shell variables. They use shell variables to find names and telephone numbers in a company telephone directory. I have modified the example a bit here. The screens are shown in Fig. 15-5.

- Screen A illustrates the entry of the telephone directory. (Don't worry about a format; you can use the editor or your word processor to create a file of names and telephone numbers.)

```
cat > phone_book
Fast, Bee 4444

^d
```

- Screen B shows the procedure for looking up a name in the directory. We will use a command called grep, which is designed to look for a specified pattern of characters (a word is a specified pattern) in a file and print out the line where it occurs. The pattern, in this case, will be a person's name.

```
grep "Fast" phone_book
```

- Screen C shows the display of our line from the file named phone_book. (Had there been two people (entries) with the same name, i.e., had you

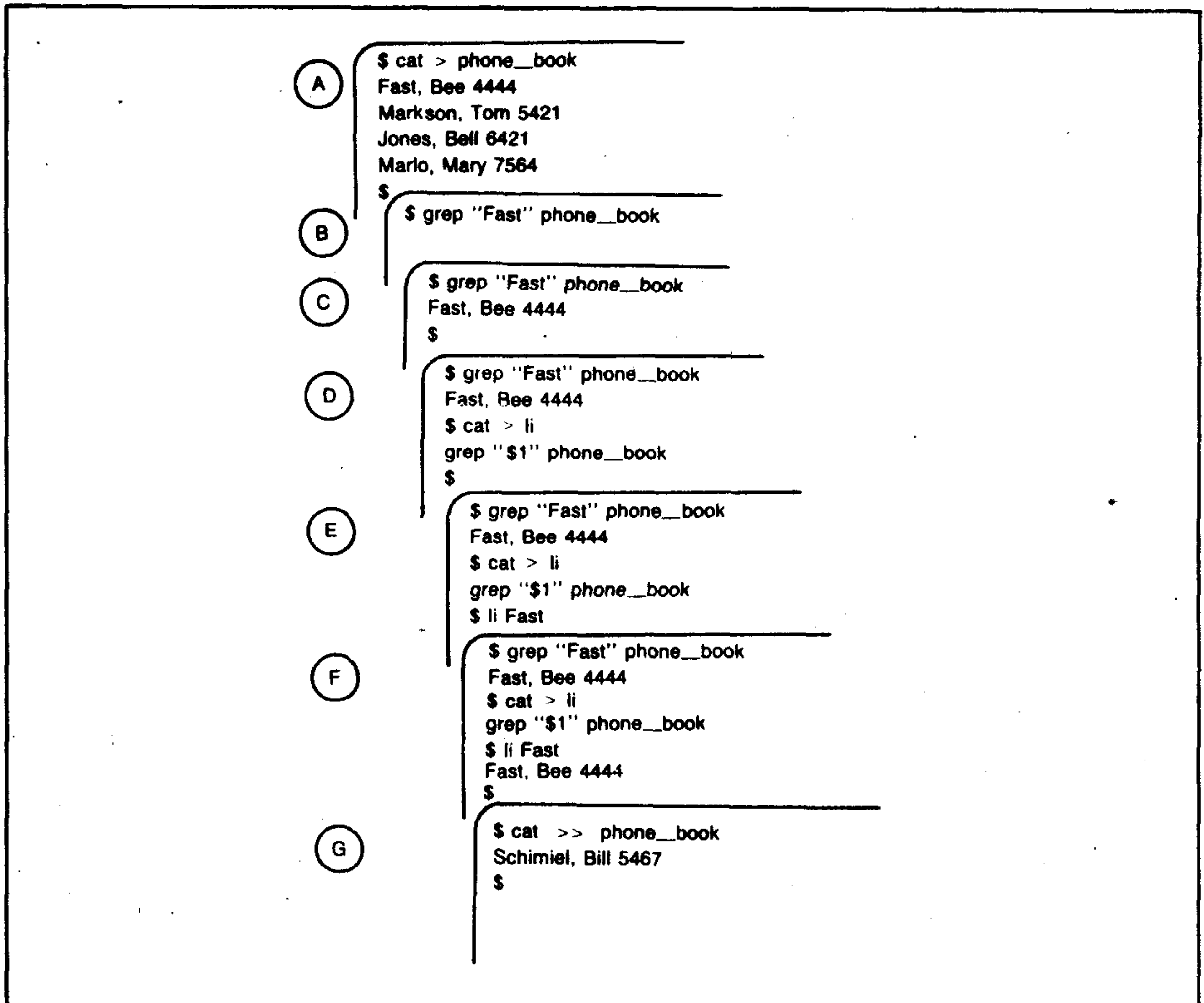


Fig. 15-5. Using shell variables in a database-like program.

entered Bob and had two Bobs been entered in the directory, the **grep** command would have displayed both lines.)

- Screen D shows the next step is to make this last command into a shell program. We will name our command li:

```
cat > li
grep "$1" phone_book
^d
```

(The quotes are used around the **$1** to emulate the command structure that we used in 2P. It has nothing to do with the shell variable **$1**.)

- Screen E shows the entry of the command li **Fast**
- Screen F indicates that our li shell program can find the specified line in the file.

- Screen G shows that, to add names to your directory, you can use the cat command with appended redirect output:

```
cat >> phone_book
Schmiel, Bill 5565
^d
```

To delete names from the directory, you will have to learn to use one of the Unix text editors (like ed), or use your word processor to add or delete names in the directory.

I will set up one last problem for you, but I will let you enter it on your own. Let's call it an inventory program. It is based on the payroll example provided in *Real World Unix*, by John Halamka.

```
cat  >  inv
part=$1
qty=$2
total=` expr  $part  \*  $qty `
echo "The value of these $1 parts at \s $2 apiece is $total"
^d
```

Here is an analysis of shell program components:

cat > inv	This is used to create a new file.
part=$1	This is a way of setting up a variable that will be used in a mathematical expression. The $1 is the shell variable command. **Part** is not programmed to look for a variable, but $1 is.
qty=$2	This is the second variable in our program. You may have as many variables in a program as you need.
total=	This is another set-up for a variable.
expr	This is a Unix command for executing a mathematical expression.
` . . . `	A pair of single back quotes causes the enclosed mathematical expression to be executed and the value inserted at this point in the program.
*	This is a tricky one. The asterisk normally is a metacharacter used for wild card matching of characters. If we want to use it as a mathematical character (the multiplication operator, in this case), we must indicate that it is not a metacharacter by preceding it with a backslash (\).
echo	This is a usage of the **echo** command that we have not yet demonstrated. You know that you can use the echo command to display a statement, but here we have also included a variable. The variable will be substituted with the value generated in the problem.
\$	The $ is preceded by a \ to indicate that it is to be read as a character and not as a shell variable.

To run this shell program, enter the file name inv, followed by the number of parts and then the dollar value of the part.

inv 40 3.43

Do not forget to make inv executable with the chmod command, or execute it as an argument to the shell program by preceding the inv with the shell command sh inv.

SHELL PROGRAMS WITH PIPES

Kaare Christian has in his book an interesting shell program using a pipe, which is presented here in slightly modified form (Fig. 15-6). It uses the who command to check which users are currently logged on the system, and with the wc -l (word count) command, converts the list of names into a number:

- Screen A shows the who : wc -l command line entered. The response is 2, indicating that there are two users currently logged on to the system.

A
```
$ who ¦ wc -l
2
$
```

B
```
$ who ¦ wc -l
2
$ who ¦  wc -l  >>  dly.usg
```

C
```
$ who  ¦  wc -l
2
$ who  ¦  wc -l  >>  dly.usg
$
```

D
```
$ who  ¦  wc -l
2
$ who  ¦  wc -l  >>  dly.usg
$ cat dly.usg
2
Tue Oct 25 16:37:18 1984
2
$
```

Fig. 15-6. Pipes may be used in shell programs.

- Screen B shows that we could add >> and a file name to this command; every time we run this command we could create a listing of the number of users on the system:

```
who | wc -l >> dly.usg
```

- Screen C shows how we could further tie this command in with the **date** function so that it would print the time, date, and then the number of users logged on:

```
date; who | wc -1 >> dly.usg
```

(A)

```
$ cat > dly
```

(B)

```
$ cat > dly
date; who ¦ wc -l >> dly.usg
$
```

(C)

```
$ cate > dly
date; who ¦ wc -l >> dly.usg
$ chmod 777 dly
$
```

(D)

```
$ cat > dly
date; who ¦ wc -l >> dly.usg
$ chmod 777 dly
$ dly
$
```

(E)

```
$ cat > dly
date; who ¦ wc -l >> dly.usg
$ chmod 777 dly
$ dly
$ cat dly.usg
```

(F)

```
$ cat > dly
date; who ¦ wc -l >> dly.usg
$ chmod 777 dly
$ dly
$ cat dly.usg
2
Tue Oct 25   16:37:18   1984
2
Tue Oct 25   16:39:20   1984
5
$
```

Fig. 15-7. Extending the daily usage example of Fig. 15-6.

- Screen D shows the cat dly.usg command entered to look at the contents of the file dly.usg. It shows the first entry we make, 2, and the second with the date and number of users.

The sequence of screens in Fig. 15-7 will demonstrate the creation of this shell program:

- Screen A shows the cat > dly command line entered to create our file.
- Screen B shows our command line for our shell program:

```
date; who | wc -l >> dly.usg
^d
```

- Screen C shows the chmod 777 dly command entered to make the file dly executable.
- Screen D shows our new command dly executed. No information will be displayed on our terminal screen, because we have redirected it to the file dly.usg

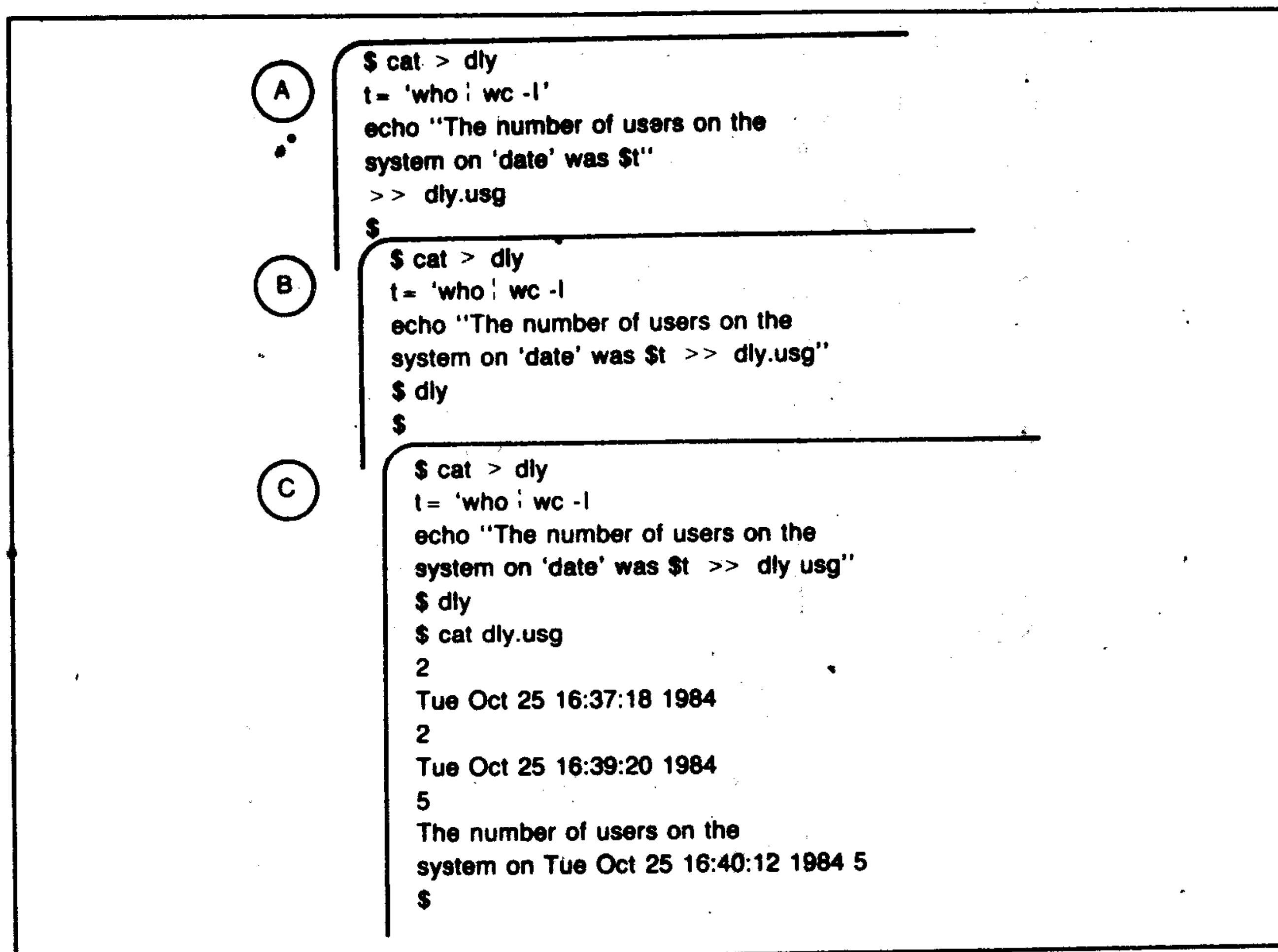


Fig. 15-8. Proving that the user-created dly utility actually works.

- Screen E shows the **cat dly.usg** command entered to read the contents of this file.
- Screen F shows the number 5. Of course, we are not sure that this is the output of our **dly** program, so let's modify the program.

Now examine Fig. 15-8.

- Screen A shows how we will modify it:

```
cat > dly
t=`who : wc -l`
echo "The number of users on the
system at `date` was $t" >> dly.usg
^d
```

- Screen B shows the entry of the command **dly**.
- Screen C shows how, on entering **cat dly.usg**, we find the contents of **dly.dsg** as conclusive evidence that our command is working.

These examples are by no means extraordinary examples of the use of shell programs, but they provide good examples of what shell programming is all about. New users are usually intimidated by the term *programming*. Now that you know what a shell program is, however, you will no longer be intimidated.

Chapter 16

Removing Files and Directories

Now that we have mucked up your file system with a lot of unnecessary files and directories demonstrating the Unix system and some of its features, it is time to clean up. However, before we call it quits, there is one final set of metacharacters to be demonstrated. These are the wild card metacharacters.

- * The asterisk is used as an all-inclusive wild card. It is used to match any number and any character.
- ? The question mark is used as a wild card for matching any single character. If you want to match two characters, you would use two question marks in the position(s) where any character will be acceptable.
- [..] The square brackets are used to designate selective characters, numbers, or range of numbers.

Figure 16-1 should represent the status of our **user1** hierarchical file structure at this time. Use it as a reference for the following file searching and file deletion exercises, which begin with Fig. 16-2.

- Screen A shows the **ls** command entered so that we can see what files we have in our directory.
- Screen B lists the contents of our **user1** home directory.
- Screen C shows the command **ls file*** entered. This demonstrates the asterisk as a wild card character indicating to the shell that you want it to list all of the files with file names beginning with the characters **file** and

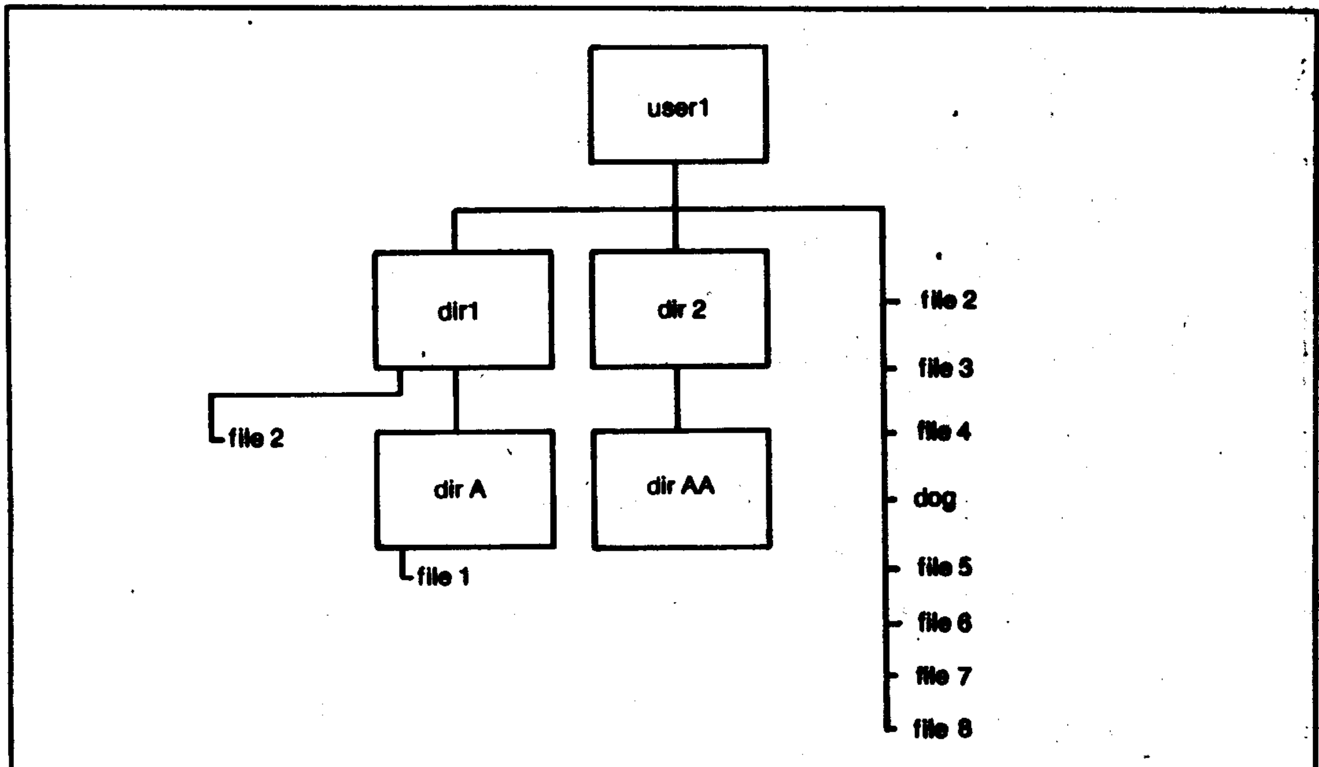


Fig. 16-1. Status of user1's directory of files.

containing any number of any additional characters (within the constraints for file naming).

- Screen D lists all of the files and only those files which have names that begin with the name **file**.
- Screen E shows the command **ls f*2** entered, to demonstrate that the asterisk can be used anywhere within the file name. In this example, the shell will list all of the files with file names beginning with an **f**, ending with a 2, and having any number of characters between the **f** and the 2.
- Screen F lists the file with the name **file2**, because it is the only file name that meets these constraints.

Figure 16-3 demonstrates the use of the ? wild card.

- Screen A shows the command **ls file?** entered. We are indicating to the shell that you want it to list all files having file name beginning with the characters **file** and ending with any single character.
- Screen B lists all of the files and only those files which have names beginning with the name **file** and containing any single additional character appended to **file**.
- Screen C shows the command **ls d*** to list all of the files beginning with the character **d** and followed by any additional characters.
- Screen D is an interesting response to this command.

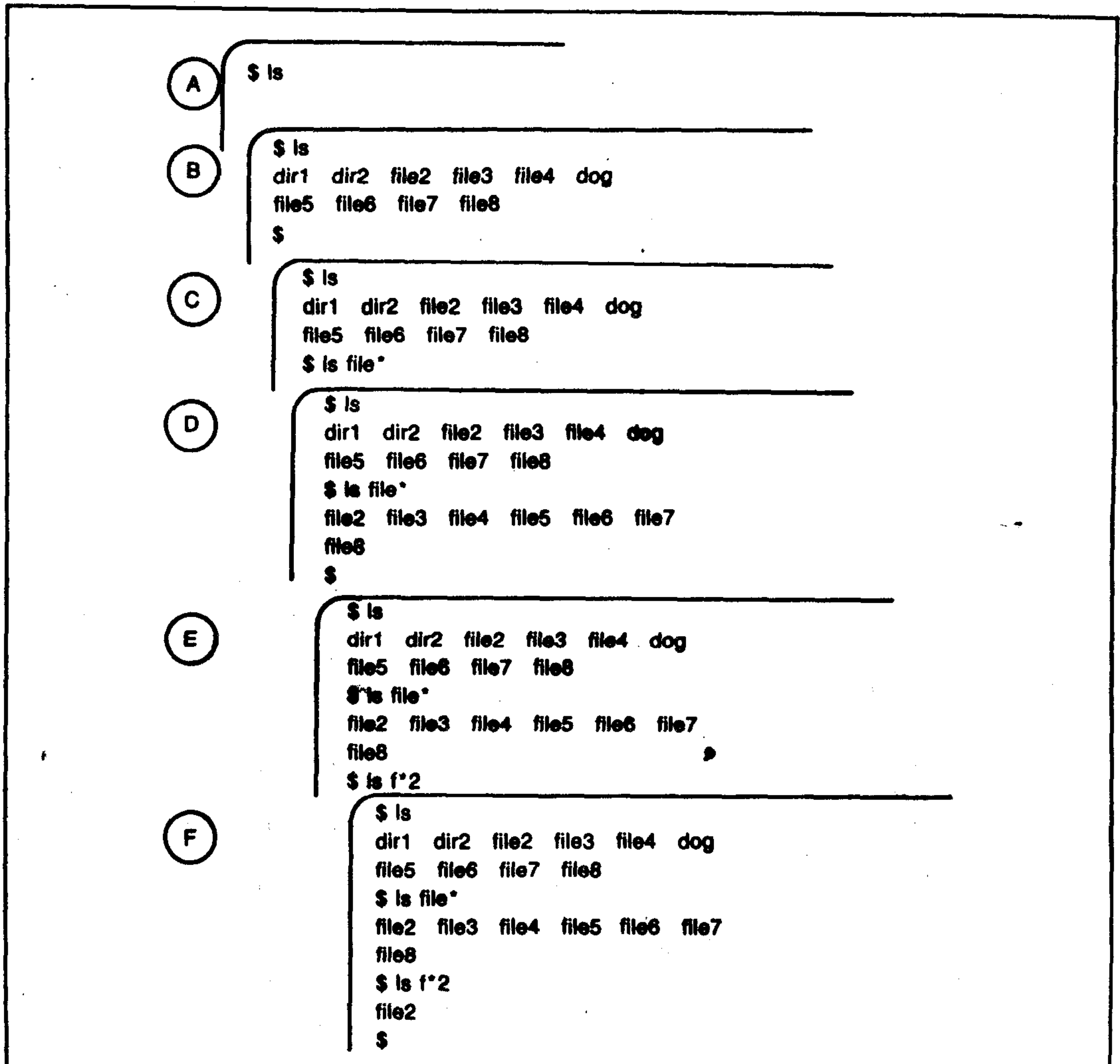


*Fig. 16-2. Searching the file structure with the * wild card.*

You probably expected the shell to display the dog—possibly the dir1 and dir2—but what is the rest of this? When you use the asterisk in this fashion, it will give you the second level of files as well. If you enter the command ls, you get a listing of the files in the directory in which you are working. If you enter an ls *, you will get a display of the second-level files as well.

- Screen D shows the command ls d? entered.
- Screen E shows the error message not found. The reason for this message is that there were no files that began with a d followed by a single character. This constraint did not match dir1, dog, or dir2.

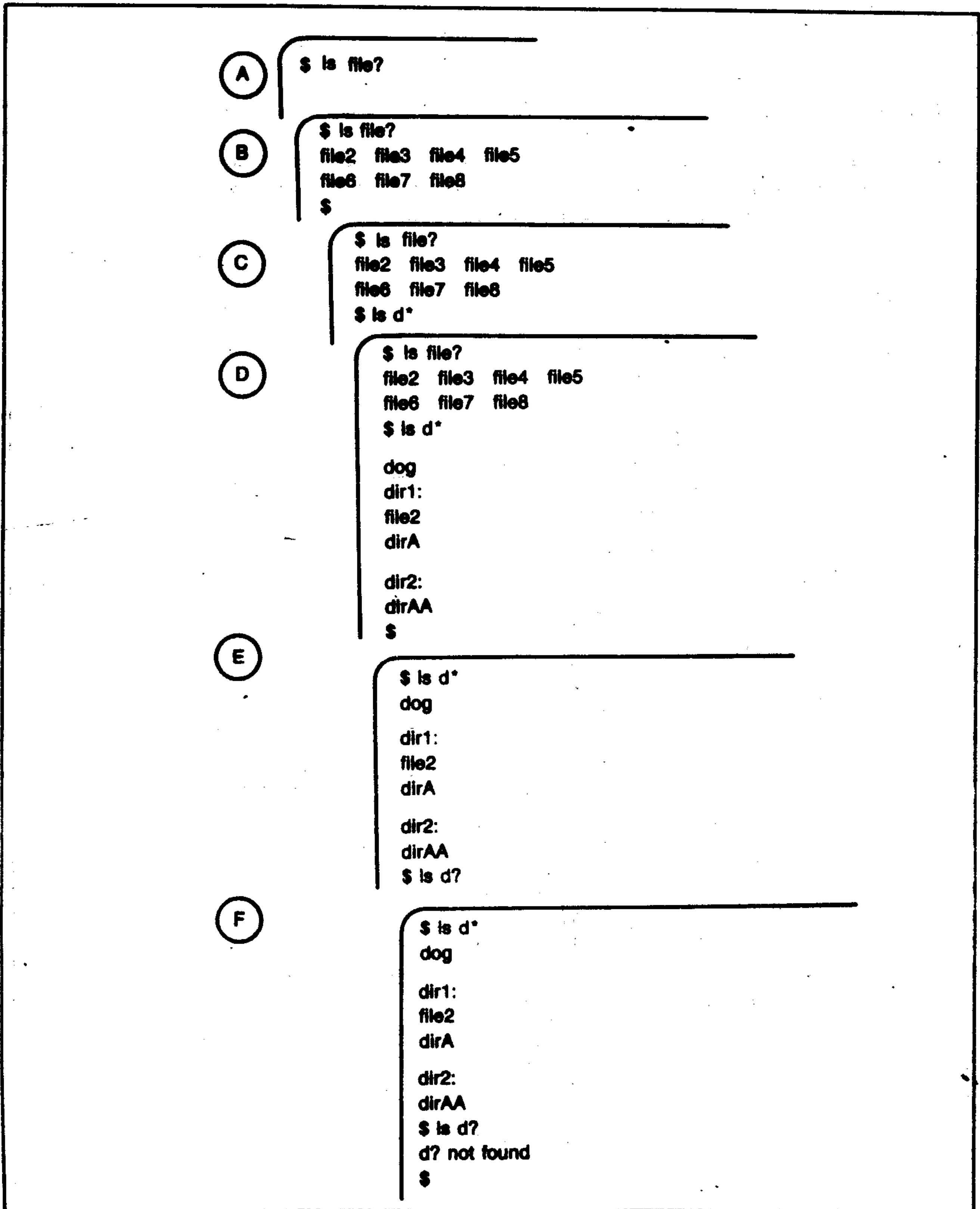


Fig. 16-3. Using the ? wild card, which can stand for only a single character.

Figure 16-4 illustrates another tack that can be taken.

- Screen A shows the command ls *[1]* entered.
- Screen B shows the directory dir1; as expected, the shell also displayed the second level files, the files in the directory dir1.
- Screen C shows the command ls file[1,2] entered. We can use the square brackets to match specified characters or ranges of characters. In this example, the only acceptable matches would be file1 or file2.

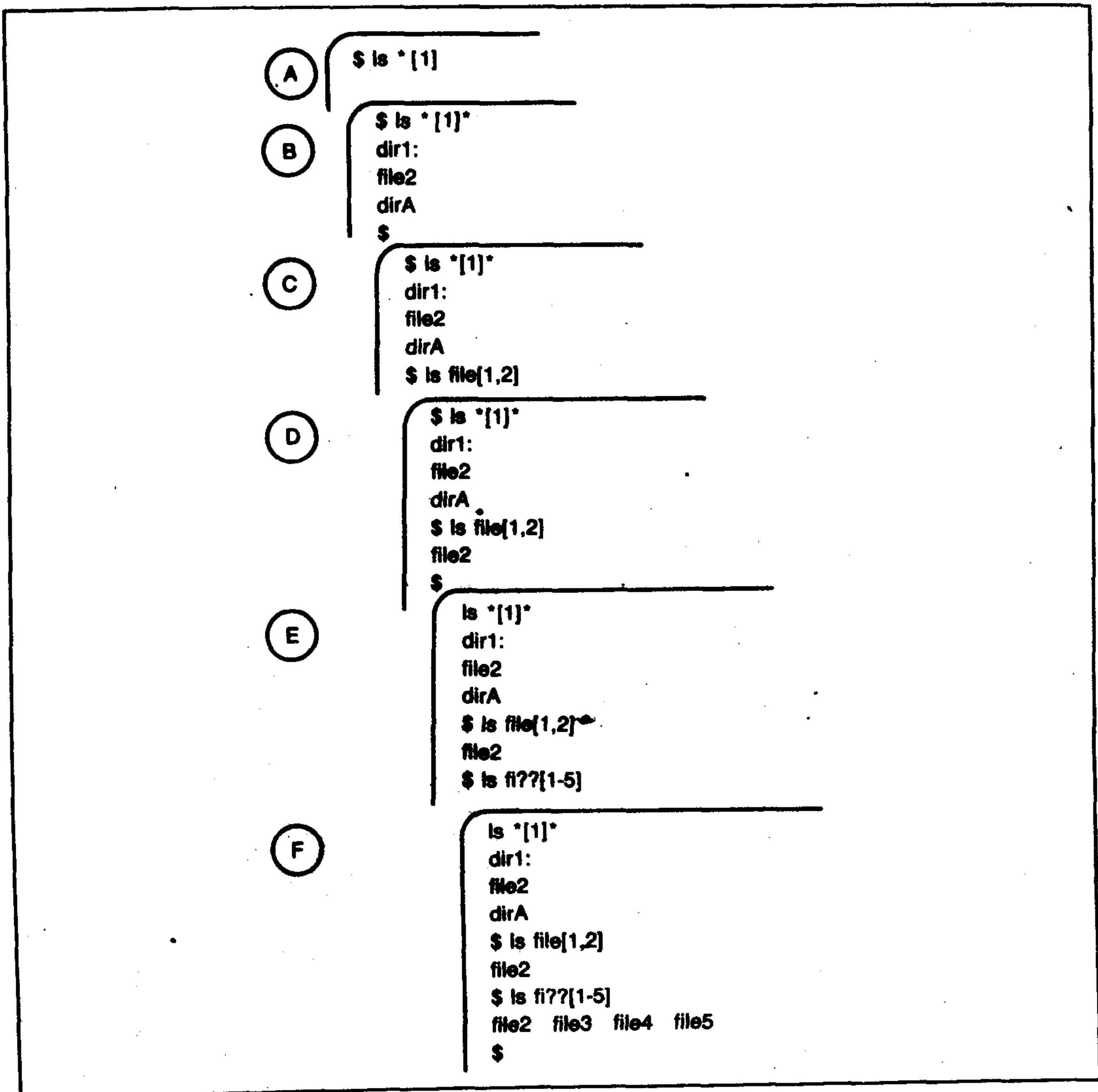


Fig. 16-4. Another file search strategy.

```
A
$ rm dir1
rm: dir1 directory
$

B
$ rm dir1
rm: dir1 directory
$ rmdir dir1 dir2
rmdir: dir1 not empty
rmdir: dir2 not empty
$

C
$ rm dir1
rm: dir1 directory
$ rmdir dir1 dir2
rmdir: dir1 not empty
rmdir: dir2 not empty
$ ls -l dir1

D
$ rm dir1
rm: dir1 directory
$ rmdir dir1 dir2
rmdir: dir1 not empty
rmdir: dir2 not empty
$ ls -l dir1
drwxrwxrwx 2 user1    18 Oct 12 18:53    dirA
-rw-r- -r- - 1    user1    12 Oct 14  2:56    file2

E
$ rm dir1
rm: dir1 directory
$ rmdir dir1 dir2
rmdir: dir1 not empty
rmdir: dir2 not empty
$ ls -l dir1
drwxrwxrwx 2 user1    18 Oct 12 18:53    dirA
-rw-r- -r- - 1    user1    12 Oct 14  2:56    file2
$ rm dir1/file2 dir1/dirA/file1
$
```

Fig. 16-5. A directory must be empty before it can be removed.

- Screen D displays the file **file2**.
- Screen E shows the command **ls fi??[1-5]** entered. This command requests the shell to find all file names beginning with the characters **fi**, followed by any two characters, and ending with any number 1 through 5.
- Screen F shows the matches for this command are files **file2**, **file3**, **file4**, and **file5**.

Now it's clean-up time! Start by examining Fig. 16-5.

- Screen A shows the command **rmdir dir1** entered to remove (delete) this directory from our file structure.
- Screen B displays an error message, **not empty**. This is an indication that there are some files in the directory. A safety feature of Unix will not allow you to remove a directory that still contains files.
- Screen C shows the **ls -l dir1** command entered, so that we can look at the names of the files in this directory.
- Screen D displays the list of files and directories remaining in directory **dir1**.
- Screen E shows the command **rm dir1/file2** entered to remove **file2**. We have to use the relative pathname, since we are operating from the working directory **user1**.

A
```
$ rmdir dirA
$
```

B
```
$ rmdir dirA
$ ls *
file2  file3  file4  dog  file5
file6  file7  file8
$
```

C
```
$ rmdir dirA
$ ls*
file2  file3  file4  dog  file5
file6  file7  file8
$ rm *
$
```

D
```
$ rmdir dirA
$ ls *
file2  file3  file4  dog  file5
file6  file7  file8
$ rm *
$ rmdir dir1 dir2/dirAA dir2
$
```

E
```
$ rmdir dirA
$ ls *
file2  file3  file4  dog  file5
file6  file7  file8
$ rm *
$ rmdir dir1 dir2/dirAA dir2
$ ls
$
```

Fig. 16-6. Closing out the lab exercise session. Your directory is now empty.

Now refer to Fig. 16-6.

- Screen A shows the command rmdir dirA entered. Now that we have removed all subordinate files and directories from dirA, the shell can remove the directory.
- Screen B shows the ls * command entered to look at the status of our directory user1 now. There are still more files and directories to be removed.
- Screen C shows the command rm* entered. This will remove all remaining files in user1.
- Screen D shows the commands rmdir dir2/dirAA dir2 entered. The rmdir command can be used to remove more than one directory at a time. However, if you are removing subordinate directories, as we are here, you must remove the directories in ascending order, i.e., begin with the lowest directory in the structure.
- Screen E shows the command ls command entered and the redisplay of the shell prompt.

There are no more files or directories in your directory. You are finished!

Appendix A

Lab Exercise Command Summary

Appendix A contains detailed information about the commands used in *Unix and Xenix Demystified.* You will note that there are only 20 commands in this appendix. That means that if you learn these 20, plus the few built-in shell commands and metacharacters that we discussed in the tutorial and lab sessions, you can be well on your way to learning to use Unix very effectively.

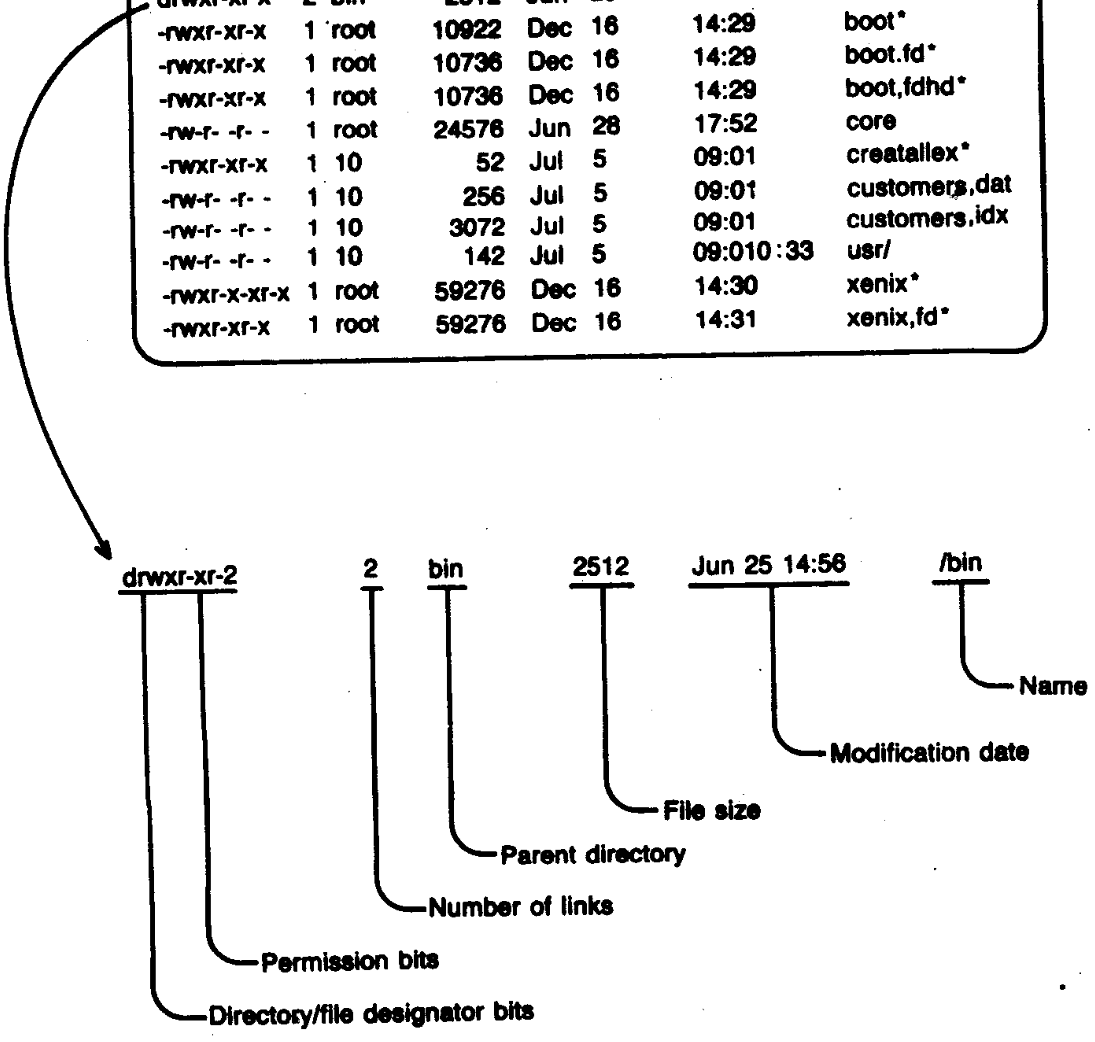
COMMAND: ls

The *list contents of a directory* command is used to list the contents (the names of the files and subdirectories) of a directory. Here is an example of the display for ls command:

```
add.hd     etc       load.hd      usr
bin        fd        lost+found   xenix
boot       fptutor   priboot      xenix.fd
boot.fd    install   pribootfd
dev        lib       tmp
```

Format: ls (options) (file name)

Options: -l — The -l (long) option provides additional administrative information about the files, such as the owner, file size, date last revised, etc. Figure A-1 provides an explanation



```
Directory: /

total 452
-rw-r- -r- -    1 root      36  Dec  1    00:00       .profile
-rw-r- -r- -    1 10       594  Jul  5    09:01       Makefile
-rw-r- -r- -    1 10       594  Jul  5    09:01       aceex
-rw-r- -r- -    1 10       422  Jul  5    09:01       aceex,arc
-rw-r- -r- -    1 10       438  Jul  5    09:01       agents,dat
-rw-r- -r- -    1 10      2560  Jul  a5   09:01       agents,{dx
-rw-r- -r- -    1 10       322  Jul  5    09:01       agentschem
-rw-r- -r- -    1 10      3067  Jul  5    09:02       allexample.c
drwxr-xr-x      2 bin     2512  Jun  25   14:56       bin/
-rwxr-xr-x      1 root   10922  Dec  16   14:29       boot*
-rwxr-xr-x      1 root   10736  Dec  16   14:29       boot.fd*
-rwxr-xr-x      1 root   10736  Dec  16   14:29       boot,fdhd*
-rw-r- -r- -    1 root   24576  Jun  28   17:52       core
-rwxr-xr-x      1 10        52  Jul  5    09:01       creatallex*
-rw-r- -r- -    1 10       256  Jul  5    09:01       customers,dat
-rw-r- -r- -    1 10      3072  Jul  5    09:01       customers,idx
-rw-r- -r- -    1 10       142  Jul  5    09:010:33   usr/
-rwxr-x-xr-x    1 root   59276  Dec  16   14:30       xenix*
-rwxr-xr-x      1 root   59276  Dec  16   14:31       xenix,fd*
```


Fig. A-1. Composition of a long (ls -l) directory listing line.

of the data contained in this printout. An example of display for ls -l command is:

```
total 363
-rwxrwxr-x 1 root        2505 Apr  6 16:42 add.hd
drwxr-xr-x  2 bin         2544 Oct 12 18:44 bin
.
.
.
-rwxrwxr-x 1 root       11040 Apr 17 15:23 boot
```

The first datum is the file designator:
d file is a directory
b file is a device file
c file is a character special file
p file is a FIFO (first-in first out) special file

-a Lists the contents of a directory, including system files which begin with a period. An example of a display for the ls -a command is:

```
.           boot.fd     lib           usr
..          dev         load.hd       xenix
.profile    etc         lost+found    xenix.fd
add.hd      fd          priboot
bin         fptutor     pribootfd
boot        install     tmp
```

-t Same as -l, except that it lists the files according to the last time that the files were opened. An example of the display for the ls -t command is:

```
tmp
bin
etc
.
.
.
xenix
boot.fd
```

-i Lists the contents of a directory, beginning with the i-number. A sample display for the ls -l command is:

```
  8 add.hd
 95 bin
102 boot
.
.
.
 91 usr
100 xenix
```

others	Other **ls** command options include:

1	b	i	q
A	c	m	r
C	d	n	s
F	f	o	u
R	g	p	x

Arguments:	The name (**file name**) of the directory whose contents you wish to list. If no file name is listed, the contents of the current (working) directory will be listed.

COMMAND: who

The **who** command identifies the names of the users currently logged on the system. An example display for the **who** command is:

```
lee     console Oct 14 04:41
joe     tty1    Oct 14 04:50
bill    tty2    Oct 14 04:42
```

Format:		**who (options)**
Options:	**am i**	Identifies your login name. An example of the display for the **who am i** command is: **lee console Oct 14 04:41**
	-u	Provides system administration information about the users currently logged on the system.
	-b	Provides the time and date of the last time the system was booted.
	-t	Provides the time and date of the last time the system clock was changed.
	others	Other **who** command options include **T**, **l**, **d**, **r**, **a**, and **s**
Arguments:		None

COMMAND: pwd

The *print the working directory* command displays the name of the directory in which you are currently working. It also is referred to as the *current directory, current working directory,* or *working directory.* A sample display for pwd is:

/usr/lee/demist

Format:	**pwd**
Options:	None
Arguments:	None

COMMAND: cd

The *change working directory* command is used to change the name of the working directory, i.e., change (move) to another directory.

Format:	**cd (file name)**
Options:	None

Arguments: The name (file name) of the new directory to which you wish to move. If no argument is stated, the shell will automatically move you to your home directory.

COMMAND: echo

The echo command is used in our tutorial to create the contents of files. It is more appropriately used inside shell programs to write diagnostic and other messages on the terminal screen.

Format: echo (information to be displayed)
Options: None
Arguments: The argument is the expression that is to be written on the terminal screen. The expression can be entered with or without single or double quotes.

COMMAND: cat

The cat command is most commonly used to print the contents of a file on the terminal screen. However, its name (*concatenate*) indicates that its true purpose is to combine the contents of several files into one.

Format: cat (options) (file name) (file name) (file name) . . .
Options: -u -s Advanced special-purpose uses.
Arguments: The argument is the name of the file or files that are to be read and displayed on the terminal screen. When combined with the redirect command, >, the cat command can be used to put the contents of the listed files into another named file.

COMMAND: date

The date command is used to display the time and date or for setting the system clock.

Format: date
Options: None
Arguments: If a date and time are entered, the shell will set the system clock to this time and date. The command allows for setting up several different display formats.

COMMAND: mv

The *move* command is used to rename a file. It is so called because, in changing the pathname of a file, you are changing its respective location in the hierarchical filing structure.

Format: mv (current file name) (new file name)
Options: None
Arguments: The mv command requires two arguments, the name of the file to be renamed and/or moved and the new name for

the file or the new location for the file. Full or relative pathnames may be used as required. If the move command is used to "pull" a file into another user's directory, the command will change the name of the owner during the transaction. If the file is sent to another user's directory, the registered owner of the file will not be changed to that of the owner of the directory where the file is being sent.

COMMAND: cp

The *copy* (cp) command is used to make a copy of an existing file.

Format: cp (file name of file to be copied) (copy file name)
Options: None
Arguments: The copy command requires two arguments, the name of an existing file to be copied and the name of the file in which the copy is to be stored. Full or partial pathnames may be used as required. If a copy of a file is being made by a user other than the registered owner of the file, the appropriate permission mode bits being set, the file into which the copy is being stored will be registered to this other user. If the current owner of the file is making a copy of one of his files and sending it to another user, the ownership of the file containing the copy will be registered as the owner of the original file.

COMMAND: mkdir

The *make a directory* command is used to create a new directory file.

Format: mkdir (new file name)
Options: None
Arguments: The argument is the name of the directory you wish to create. Directory names can be created with a full or relative pathname.

COMMAND: ln

The *link* command is used to link one or more file names to a file, i.e., create alias names for a file. The linked file names can be within the registered owner's directory system or within another user's file structure.

Format: ln (file name) (alias file name)
Options: None
Arguments: The link command requires two arguments, the name of the file and the new name that is to be created and linked to it. The argument may be a full or relative pathname.

COMMAND: chmod

The *change mode* command is used to change the permission or mode bits which control access security to a file (or directory).

Format:	chmod (new permission code) (file name)
Options:	None
Arguments:	Two arguments are required for the change mode command. The first argument is the permissions setting to be effected, and the second is the name of the file or directory for which this setting is ordered. There are several methods that can be used to change a file's mode bytes; for simplicity, however, the lab exercises introduced the simplest.

COMMAND: chown

The *change the name of the owner of a file* command is used to change the registered owner of a file.

Format:	chown (new owner login name) (file name)
Options:	None
Arguments:	The change owner command requires two arguments; the first argument is new file owner's login name, the second is the name of the file on which ownership is being changed.

COMMAND: more

The more command is used to display the contents of a file in paginated form on your terminal screen. When you use the concatenate (cat) command to display the contents of a file, if the contents take more than the number of lines that can be fit on the terminal screen, it is difficult to read the information. The more command will fill the screen with data and then await your command to display another line or another screen of data.

Format:		more (options) (file name)
Options:	+line number	A + followed by a line number will cause the more command to begin displaying the contents of the file at that line number.
	+pattern	A + followed by a string of characters will cause the more command to look for a matching string of characters in the contents of the file, and will begin displaying the contents of the file beginning two lines before the line containing the matching string.
		The options -n, d, f, and l are available. However, they provide no significant additional services that warrant discussion.

Arguments:	There are two types of arguments that can be used with the **more** command. The main or more common argument is the name of the file. As noted in the options, however, you can precede this argument with an argument that will start the display of the contents of the file at a specific line number, or two lines prior to a specified pattern.
Comments:	There are subcommands that can be initiated while using the **more** command to display the contents of a file; they are entered when **more** pauses. These commands can be used to skip a series of lines or screens, skip to a specified line, go back to a previous specified line, etc.

COMMAND: grep

The **grep** command is used to find specified words, phrases, or other character strings in a file. The **grep** command will find the specified information in a file and then display the full line in which it occurs. If it occurs on more than one line, **grep** will display all lines containing the information. Several file names may be entered at a time.

Format:		grep (options) (target string) (file name) A file name is not required if the **grep** command is used in conjunction with a pipe, as demonstrated in Chapter 16.
Options:	-n	In addition to displaying the line on which the target information occurs in the file, the -n option will also display the line number.
	-c	This option will suppress the display of the actual line Instead **grep** will only display the number of times that the information occurred in the file.
	-l	The -l option will also not display the line on which the information occurs. It will only indicate the names of the files in which it occurred.
	-v	This option causes the grep command to display all of the lines in a file which do not contain the specified word, etc.
	others	There are two other options, **b** and **s**.
Arguments:		File names are the only arguments.
Comments:		The **egrep** and **fgrep** are similar to the **grep** command.

COMMAND: sort

This command is used to sort the contents of a file, usually alphanumerically on one of the words or columns in a line. Unless otherwise indicated with options and position indicators, **sort** will begin sorting on the first column of each line in a file and end at the first blank. Several files already sorted also may be merged with the **sort** command.

The **sort** command is a very simple command, as we have seen in the lab session examples. When you begin to examine all of the "if, ands, and buts" of the usage of **sort**, however, it could take several pages—and none of the other texts provide very much information. The best way to approach **sort** is to learn to use it in its basic format and then experiment with some of its additional features.

Format: **sort (options) (file name)**

Options:

- **-b** This option makes the **sort** command ignore any blanks that precede the first word on the line. This option is required should you have a "ragged" left margin.
- **-d** This option will ignore all non-alphanumeric characters. Blanks are still significant with this option.
- **-f** This option treats uppercase letters as if they were lowercase. Without the **-f** option, uppercase words would be sorted separately from lowercase.
- **-n** This option sorts numeric character strings. The **-n** option will sort by arithmetic value, take into consideration positive and negative values, and recognize decimal values.
- **-r** This option will sort the contents of a file in reverse alphanumeric order.

Arguments: The basic argument is the name of file whose contents are to be sorted. The **sort** command can be used in conjunction with a pipe, in which case a file name is not required.

Other arguments exist, as follows:

sort (options) (begin/end positions) (file name)

indicates beginning and ending positions of a sort. The beginning and ending positions argument enables you to instruct the **sort** command to begin the sort on a specific field (a character string followed by a blank, such as a word) and character, in a field. For example, a +2 following the command (or option if used) will direct the sort to skip the first and second fields, and begin with the third field on each line in the file. A +2.2 would indicate that you want to begin sorting on the third character of the third field. A +2 is assumed to be a +2.0. You can also use a number like a +.1 to indicate that you want to skip the first character in the first field and begin the sort with the second character, etc.

The end of the sort range is indicated by a negative number such as −2.4. This number would count two fields and four characters from the right, just as the positive number skipped fields and characters from the left.

Redirect output of the sort to a file is accomplished with the following command line:

sort (options) (-ofile name) (file name)

By preceding a file name with **-o**, the **sort** command will redirect the output of the sort to a specified file. This is similar to the operation of the > (redirect output) shell metacharacter. The redirect output argument can be used in conjunction with options and/or beginning and ending position indicator arguments.

You may merge two files with this command line:

sort (options) (file name 1) (file name 2)

The **sort** command can be used to merge two files that have been previously sorted to the same rules (i.e., options and arguments). This should not be confused with the fact that you can use the **cat** and redirect output commands to first combine two files and then sort the contents of this new file. Several sub-options exist in this mode:

- **-m** This option is used to merge files that are already sorted. The sort order must be the same for each file.
- **-c** This option is used to check that a file is sorted in the desired order. If the files to be merged are not properly sorted, **sort** will not output the data to the new named file.
- **-u** This option will suppress sending duplicate lines to the new named file. However, keep in mind that it will only check for duplication based on the indicated sort range and conditions specified by the other options and arguments.

COMMAND: wc

The *word count* command is used to count the number of lines, words, and/or the number of characters in a file.

Format:	**wc (options) (file names)**
Options:	If no option is specified, the word count command will automatically display the number of lines, words and characters in a file. If you use any one or two options only those items will be displayed.
-l	Displays only the number of lines.
-w	Displays only the number of words.
-c	Displays only the number of characters.
Arguments:	File names are the only arguments.

COMMAND: expr

This command is used to evaluate the specified argument, referred to here as an *expression*, and use the determined "value" in conjunction with the command line or shell program in which it is used. Examples of the use of this command are found in the lab sessions.

Appendix B

Consolidated Directory of Unix Utilities

This appendix is a consolidated listing of the Unix utility files found in Unix Version 7, System III, System V, Berkeley 4.2, PC/IX, Xenix, and UniPlus+. I have developed this appendix so that you can find a brief description of just about any utility that exists. Appendix C provides a table of the systems in which these utilities are available. Keep in mind, however, that Unix systems may be packaged, i.e., sold with more or fewer utilities than indicated as part of a particular system in these tables.

I have selected these systems because of their prominence in the development of Unix, their popularity, and/or because of the amount of general interest that has been generated through industry publications.

The commands presented in this composite listing represent those generally attributed to section 1 and sections 4 through 8 in the Unix *Programmer's Reference Manual* or equivalent sections in the manuals for Unix-based systems; they include file handling, text processing, timesharing, telecommunications, database management, hardware interface, and general-purpose utilities. While a certain number of the most fundamental software tools—such as interpreters, compilers, debuggers, and other utilities for executing executable files—are included also, the utility libraries for program development in the C language generally are not. Users who need this information are referred to the appropriate system manuals for their installation. For quick reference, the Software Development Library for Xenix found in Appendix D is a fair representation of the C utilities available.

UTILITY NAME	UTILITY DESCRIPTION
300	Supports special functions of DASI 300 and 300s terminals.
4014	Special interface for the Tektronix 4014 terminal.
450	Supports special functions of the DASI 450 terminal.
a.out	Assembler and line editor output file.
a.out.pdp	Same as **a.out**, but for Digital Equipment Corporation PDP-series computers.
aardvark	An adventure game.
ac	Login for system accounting.
acc	Special file for ACC LH/DH IMP interface.
acct	Enables or disables process accounting.
acctcms	Command summary from per-process accounting records.
acctcom	Searches and prints process accounting file(s).
acctcon	Connect-time accounting.
acctcon1	Records user login/logout.
acctcon2	User login/logout accounting record.
acctdisk	Maintains accounting record of blocks, used by user.
acctdusg	Accounting for computer disk usage.
acctmerg	Merges, creates, or formats total accounting files.
accton	Turns on system accounting.
acctprc	Process accounting.
acctprc1	Provides system accounting specifics.
acctprc2	Records system accounting data.
acctsh	Shell procedures for accounting.
acctwtmp	Writes a **wtmp** record to standard output.
ad	Special file for data translation analog-to-digital converter.
adb	A C program debugging aid.
addbib	Creates or extends bibliographic database.
adduser	Adds, deletes, and changes user and group information.
admin	Creates and administers SCCS files.
adventure	The game of Adventure.
aliases	Aliases file for **sendmail**.
aliens	Alien invaders attack the earth.
altblk	Information for alternate for bad disk block.
analyz	Post-mortem of system crashes.
apropos	Locates commands by keyword look-up.
ar	Maintains groupings of files in a single archive file.
ar.pdp	Same as **ar**, but for PDP-series computers.
arcv	Reformats archives.
arff	Floppy archiver and copy utility.
arithmetic	Game providing drills in number facts.
arp	Special file for address resolution protocol.
ar.pdp	Maintains PDP-11 files in a single archive file.
as	Assembler programming language.
as.pdp	Assembler for the PDP-11.
asa	Interprets ASA carriage control characters.
ascii	Listing of ASCII character set in octal and hexadecimal.
asktime	Sets system date and time.
assign	Assigns a device to a user.
asy	An asynchronous adapter.
at	Executes commands at a later time.

atq	Examines the **at** job queue.
atrm	Removes an **at** job from the queue.
autoconf	Special file for diagnostics from autoconfiguration code.
awk	Scans a file for a specified set of patterns and modifies the data as programmed.
back	The game of backgammon.
backgammon	Another backgammon game.
badsect	Flags bad sectors.
banner	Prints large letters to make posters.
bas	Interactive interpreter for unnumbered statements.
basename	Strips file name prefix endings and deliver portions of pathnames.
bc	An arithmetic programming language.
bcd	A game for converting to antique media.
bdiff	Compares differences between contents of very large files.
bfs	Reads contents of very large files.
biff	Notifies you that you have mail and who it is from.
binmail	Sends or receives mail between users.
bj	The game of blackjack.
bk	Special file for machine-machine communications.
boggle	The game of boggle.
boot	Starts up operating system.
bugfiler	Files bug reports.
bs	A compiler/interpreter for the **bs** programming language.
cal	Prints a calendar for any specified year.
calendar	Prints messages you put into the reminder service.
canfield	A solitaire card game.
cat	Combines and/or displays contents of files.
catman	Creates text on **cat** for Unix manual.
cb	C program beautifier that displays program structure.
cc	C programming language compiler.
cd	Changes the current or working directory.
cdc	Changes delta commentary of an SCCS delta.
cflow	Builds a graph of the external references for C, **yacc**, **lex**, and object files.
chargefee	Records system usage fee.
chase	A game in which you must escape the killer robots.
checkcw	Checks that left and right margins are balanced.
checkeq	Mathematical expressions text formatter for **troff**.
checkers	The game of checkers.
checklist	List of file systems that are processed by the **fsck** utility.
checkmm	Checks contents of named files for errors in use of **mm**.
checknr	Checks **nroff/troff** files.
chess	The game of chess.
chfn	Changes finger entry.
chgrp	Changes the group in which a user is included.
ching	A game featuring the Chinese book of changes and other goodies.
chmem	Changes the amount of dynamic memory allocated to a specified file.
chmod	Sets file and directory user access permissions.
chown	Changes the name of the owner of a file or directory.

chparm	Changes or examines system parameters.
chroot	Changes root directory for a command.
chsh	Changes default login shell.
ckpacct	Periodically checks file size of /usr/adm/pacct.
clear	Clears terminal screen.
clri	Clears i-node.
cmp	Compares two files to determine where differences occur.
col	Filters reverse line feeds.
colcrt	Formats nroff for review on terminal.
comb	Combines SCCS deltas by restructuring specified files.
comm	Selects or rejects lines common to two sorted files.
compact	Compresses and uncompresses files and then prints contents.
comsat	biff interface.
config	Configures an operating system.
connect	Permits establishing a connection to a remote system.
cons	Special file for DEC VAX-11 console interface.
convert	Updates older object and archive files to System V.
copy	Copies contents of groups of files.
core	Writes out a core image of a terminated process.
cp	Copies contents of a file into another file.
cpio	Copies file archives in and out.
cpp	C language preprocessor.
cprs	Compresses the size of an IS25 object file.
craps	The game of craps.
crash	Examines system images.
cref	Makes cross-reference listing of assembler or C programs.
cribbage	The card game of cribbage.
cron	Background clock daemon.
crypt	Encodes or decodes contents of a file.
csh	The Berkeley shell with C-like syntax.
csplit	Splits content of file as specified.
css	Special file for DEC IMP-11A LH/DH IMP interface.
ct	Special file for phototypesetter interface.
ctags	Creates a tags file.
cu	Calls the Xenix system.
cut	Cuts out selected fields from each line of a file.
cw	Prepares constant-width font for troff text.
cwcheck	Checks CW macro text.
cxref	Analyzes C program files and builds cross-reference symbol table.
dab144	Bad sector information.
date	Prints and/or sets the date.
dbx	Debugger.
dc	A desk calculator for precision arithmetic.
dcheck	File system directory consistency check.
dd	Converts contents of EBCDIC file to ASCII and vice-versa.
deassign	Deassign a device.
del	Deletes files and confirms the deletion by printing the names of the deleted files.
delta	Makes a delta (change) to an SCCS file.
deroff	Removes nroff, troff, tbl, and eqn embedded commands.
devinfo	Device characteristics.

devnm	Device name.
df	Prints amount of free disk file space.
dh	Special file for DH-11/DM-11 communications multiplexer.
diction	Reads and comments on writing style.
diff	Prints location of lines that differ in two files.
diff3	Three-way differential file comparison.
diffmk	Marks the differences between two files.
dir	Format of directories.
dircmp	Lists differences between two directories.
dirname	Delivers a portion of a pathname.
dis	A 3B20S disassembler.
disable	Turns terminal usage off.
disk	Partitions hard disk into smaller sectors.
diskpart	Calculates default disk partition sizes.
disktab	Disk description file.
display	System console display (color graphics or monochrome adapter utility.
dmc	Special file—point to point communications device.
dmesg	Collect system diagnostic messages in error log file.
dmf	Special file—DMF-32 terminal multiplexer.
dn	Special file—autocall interface.
doctor	Game: interaction with a computer psychoanalyst.
dodisk	Disk accounting function.
doscat	Executes **cat** on an MS-DOS floppy disk file.
doscp	Copies files to/from an MS-DOS floppy disk.
dosdel	Deletes specified DOS files.
dosdir	Lists contents of a PC-DOS disk.
dosls	Lists directory for files on an MS-DOS disk.
dosmkdir	Creates a directory on an MS-DOS disk.
dosread	A DOS file read utility.
dosrm	Deletes a file on an MS-DOS disk.
dosrmdir	Deletes a directory on an MS-DOS disk.
doswrite	A DOS file write utility.
dpd	A sending daemon for the Honeywell line printer.
dpr	Off-line print utility.
drtest	Disk testing program.
drum	Special file for a paging device.
dtype	Prints disk type (Xenix, MS-DOS, **tar**, etc.).
du	Prints out a summary of current disk usage.
dump	Incremental file system dump and file system backup utility.
dumpdir	Prints the names of files on a dump tape.
dumpfs	Dumps file system information.
dz	Special file for communications multiplexer.
e	The **INed** screen editor.
ebcdic	A hexadecimal listing of the EBCDIC character set.
ec	Special file for Ethernet interface.
echo	Prints the following (argument) message.
ed	The **ed** editor, a line-oriented text editor.
edquota	Edits user quotas.
efl	Extended FORTRAN programming language.
egrep	Searches a file for a specified pattern.

en Special file for Ethernet interface.

enable Controls or reports the availability of port for logging in.

env Sets environment for command execution.

environ User environment utility.

eqn Formats mathematical text for **nroff** or **troff**.

eqnchar Special character definitions for **eqn** and **neqn**.

eqncheck Typesets mathematics.

errfile A log of hardware errors.

error Analyzes and disperses compiler error messages.

ex A line editor.

expand Expands tabs to spaces and vice versa.

explain Explains diction and prints wordy sentences.

expr Evaluates arguments as an expression.

eyacc Modifies **yacc** to provide improved error recovery.

f77 A FORTRAN compiler.

factor Factors a number.

false Provides truth table values.

fastboot Reboots or halts system without file checking.

fcntl Opens file control.

fd A device file for floppy disk devices.

fed A font editor.

ffill A fast version of **fill** which does not invoke **nroff**.

fget Retrieves files from the Honeywell 6000.

fgrep Searches a file for a specified pattern.

file Determines which programming language is used in a file.

filehdr A file header for common object files.

filesystems A file system description file.

fill Arbitrarily fills broken lines or text using the **INed** editor.

find Finds the full pathname of a file in the hierarchical structure.

finger User information lookup program.

fish The game of fish.

fixascii Fixes (with **INed**) ASCII files broken due to system crash.

fl Special file for console floppy interface.

fmt A simple text formatter.

fold Folds long lines for a device with limited display.

format pcix - INed paragraph formatter.

format Formats Xenix diskettes.

fortran A FORTRAN compiler.

fortune A game that prints fortunes and aphorisms.

fp Device file for floating-point chip.

fpr Prints FORTRAN program lines.

freq Reports the frequency of characters used in a file.

fs Contains format of file system storage volume.

fsck File system consistency check and interactive repair utility.

fsdb A file system debugger.

fsend Sends files to the Honeywell 6000.

fspec Format specification in text files.

fsplit Splits **f77**, **ratfor**, or **efl** FORTRAN files.

fstab Contains information about the file system.

ftpd Protocol interface.

fwtmp Manipulates **wtmp** records.

gcat	Sends phototypesetter output to the Honeywell 6000.
gcore	Prints core image of a running process.
gcosmail	Sends mail to HIS user.
gdev	Graphic device routines and filters.
ged	A graphic editor.
get	Generates an ASCII text file from each named SCCS file.
getopt	Separates command options for easy parsing.
gets	Gets a string from standard input.
gettable	Gets NIC format host tables from host.
getty	Sets login port characteristics.
gettydefs	A reference file for speed and terminal settings (used by **getty**).
gettytab	Terminal configuration database.
ghost	Reconstructs previous versions of an **INed** structured file.
gps	Formats graphic files.
graph	Draws a graph.
graphics	Accesses graphics and numerical commands.
greek	Character interface to specified terminals.
grep	Searches a file for a specified pattern.
group	Contains the information about group file.
groups	Prints group membership.
grpcheck	A group file checker.
gutil	Graphic utilities file.
halt	Stops the processor.
haltsys	Shuts down the operating system.
hangman	Plays the familiar guess-the-word game.
hd	Device file for the PC/IX hard disk.
hd	Xenix utility that gives a hex dump of a file.
hdr	Prints binary file header information.
head	Gives the first few lines of a data stream.
help	Explains message from a command or the use of a command.
hex	Translates object files.
hier	File system hierarchy.
history	Prints history of changes made to **INed** structured files.
hk	Special file for moving head disk.
hold	Allows logged-in users to continue, but disallows new user log on.
hostid	Sets or prints identifier of current host system.
hostname	Sets or prints name of current host system.
hosts	Host name database.
hp	Handles special functions of HP 2640- and 2621-series terminals.
hpio	Terminal tape file archiver for HP 2645A.
ht	Special file for magtape interface.
htable	Converts NIC host tables.
hy	Special file for Network Systems Hyperchannel interface.
hyphen	Finds and prints all hyphenated words in a file.
icheck	Performs file system storage consistency check.
id	Prints user and group IDs and names.
ifconfig	Configures network interface parameters.
ik	Special file for Ikonas graphics device interface.
il	Special file for Interlan Ethernet interface.
imp	Special file for 1822 network interface.
implog	IMP log interpreter.

implogd	IMP logger process.
indent	Indents and formats C program.
ined	The **INed** editor and files used by the **INed** system.
inet	Special file for Internet protocol.
init	System initialization process.
inittab	Controls information for **init**.
inode	Contains format of an i-node.
install	Installs commands (utility files).
intro	Gives an introduction to system maintenance procedures.
iostat	Reports input/output statistics.
ip	Special file for Internet protocol.
ipcrm	Removes a message queue.
ipcs	Interprocess communications facilities status report.
issue	Prints project I.D. as a login prompt.
istat	Prints the contents of the i-node for a file.
join	Relational database operator.
jotto	A secret word game.
just	Fills and justifies (using **INed**) arbitrarily indented paragraphs of text.
kasb	An assembler/disassembler for the KMC11B microprocessor.
keyboard	Device file for the system console keyboard.
kg	Special file for the line clock.
kgmon	Dumps operating system's profile buffer.
kill	Terminates a specified process.
killall	Terminates all user-controlled processes.
kmem	Device file for kernel memory image.
l	Lists with pagination.
last	Prints the last date a user logged on.
lastcomm	Prints a reverse-order list of commands processed.
lastlogin	Prints the last date a user logged on.
lc	Formatted file list.
ld	Combines (links) several object programs into one.
ld.pdp	A link editor for DEC PDP-series computers.
ldfcn	Common object file access routines.
learn	Runs a tutorial.
leave	A message reminder service.
lex	Generates programs for simple lexical analysis text.
li	Lists contents of directory.
life	The game of life.
line	Copies one line from standard input, writes to standard output.
linenum	Generates line numbers for C source lines in common object files.
link	Exercises **link** and **unlink** system calls.
lint	A C program checker for potential bugs.
lisp	A Lisp interpreter.
liszt	A program interpreter for the Franz Lisp dialect.
ln	Makes a link.
lo	Special file for software loopback network interface.
lock	Reserves a terminal.
login	The system sign-on utility.
logname	Gets a login name.
look	Finds lines in a sorted list.

lookbib	Builds inverted index or finds references in a bibliography.
lorder	Finds ordering relation for an object library.
lp	Device file for parallel printer adapter.
lpc	A line printer control program.
lpd	Line printer daemon.
lpq	Prints the spool queue.
lpr	A line printer spooler.
lprm	Cancels a job in the spooler queue.
lpstat	Prints line printer status information.
ls	Lists a file contained in a directory.
ls7	Berkeley version of ls in UniPlus+.
lxref	A Lisp cross-reference program.
m4	Processor for programming language macros.
machid	Provides truth value about your processor type.
mail	Sends or receives mail among system users.
mailaddr	Mail addressing utility.
make	Maintains, updates, and regenerates groups of programs.
makedev	Makes a special device file.
makekey	Generates an encryption key.
man	Finds or makes entries in the Unix manual.
manroff	Formats and prints entries in the Unix manual.
mant	Typesets documents and manual pages.
master	Master device information table.
master.dec	Master device table for DEC computers.
master.u3b	Master device table.
maze	A game of solving random mazes.
me	Macro formatter utility.
mem	Device file for memory image and kernel memory image.
memuse	Reports the amount of data space not used during program execution.
mesg	Permits or denies messages to be sent to your terminal.
mille	The game of Mille Bournes.
mkdir	Creates a directory.
mkfs	Constructs a file system.
mklost+found	Makes a lost and found directory and collects lost files.
mknod	Builds a special device file.
mkproto	Makes a prototype file system.
mkstr	Creates an error message file.
mkuser	Adds a new user account.
mm	Prints documents formatted with the MM macros.
mmcheck	Checks mm source.
mmt	Typesets documents and manual pages for troff.
mnttab	Contains a table of devices mounted by the mount command.
monacct	Indicates the system accounting period.
monop	The game of Monopoly.
moo	A numbers guessing game.
more	Prints contents of a file, a screen and/or a line at a time.
mosd	Macros for formatting documents.
mount	Mounts a file system.
mptx	Macros for formatting a permuted index.
ms	Macro formatters.

msgs	Program for sending system messages.
mt	Special file of typeset documents and manual pages.
mtab	Mounted file system table.
mtio	Special file for magtape interface.
mv	Moves or renames contents of files and directories.
mvdir	Moves or renames directory.
ncheck	Generates names from i-numbers.
neqn	Typesets mathematics.
net	Executes a command on the PCL network.
netstat	Prints network status.
netutil	Administers mail network.
newaliases	Rebuilds the database for the mail aliases file.
newfile	Converts an ASCII text file into an INed structured file.
newform	Changes the format of a text file.
newfs	Constructs a new file system.
newgrp	Logs the user into a new group.
news	Prints news items, usually system administration messages.
nice	Runs a command at low priority.
nl	Numbers lines as specified.
nm	Prints a name list of each object file.
nm.pdp	Prints name list for PDP-series computers.
nohup	Runs background process after user logs off.
nroff	Text formatting and typesetting utility.
nroff7	Text formatting and typesetting utility.
nscstat	Queries the operation status of the NSC network.
nsctorje	Re-routes jobs from NSC network to RJE.
null	The null file.
nulladm	Maintains ownership and permissions on system accounting data files.
number	A game that converts numerical to written form of number.
nusend	Sends files over NSC network to another system.
od	Octal dump utility.
pac	Printer/plotter usage accounting utility.
pack	Compresses or expands the contents of a file.
pagesize	Prints the system page size.
panic	Panic messages file.
param	System parameters file.
passwd	Changes login password.
paste	Merges same lines of several files or subsequent lines of one file.
pc	A Pascal compiler.
pcat	Looks at packed files.
pcl	Special file for network interface.
pdx	A Pascal debugger.
phones	Remote host phone number database.
pi	A Pascal interpreter code debugger.
pix	A graphics interface utility.
plot	Graphics plotting instructions file.
pmerg	A Pascal file merger.
pnch	Creates a file index for card images.
ports	Device file for terminal port description.
portstatus	A schedule of prime time and holidays.

pr	Prints the contents of a file.
prctmp	Prints the session record file.
prdaily	Prints a report of the previous day's accounting.
prep	Prepares text for statistical processing.
primetime	Schedule file for prime time and holidays.
print	Enters files into the printer spooler queue.
printcaps	Printer capabilities database.
printenv	Prints out the environment.
prmail	Prints out mail in the post office.
prof	Displays profile data.
profile	A utility for setting up a user's environment at login time.
proto	Constructs a prototype file for a file system.
protocols	Protocol name database.
prs	Prints an SCCS file.
prtacct	Formats and prints total system accounting file.
ps	Print status of all processes.
ps	Special graphics device file for 4.2bsd.
pstat	Prints system facts.
pti	Phototypesetter interface utility.
ptx	Permuted index utility.
pty	Special file for pseudo-terminal driver.
pup	Special file for Xerox PUP-1 protocol.
put	Utility for transferring files between Unix systems.
put7	Utility for transferring files between Unix systems.
pwadmin	Administers the password file.
pwck	Password file checker.
pwcheck	Password file checker.
pwd	Prints current or working directory name.
px	A Pascal interpreter.
pxp	A Pascal execution profiler.
pxref	A Pascal cross-reference program.
qconfig	Configuration file for the queue system.
qdaemon	Queueing daemon.
quiz	A test-your-knowledge game.
quot	Summarizes file system ownership.
quota	Displays disk usage and limits.
quotacheck	File system quota consistency check utility.
quotaon	Switches file system quotas on or off.
rain	Animated raindrops display game.
random	Random number generator routine.
ranlib	Convert archives to random libraries.
ratfor	Rational FORTRAN dialect.
rc	Normal startup initialization.
rcp	Copies files between machines.
rcvhex	Translates Motorola S-records transferred into a file.
rdump	File system dump utility.
readfile	Prints INed structured file text.
reboot	Restarts the operating system.
refer	Finds and inserts literature references in files.
reform	Reformats text files.
regcmp	Regular expression compile command.
regexp	Regular expression compile and match routines.

reloc	Relocation information for a common object file.
remote	Executes a command on another machine.
remsh	Remote shell file.
renice	Alters priority for running processes.
repquota	Summarizes quotas.
reset	Resets teletype bits.
restor	Incremental file system restore utility.
restore	Incremental file system restore utility.
rev	Reverses lines of a file.
reversi	The game of reversi.
rexecd	Remote execution server utility.
rjestat	RJE status report file.
rlogin	Remote login utility.
relogind	Remote login utility.
rm	Removes specified files.
rmail	Sends mail among users.
rmdel	Removes a delta from an SCCS file.
rmdir	Removes specified directories.
rmhist	Removes history information from **INed** structured files.
rmt	Remote magtape protocol module.
rmuser	Deletes a user account.
robots	Escape from the robots game program.
roff	A typesetting program for terminals.
roffbib	Runs off bibliography of a database.
rogue	A game of dungeons and doom.
route	Allows manual manipulation of the routing table.
routed	Network routing daemon.
rpl	Replaces all occurrences of a string in a file being edited by **INed**.
rrestore	Restores a file system dump across the network.
rsh	A restricted shell used in tightly controlled environments.
rsh	Remote shell for **4.2 bsd**.
rshd	Remote shell interface utility.
rstat	Network statistics program.
runacct	Runs daily system accounting.
ruptime	Prints host status of local devices.
rwho	Prints who is logged on to local terminals.
rwhod	System status interface utility.
rx	Special file for floppy disk interface.
rxformat	Formats floppy disks.
sa	System accounting utility.
sact	Prints current SCCS file editing activity.
sadp	Disk access profiler utility.
sag	System activity graphic utility.
sar	System activity reporter.
sash	Second-level bootstrap program.
savcore	Saves a core dump of the operating system.
scat	Prints current SCCS file editing activity.
scc	A C compiler for stand-alone programs.
sccsdiff	Compares and prints differences between two SCCS files.
sccsfile	Format of SCCS file.
scnhdr	Section header for a common object file.
script	A stream editor.
sdb	Symbolic debugger utility.
sddate	Prints and sets dump dates.
sdiff	Side-by-side difference program.
se	Screen editor for video terminals.
sed	The **sed** stream editor.

see	Displays nonprinting characters contained in a file.
send	Submits RJE jobs.
sendbug	Mails a systembug report to **4bsd-bugs**.
sendmail	Sends mail over the Internet.
services	Service name database.
setmnt	Establishes mount table.
setnode	Sets system node name.
settime	Changes file access and modification dates.
sh	AT&T shell program.
shutacct	Turns process accounting off at system shutdown.
shutdown	Shuts down system.
size	Prints size of an object file in octal and hexadecimal.
size.pdp	Size program for DEC computers.
skulker	Cleans up file systems by removing unwanted files.
sky	Obtain ephemerides, an astronomy game.
sleep	Suspends execution of a command for specified interval.
snake	A chase game.
sno	The SNOBOL programming language compiler and interpreter.
soelim	Expands **nroff** **.so** statements.
sorry	Invalid account response utility.
sort	Sorts contents of file, or merges and sorts two or more files.
sortbib	Sorts bibliographic database.
spell	Finds spelling errors.
spline	Interpolates a smooth curve from points.
split	Splits a file into n-line pieces.
splp	Sets or reports parallel printer drive options.
ssp	Makes output single-spaced.
stab	Symbol table types file.
stackuse	Determine stack requirements for C programs.
stat	Contains data returned by **stat** system call.
sticky	Executable files with persistent text.
stlogin	Sign on to synchronous terminal utility.
strings	Finds the printable strings in binary file.
strip	Removes symbols and relocation bits of assembler or link editor.
strip.pdp	The **strip** utility for PDP computers.
struct	Sets terminal interface.
ststat	Synchronous terminal facilities status report file.
stty	Sets up interface options for a terminal.
style	Comments on writing style.
su	Obtains the privileges of the superuser for another user.
subset	Subsets of PC/IX.
sum	Sums and counts blocks in a file.
sum7	Calculates and prints **checksum** and the number of blocks in a file.
sumdir	Calculates and prints **checksum**, and the number of characters in a file.
swapon	Specifies an additional device for paging and swapping.
symorder	Rearranges name list.
syms	Common object file symbol table format file.
sync	Updates the super block.
sysadmin	Generic interface to backup/restore mechanism.
syslog	Logs system messages.
system	Format of 3B20S system description file.

tab	Changes blanks into tabs.
tabs	Sets tabs on an output device.
tail	Prints the last (as specified) part of a file.
take	Utility for transferring files between Unix systems.
take7	Utility for transferring files between Unix systems.
talk	Sends messages to another user.
tar	Tape archiver utility.
tbl	Formats tables for nroff or troff.
tc	Phototypesetter simulator.
tcp	Special file for Internet Transmission Control protocol.
tee	Channels data between commands.
telnet	TELENET protocol interface.
telnetd	TELENET protocol interface.
term	Conventional names for terminals and other output devices.
termcap	Database of terminal capability for interfacing the computer with the terminal.
test	Condition evaluation command.
tftpd	DARPA protocol interface.
time	Prints elapsed time a command spent in the system.
timex	Prints elapsed time required to execute a command.
tip	Connects to a remote system.
tm	Special file for magtape interface.
toc	Table of graphic routines.
touch	Updates access and modifies times of a file access.
tp	Manipulates tape archive.
tplot	Produces graphing instructions.
tr	Translates characters.
trek	The game of Trekkie.
trman	Translates Version 6 macros to Version 7 macros.
troff	Text formatting and typesetting utility.
troff7	Formats text for printing on a Graphic Systems C/A/T phototypesetter.
trouble	Trouble log file.
trpt	Transliterate protocol trace utility.
true	Provide true values.
ts	Special file for magtape interface.
tset	Sets terminal type.
tsort	Topological sort utility.
ttt	The game of tic-tac-toe.
tty	Device file for terminal.
tty	Prints the pathname of the terminal you are using.
ttys	Terminal initialization data file.
ttytype	Database of terminal types.
tu	Special file for console cassette interface.
tunefs	Tunes up an existing file system.
turnacct	Switches process accounting on or off.
twinkle	Game that twinkles stars on the screen.
types	Primitive system data types file.
typo	Reports possible typographical errors.
uda	Special file for disk controller interface.
udp	Special file for Internet User Datagram protocol.

ul	Underlining utility.
umask	Sets file-creation mode mask.
umount	Dismounts a file system.
un	Special file for Ungermann Bass interface.
uname	Prints the name of the system you are currently using.
unget	Undoes a previous get of an SCCS file.
uniq	Prints and/or removes repeated lines of text in a file.
units	Standard units of measure conversion program.
unlink	Removes specified links between files.
unmount	Unmounts a mounted file system.
unpack	Unpacks packed files.
untab	Changes tabs into spaces.
up	Special file for DEC Unibus storage module controller/drives.
update	Periodically updates the super block.
updater	Utility for updating files between two machines.
uptime	Prints how long the system has been up.
ut	Special files for tape drive interface.
utmp	Stores user and system accounting information.
users	Produces a compact list of users who are on the system.
uu	Special file for cassette interface.
uuclean	Uucp spool directory clean-up.
uucp	Copies files from one Unix system to another.
uuencode	Encodes or decodes a binary file for transmission via mail.
uulog	Maintains a summary log of uucp and uux transactions.
uuname	Lists uucp names of known systems.
uupick	Searches the PUBDIR for files sent to you via uucp.
uusend	Sends a file to a remote host.
uusnap	Prints a snapshot of uucp processes.
uustat	Prints status of or cancels uucp commands.
uusub	Monitors uucp network.
uuto	Mass or public transfer of files between Unix systems.
uux	Unix-to-Unix command execution utility.
va	Special file for Benson Varian interface.
val	Determines if a file is an SCCS file meeting the characteristics specified.
vc	Copies files with version control (keyword replacement).
versions	Prints out modification dates in an INed structured file.
vfont	Font formats utility.
vfontinfo	Inspects and prints out information about Unix fonts.
vgrind	Grind nice listings of programs.
vgrindefs	Vgrind language definition database.
vi	Screen editor (Version 2.13).
vip	Formats and prints Lisp programs noroff, vtroff, and troff.
vipw	Edits the password file.
vmstat	Reports virtual memory statistics.
vp	Special file for Versatec interface.
vpr	Raster printer/plotter spooler utility.
vsh	Visual shell.
vtroff	Interfaces troff to a raster plotter.
vv	Special file for Proteon proNET 10Mb ring.
vwidth	Makes troff width table for a font.

w	Prints users logged on and what they are doing.
wait	Await completion of process command.
wall	Writes message to all users logged on to the system.
wc	Counts lines, words, and/or characters in a file.
what	Searches a file for patterns identifying it as an SCCS file.
whatis	Describes what a command is.
whereis	Locates source, binary, and/or manual for a program.
which	Locates a program file, including aliases and paths.
who	Lists names of users currently logged on the system.
whoami	Prints current user's login I.D.
whodo	Lists the process each user is doing.
worm	The game of growing worms.
worms	Animated display game.
write	Sends message to other users logged on the system.
wtmp	Stores user and accounting information.
wtmpfix	Corrects time/date stamp in **wtmp** format.
wump	Hunt-the-wumpus game.
xargs	Constructs argument list(s) and executes commands.
xref	Cross-reference for C programs.
xsend	Secret mail utility.
xstr	Extracts strings from C programs.
yacc	Yet another compiler-compiler.
yes	Writes "yes" to output.
zork	A dungeon game.

Appendix C

Unix Utility Cross-Reference Matrix

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
300		●	●			●	●
4014		●	●			●	●
450		●	●			●	●
a.out			●			●	●
a.out.pdp			●				
aardvark				●			
ac	●	●		●			
acc				●			
acct			●	●		●	●
acctcms			●			●	
acctcom		●	●		●	●	●
acctcon						●	
acctcon1		●					
acctcon2		●					

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
acctdisk		●					
acctdusg		●					
acctmerg		●				●	
accton		●			●		
acctprc						●	
acctprc1		●					
acctprc2		●					
acctsh						●	
acctwtmp		●					
ad				●			
adb	●	●	●	●	●	●	●
addbib				●			
adduser				●		●	
admin		●	●		●	●	●
adventure				●			●
aliases				●			
aliens							●
altblk							●
analyz				●			
apropos				●			
ar	●	●	●	●	●	●	●
ar.pdp			●				
arcv			●	●			
arff				●			
arithmetic	●	●	●	●		●	●
arp				●			
ar.pdp			●				
as	●	●	●	●	●	●	●
as.pdp			●				

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
asa			●				●
ascii			●	●		●	●
asktime					●		
assign					●		
asy						●	
at		●	●		●		●
atq					●		
atrm					●		
autoconf				●			
autorobots							●
awk	●	●	●	●	●	●	●
back			●			●	●
backgammon	●	●		●			
badsect				●			
banner	●	●	●	●	●	●	●
bas	●						
basename			●	●	●	●	●
bc	●	●	●	●	●	●	●
bcd	●	●		●			●
bdiff		●	●		●	●	●
bfs		●	●		●	●	●
biff				●			
binmail				●			
bj	●	●	●			●	●
bk				●			
boggle				●			
boot						●	
bugfiler				●			
bs		●	●			●	●

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
cal	●	●	●	●	●	●	●
calendar	●	●	●	●	●	●	●
canfield				●			
cat	●	●	●	●	●	●	●
catman				●			
cb	●	●	●	●	●	●	●
cc	●	●	●	●	●	●	●
cd	●	●	●	●	●		●
cdc		●	●		●	●	●
cflow			●				●
chargefee			●				
chase							●
checkcw			●				
checkeq	●	●					
checkers	●	●					
checklist		●					●
checkmm			●				
checknr				●			
chess	●	●	●	●			
chfn				●			
chgrp	●	●		●	●		
ching	●	●		●			
chmem						●	
chmod	●	●	●	●	●	●	●
chown	●	●	●	●	●	●	●
chparm						●	
chroot					●	●	
chsh				●			
ckpacct		●					

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
clear							●
clri	●	●		●		●	
cmp	●	●	●	●	●	●	●
col	●	●	●	●	●	●	●
colcrt				●			
comb		●	●		●	●	●
comm	●	●	●	●	●	●	●
compact				●			
comsat				●			
config				●		●	
connect						●	
connect.con						●	
cons				●			
convert			●				
copy					●		
core			●	●		●	●
cp	●	●	●	●	●	●	●
cpio		●	●		●	●	●
cpp			●				●
cprs			●				
craps			●			●	●
crash		●		●		●	
cref		●			●	●	
cribbage				●			●
cron	●	●		●	●	●	
crypt	●	●	●	●	●		●
csh				●	●		●
csplit		●	●		●	●	●
css				●			

F8706.54

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
ct		●	●	●			●
ctags				●	●		●
cu	●	●	●		●		●
cut		●	●		●	●	●
cw		●	●		●	●	●
cwcheck					●		
cxref			●				●
dab144				●			
date	●	●	●	●	●	●	●
dbx				●			
dc	●	●	●	●	●	●	●
dcheck	●	●		●			
dd	●	●	●	●	●	●	●
deassign					●		
del						●	
delta		●	●		●	●	●
deroff	●	●	●	●	●	●	●
devinfo						●	
devnm		●			●	●	
df	●	●		●	●	●	
dh				●			
diction				●	●		
diff	●	●	●	●	●	●	●
diff3			●	●	●	●	●
diffmk		●	●		●	●	●
dir			●	●		●	●
dircmp		●	●		●	●	●
dirname		●			●		
dis			●				

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
disable					●		
disk						●	
diskpart				●			
disktab				●			
display						●	
dmc				●			
dmesg				●			
dmf				●			
dn				●			
doctor				●			
dodisk		●					
doscat					●		
doscp					●		
dosdel						●	
dosdir					●	●	
dosls					●		
dosmkdir					●		
dosread						●	
dosrm					●		
dosrmdir					●		
doswrite						●	
dpd			●				
dpr			●				
drtest				●			
drum				●			
dtype					●		
du	●	●	●	●	●	●	●
dump	●	●	●	●	●	●	●
dumpdir					●		

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
dumpfs				●			
dz				●			
e						●	
ebcdic						●	
ec				●			
echo	●	●	●	●	●	●	●
ed	●	●	●	●	●	●	●
edquota				●			
efl		●	●	●			●
egrep					●		
en				●			
enable			●		●	●	●
env		●	●		●	●	●
environ			●	●		●	●
eqn	●	●	●	●	●	●	●
eqnchar			●	●		●	●
eqncheck					●		
errfile			●				●
error				●			
ex				●	●		●
expand				●			
explain				●			
expr	●	●	●	●	●	●	●
eyacc				●			
f77	●	●	●	●			
factor	●	●	●			●	●
false		●		●	●		
fastboot				●			
fcntl			●			●	●

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
fd						●	
fed				●			
ffill						●	
fget			●				
fgrep					●		
file	●	●	●	●	●	●	●
filehdr			●				
filesystems						●	
fill						●	
find	●	●	●	●	●	●	●
finger				●	●		
fish	●	●		●		●	●
fixascii						●	
fl				●			
fmt				●			
fold				●		●	
format				●		●	
fortran							●
fortune	●	●		●		●	●
fp				●		●	
fpr				●			
freq							●
fs			●	●		●	●
fsck				●	●	●	
fsdb		●				●	
fsend			●				
fspec			●			●	●
fsplit			●	●			●
fstab				●			

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
ftpd				●			
fwtmp		●				●	
gcat			●				
gcore				●			
gcosmail			●				
gdev		●	●				
ged		●	●				
get		●	●		●	●	●
getopt			●		●	●	●
gets					●		
gettable				●			
getty				●		●	
gettydefs			●				●
gettytab				●			
ghost						●	
gps			●				●
graph	●	●	●	●		●	
graphics		●	●				
greek	●	●	●			●	●
grep	●	●	●	●	●	●	●
group			●	●		●	●
groups				●			
grpcheck					●		
gutil		●	●				
halt				●			
haltsys					●		
hangman	●	●	●	●		●	●
hd					●	●	
hdr					●		

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
head				●	●		●
help		●	●		●	●	●
hex							●
hier				●			
history						●	
hk				●			
hold						●	
hostid				●			
hostname				●			●
hosts				●			
hp		●	●	●		●	●
hpio			●				●
ht				●			
htable				●			
hy				●			
hyphen		●	●		●	●	●
icheck	●	●					
hy				●			
hyphen		●	●		●	●	●
icheck	●	●		●			
id		●	●		●	●	●
ifconfig				●			
ik				●			
il				●			
imp				●			
implog				●			
implogd				●			
indent				●			
ined						●	

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
inet				●			●
init				●		●	
inittab			●				●
inode			●			●	●
install		●		●		●	
iostat	●	●		●			
ip				●			●
ipcrm			●				●
ipcs			●				●
issue			●				●
istat						●	
join	●	●	●	●	●	●	●
jotto			●				
just						●	
kasb			●				
keyboard						●	
kg				●			
kgmon				●			
kill	●	●	●	●	●	●	●
killall						●	
kmem						●	
l					●	●	
last				●			●
lastcomm				●			
lastlogin		●					
lc					●		
ld	●	●	●	●	●	●	●
ld.pdp			●				
ldfcn			●				

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
learn	●	●		●	●		
leave				●			
lex	●	●	●	●	●	●	●
li						●	
life							●
line			●		●	●	●
linenum			●				
link		●	●	●		●	
lint	●	●	●	●	●	●	●
lisp				●			
liszt				●			
ln	●	●		●	●		
lo				●			●
lock				●			
login			●	●		●	●
logname		●	●		●	●	●
look	●	●		●	●		
lookbib				●			
lorder	●	●	●	●	●	●	●
lp			●	●		●	●
lpc				●			
lpd				●			
lpq				●			
lpr	●	●	●	●	●		●
lprm				●			
lpstat			●				●
ls	●	●	●	●	●	●	●
ls7							●
lxref				●			

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
m4	●	●	●	●	●	●	●
machid			●				●
mail	●	●	●	●	●	●	●
mailaddr				●			
make	●	●	●	●	●	●	●
makedev				●			
makekey			●	●		●	●
man	●	●	●	●		●	●
manroff						●	
mant						●	
master						●	●
master.dec			●				
master.u3b			●				
maze	●	●	●				●
me				●			
mem				●		●	
memuse						●	
mesg	●	●	●	●	●	●	●
mille				●			
mkdir	●	●	●		●	●	●
mkfs	●	●		●	●	●	
mklost + found				●			
mknod	●	●		●	●	●	
mkproto				●			
mkstr				●	●		●
mkuser					●		
mm		●	●		●	●	●
mmcheck		●			●		
mmt		●	●		●	●	●

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
mnttab			●			●	●
monacct		●					
monop				●			
moo	●	●	●			●	●
more				●	●		●
mosd			●				●
mount	●	●	●	●	●	●	
mptx			●				●
ms				●			
msgs				●			
mt				●			
mtab				●			
mtio				●			
mv	●	●	●	●	●		●
mvdir		●				●	
ncheck				●	●	●	
neqn	●	●			●		
net			●				●
netstat				●			
netutil					●		
newaliases				●			
newfile						●	
newform		●					●
newfs			●				
newgrp	●	●	●		●	●	●
news		●	●			●	●
nice	●	●	●	●	●	●	●
nl		●	●		●	●	●
nm	●	●	●	●	●	●	●

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
nm.pdp			●				
nohup	●	●	●		●	●	●
nroff	●	●	●	●	●		●
nroff7							●
nscstat			●				
nsctorje			●				
null				●		●	
number		●		●		●	●
nusend			●				
od			●	●	●	●	●
pac				●			
pack		●	●		●	●	●
pagesize				●			
panic						●	
param						●	
passwd			●	●	●	●	●
paste		●	●		●	●	●
pc				●			
pcat		●			●		
pcl				●			
pdx				●			
phones				●			
pi				●			
pix				●			
plot	●	●	●	●		●	●
pmerg				●			
pnch			●				●
ports						●	
portstatus						●	

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
pr	●	●	●	●	●	●	●
prctmp		●					
prdaily		●					
prep					●		
primetime						●	
print				●		●	
printcaps				●			
printenv							●
prmail				●			
prof	●	●	●	●	●	●	●
profile						●	●
proto						●	
protocols				●			
prs		●	●		●	●	●
prtacct		●					
ps	●	●	●	●	●	●	●
pstat				●	●		
pti				●			
ptx	●	●	●	●	●	●	●
pty			●				
pup			●				
put							●
put7							●
pwadmin					●		
pwck		●					
pwcheck				●			
pwd	●	●	●	●	●	●	●
px				●			
pxp				●			

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
pxref				●			
qconfig						●	
qdaemon						●	
quiz	●	●	●	●		●	●
quot	●	●		●	●		
quota				●			
quotacheck				●			
quotaon				●			
rain				●			●
random					●		
ranlib				●	●		
ratfor	●	●	●	●	●		
rc				●		●	
rcp				●	●		●
rcvhex							●
rdump				●			
readfile						●	
reboot				●			
refer	●	●		●			
reform						●	
regcmp		●	●		●	●	●
regexp			●			●	●
reloc			●				
remote				●	●	●	
remsh							●
renice				●			
repquota				●			
reset				●			●
restor	●	●		●	●		

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
restore						●	
rev				●			
reversi	●	●	●				
rexecd				●			
rjestat			●				
rlogin				●			●
relogind				●			
rm	●	●	●	●	●	●	●
rmail		●		●	●	●	
rmdel		●	●		●	●	●
rmdir	●	●		●	●		
rmhist						●	
rmt				●			
rmuser					●		
robots							●
roff	●						
roffbib				●			
rogue				●			
route				●			
routed				●			
rpl						●	
rrestore				●			
rsh		●		●	●		
rshd				●			
rstat							●
runacct		●				●	
ruptime				●			●
rwho				●			●
rwhod				●			

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
rx				●			
rxformat				●			
sa	●	●		●			
sact			●		●		●
sadp			●				●
sag		●	●				●
sar			●				●
sash						●	
savcore				●			
scat			●			●	
scc			●			●	
sccsdiff		●	●		●	●	●
sccsfile			●			●	●
scnhdr			●				
script				●			
sdb			●				
sddate					●		
sdiff		●	●		●	●	●
se			●				●
sed	●	●	●	●	●	●	●
see							●
send			●				
sendbug				●			
sendmail				●			
services				●			
setmnt		●			●		
settime					●		
sh	●	●	●	●	●	●	●
shutacct		●					

Utility	7	HI	V	4.2	Xenix	PC/IX	Uni-Plus +
shutdown		●		●	●	●	
size	●	●	●	●	●	●	●
size.pdp			●				
skulker						●	
sky			●				
sleep	●	●	●	●	●	●	●
snake				●			
sno			●			●	●
soelim				●	●		
sorry						●	
sort	●	●	●	●	●	●	●
sortbib				●			
spell	●	●	●	●	●	●	●
spline	●	●	●	●	●	●	●
split	●	●	●	●	●	●	●
splp						●	
ssp							●
stab				●			
stackuse					●		
stat		●	●			●	●
sticky				●			
stlogin			●				
strings				●	●		●
strip	●	●	●	●	●	●	●
strip.pdp			●	●			
struct	●	●		●			
ststat			●				
stty	●	●	●	●	●	●	●
style				●	●		

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
su	●	●	●	●	●	●	●
subset		●				●	
sum	●	●	●	●	●	●	●
sum7							●
sumdir							●
swapon				●			
symorder				●			
syms			●				
sync	●	●	●	●	●	●	●
sysadmin					●		
syslog				●			
system			●				
tab						●	
tabs	●	●	●	●		●	●
take							●
take7							●
tail	●	●	●	●	●	●	●
talk				●			
tar	●	●	●	●	●		●
tbl	●	●	●	●	●	●	●
tc	●	●	●	●		●	●
tcp				●			●
tee	●	●	●	●	●	●	●
telnet				●			
telnetd				●			
term			●	●		●	●
termcap				●	●	●	●
test	●	●	●	●	●	●	●
tftpd				●			

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
time	●	●	●	●	●	●	●
timex		●	●			●	●
tip				●			
tm				●			
toc			●				
touch			●	●	●	●	●
tp	●	●		●			●
tplot		●	●			●	●
tr	●	●	●	●	●	●	●
trek				●			●
trman				●			
troff	●	●	●	●	●	●	●
troff7							●
trouble			●				
trpt				●			
true			●	●	●	●	●
ts				●			
tset				●	●		●
tsort	●	●	●	●	●	●	●
ttt	●	●	●		●	●	●
tty	●	●	●	●	●	●	
ttys				●			
ttytype				●			●
tu				●			
tunefs				●			
twinkle							●
types			●	●		●	●
typo	●						
uda				●			

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
udp							●
ul				●			●
umask		●	●		●	●	●
umount	●	●			●		
un				●			
uname		●	●		●	●	●
unget		●	●		●	●	●
uniq	●	●	●	●	●	●	●
units	●	●	●	●	●	●	●
unlink		●					
unmount						●	
unpack		●			●		
untab						●	
up				●			
update				●			
updater							●
uptime				●			
ut				●			
utmp			●	●		●	●
users				●			
uuclean				●		●	
uucp	●	●	●	●	●	●	●
uuencode				●			
uulog					●		
uuname	●	●					
uupick	●	●					
uusend			●	●			
uusnap				●			
uustat	●	●	●	●		●	●

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
uusub						●	
uuto	●	●	●			●	●
uux				●	●	●	●
va				●			
val	●	●	●		●	●	●
vc	●	●	●			●	●
versions						●	●
vfont				●			
vfontinfo				●			
vgrind				●			
vgrindefs				●			
vi				●	●		●
vip				●			
vipw				●			
vmstat				●			
vp				●			
vpr			●	●			
vsh					●		
vtroff				●			
vv				●			
vwidth				●			
w				●			
wait	●	●	●	●	●	●	●
wall	●	●		●	●		
wc	●	●	●	●	●	●	●
what		●	●	●	●	●	●
whatis		●		●			
whereis				●			
which				●			

Utility	7	III	V	4.2	Xenix	PC/IX	Uni-Plus +
who	●	●	●	●	●	●	●
whoami				●			
whodo		●			●		
worm				●			●
worms				●			●
write	●	●	●	●	●	●	●
wtmpfix		●					
wtmp	●	●	●	●		●	●
xargs		●	●		●	●	●
xref					●	●	
xsend				●			
xstr				●	●		
yacc	●	●	●	●	●	●	●
yes				●	●		
zork				●			

Appendix D

The Xenix System

As an example of Unix-based systems, this appendix presents a detailed examination of the very popular Xenix operating system. This material appeared originally as *Microsoft Xenix Operating System Version 3.0 Release Summary*, and is reprinted here with the kind permission of Microsoft Corporation. Xenix is a registered trademark of Microsoft.

PRODUCT OVERVIEW

Xenix Version 3.0 is a significantly enhanced version of the Bell Labs Unix System III Operating System. It is derived from the Bell source distribution, with modifications and enhancements to tailor the system to the microcomputer environment. Version 3.0 represents a significant step forward, both in the quality and functionality of the software, and in the documentation.

The product is provided as three packages, and the documentation is structured to reflect this. All the manuals are produced in 8.5 x 5 inch "downsize" format.

Software

Xenix Version 3.0 is provided as three packages. The *Timesharing System* contains the Xenix system kernel, plus a large number of standard utilities. This package is sufficient to provide an effective multi-user environment. The *Software Development System* contains compilers, the linker, and a number of other utilities useful for program development. It also contains the C libraries, and **include** files. Finally, the *Text Processing System* contains the text formatters and macro packages, and a number of other useful utilities.

The Timesharing System is required to use either of the other two packages, but the two add-on packages are independent of each other, both in documentation and software.

Documentation

A detailed description of the documentation is given later in this appendix. Each of the three packages comes with its own independent set of documentation. The two add-on packages also contain reference manual insert pages, so that the reference manual in the Timesharing System can be upgraded easily.

Online documents and manual pages are no longer provided. Documentation is provided in Laser Printer output. In special cases the documents and the mm macros will be provided. Significant modification may be required to adapt the documents to other laser printers.

NEW FEATURES

Shared Data. A new system call will be added to allow user processes to share data areas. This will be implemented on all systems, regardless of the memory management model. However on some systems the performance will be better than on others.

Fixed Stack Analysis Utilities. A set of utility programs will allow analysis of C programs to determine stack size requirements. This is useful when developing software for fixed stack machines (eg unmapped 8086, 286, and some M68000 systems).

Inter-Machine Mailer. The mailer has been completely replaced with a significantly enhanced product. The new mailer has a user interface based on the Berkeley mail program, and is integrated with a new communications package to send mail between local machines over serial lines. Using this users can network several machines together reliably. This package replaces uucp for local machine communications, and also allows remote command execution and inter-machine file transfer.

System Administration Utilities. A number of utility programs have been added to the Xenix System to make system administration easier. For example, adding and deleting user accounts can now be done with a single command.

Visual Shell. The visual shell will be provided in the Timesharing System. This shell runs under both the Xenix System and under MS-DOS, and provides a closely similar user interface in both cases. It is a menu-driven command interpreter which makes full use of the screen to display status and environment information to the user. It has a built-in help facility, and users can add new applications to the menu. The command interface is modeled after the Microsoft Multi-Tools, and therefore is easy to learn by nontechnical users.

MS-DOS File Access Utilities. Several utilities will be provided in Xenix Version 3.0 to allow MS-DOS files and directories to be read and written. This will be especially useful for machines which can operate both MS-DOS and the Xenix System. Access to IBM DOS 1.1 and 2.0 format diskettes will be supported.

Secure Boot Sequence. The standard boot sequence under Xenix Version 3.0 prevents entering single-user mode without knowing the super-user password. This closes a significant security hole.

Password Administration. The system can now be set up to enforce password aging on a per-user basis. In addition, a new command pwadmin is provided for making changes to the password file.

Source Code Control System. The SCCS package is provided with the Software Development System. This consists of the following new commands: admin, cdc, comb, delta, help, prs, rmdel, sccsdiff, and unget.

Memorandum Macro Package. The memorandum macros and the new mm command are provided with the Text Processing System. These are a significant functional improvement over the ms macros in Xenix Version 2.3.

System Calls. Xenix Version 3.0 contains all the Xenix Version 2.3 system calls, plus all those in AT&T's System III product. In addition, the following are new:

shared data	As mentioned above, a call will be provided to allow unrelated processes to share data.
chsize	A system call to truncate files to a given length.
nap	A new system call to allow a process to sleep for very short periods of time. This is useful for interactive, screen oriented packages.
lock	A new system call to allow processes to lock themselves in physical memory to guarantee a greater share of machine resources.

Language Tools. The initial Xenix Version 3.0 release will contain a new compiler with the Unix System III language extensions. This compiler will support large text and large data on Xenix-286. It will also support individual data items greater than 64K.

The assembler provided with 286 systems does not support generation of 286-specific instructions, but can be made to do so using one of the macroprocessors provided with the Software Development Package. Xenix 286 includes an 80287-compatible floating point emulator or support for the 80287 floating-point hardware.

COMPATIBILITY

Systems previously supplied as Version 2.3 will continue to support execution of old binaries. A compile-time option will allow compilation of Version 2.3 sources also. Thus, all Xenix Version 2.3 binaries and source code are usable under Xenix Version 3.0 without modification. Xenix Version 2.3 file systems can be used with systems. The **fsck** program should be used on the file system before use with a system.

There are a few exceptions to the above. Any utilities which make use of detailed internal knowledge of the kernel or file system format will need modification. It is not expected that there will be any of these outside the standard Xenix System utilities.

HARDWARE REQUIREMENTS

The absolute minimum hardware requirements for Xenix Version 3.0 are as follows:

- 512K bytes of main memory.
- 10M bytes of hard disk storage.
- One backup device (magtape or floppy disk).

This minimum hardware is sufficient to support the full Xenix Version 3.0 system and run all the utilities.

It is important to note that the exact amount of memory required on a given system depends on usage patterns and the specific application packages added. Thus the above system is sufficient for a small number of users using the standard Xenix utilities, but might not be enough to support a large number of users, or a large and sophisticated application package. These figures are minimums.

It is possible that a single-user system running just the Timesharing Package with simple and small applications could run with slightly less disk and memory. However, not all the Timesharing utilities will run with reasonable performance in a system with less main memory. In particular, using the inter-machine mail system is equivalent to running multi-

user, since mail can arrive synchronously. Use of this facility definitely requires a 512K system, as does any other Timesharing system with any background processing.

No system without a hard disk will be able to run Xenix Version 3.0.

DETAILED SUMMARY

The next few sections list in detail the specific system calls, library routines, and utility commands available under, broken down by individual packages. Commands marked * are new in Xenix Version 3.0.

Timesharing System

The Timesharing System contains the Xenix System kernel, and the commands listed below. *[Refer to Appendix B for capsule summaries of the command functions.]*

*acctcom	diff	ln
accton	diff3	*logname
asktime	*dircmp	look
assign	*dirname	lpr
at	disable	ls
*atq	*dtype	mail
*atrm	du	mesg
awk	dump	mkdir
*banner	dumpdir	mkfs
basename	echo	mknod
bc	ed	*mkuser
*bdiff	egrep	more
bfs	enable	mount
cal	*env	mv
calendar	ex	ncheck
cat	expr	*netutil
cd	false	newgrp
chgrp	fgrep	nice
chmod	file	*nl
chown	find	nohup
*chroot	finger	od
cmp	fsck	*pack
comm	*getopt	passwd
copy	grep	*pcat
cp	*grpcheck	pr
*cpio	haltsys	ps
cron	*hd	pstat
crypt	head	*pwadmin
*csplit	*id	*pwcheck
cu	join	pwd
date	kill	quot
deassign	l	random
dc	lc	*rcp
dd	ld	*remote
*devnm	learn	restor
df	*line	rm

rmail
rmdir
*rmuser
*rsh
sddate
*sdiff
sed
*setmnt
settime
sh
shutdown
sleep
sort
split
stty
su
sum
sync
*sysadmin
tail
tar
tee
test
touch
tr
true
tset
tty
umask
umount
*uname
uniq
units
*unpack
vi
vsh
wait
wall
wc
what
who
*whodo
write
*xargs
yes

Note that the **learn** command contains lessons on the following subjects:

- Files and the Xenix System file system. The two lessons are called **files** and **morefiles**.
- **Macros**. This lesson describes the use of the **ms** macro package.
- **C**. This provides an introduction to the C language.
- **Editor**. This describes the the Xenix System line editor **ed**.

Note also that all lessons are provided with the Xenix System Timesharing System, even though some refer to programs present in the Text Processing and Software Development Packages. Also note that the **ms** macro package is the one described, not **mm**.

Text Processing System

The Text Processing System contains several text formatting programs, and three macro packages for document preparation.

Commands. *[Listed below are the text processing commands supported under Xenix Version 3.0. For capsule descriptions of functions see Appendix B.]*

col
*cut
*cw
*cwcheck
deroff
*diction
*diffmk
eqn
eqncheck
*hyphen
*mm
*mmcheck
*mmt
neqn
nroff
*paste
prep
ptx
*soelim
spell
*style
tbl
troff

Macro Packages. The macro packages supported under Xenix Version 3.0 are as follows:

mm The *Memorandum* macros are the standard method for producing formatted documents under Xenix Version 3.0. These macros are documented and presented as the standard Xenix System macro package.

ms The *Manuscript* macros are those provided with Xenix Version 2.3. They are less powerful and less easy to use than the Memorandum macros but, since many existing documents are in this format, the macros are provided so these documents can still be processed. These macros are not documented, and are not intended for the production of new documents.

man The *Manual* macros are used for formatting outline manual pages. Although online manual pages are not provided for the standard Xenix System software, additional applications may include documentation in this format, so these macros are provided. These macros are supplied purely for processing foreign documentation, and are not documented.

Software Development System

The Software Development System contains commands, library routines, and interfaces to the kernel.

System Calls. *[Listed below are system calls supported under Version 3.0; an* ***asterisk*** *(*) marks those new with this release.]*

access	Determines the accessibility of a file.
acct	Enables or disables process accounting.
alarm	Sets a process's alarm clock.
sbrk	Changes data segment space allocation.
chdir	Changes the working directory.
chmod	Changes the permission mode of a file.
chown	Changes the owner and group of a file.
chroot	Changes the root directory.
*chsize	Changes file size.
close	Closes a file descriptor.
creat	Creates a new file or rewrites an existing one.
creatsem	Creates an instance of a binary semaphore.
dup	Duplicates an open file descriptor.
dup2	Duplicates an open file descriptor.
execl	Executes a file.
exit	Terminates a process.
fcntl	File control utility.
fork	Creates a new process.
fstat	Gets file status.
ftime	Gets system time.
getpid	Gets process I.D.
getpgrp	Gets process group.
getppid	Gets parent process I.D.
getuid	Gets real user I.D.
geteuid	Gets effective user I.D.
getgid	Gets group I.D.
getegid	Gets effective group I.D.
ioctl	Device control utility.
kill	Sends a signal to a process or group of processes.
link	Links to a file.

*lock	Locks a process in memory.
locking	Locks or unlocks a file region for reading or writing.
lseek	Moves a read/write file pointer.
mknod	Makes a file.
mount	Mounts a file structure.
*nap	Sleep for a short time.
nice	Changes priority of a process.
open	Opens a file for reading or writing.
opensem	Opens a semaphore.
pause	Suspends a process until signal.
pipe	Creates an interprocess channel.
profil	Execution time profile utility.
ptrace	Process trace utility.
rdchk	Checks if there is data to be read.
read	Reads from a file.
*sdget	Attaches to a shared data region.
*sdfree	Releases a shared data region.
*sdgetv	Synchronizes the use of shared data.
*sdenter	Enters a shared data region.
*sdleave	Leave a shared data region.
*sdwaitv	Synchronizes use of shared data.
setpgrp	Sets process group I.D.
setuid	Sets user I.D.
setgid	Sets group I.D.
shutdn	Utility to flush block I/O and halt system.
signal	Specifies what to do on receipt of a signal.
sigsem	Signals a process waiting on a semaphore.
stat	Gets file status.
stime	Sets time.
sync	Updates super block.
time	Gets time.
times	Gets process and child process times.
ulimit	Gets and sets user limits.
umask	Sets and gets file creation mask.
umount	Unmounts a file system.
uname	Gets name of current system.
unlink	Removes a directory entry.
ustat	Gets file system statistics.
utime	Sets file access and modification times.
wait	Waits for child process to stop or terminate.
waitsem	Waits on a semaphore.
write	Writes on a file.

Library Routines. The following libraries are provided as standard with Xenix Version 3.0. On 8086/88 and 286 systems, versions for Small, Middle, and Large model programs will be provided (i.e., three of each library). They are included at link time by specified **-lname** to the compiler or linker, where **name** is the name listed below less the **lib** prefix. For example **-lm**, and **-ltermcap**.

libc	The standard library containing all system call interfaces,

	standard I/O routines, and other general-purpose services.
libm	The standard math library.
libl	Library for use with programs produced by **lex**.
liby	Library for use with programs produced by **yacc**.
libtermcap	Routines for accessing the **termcap** data base describing terminal characteristicsl.
libtermlib	The same as **libtermcap**.
libcurses	Screen and cursor manipulation routines.
libdbm	Database management routines.

The Standard C Library (**libc**) consists of the following.

_tolower	Converts to lowercase.
_toupper	Converts to uppercase.
a641	Converts base-64 ASCII to long integer.
abort	Generates an IOT fault.
abs	Integer absolute value function.
asctime	Convert time data to ASCII.
assert	Program verification utility.
atof	Converts ASCII string to floating number.
atoi	Converts ASCII string to integer.
atol	Converts ASCII string to long integer.
bsearch	Binary search utility.
calloc	Allocates memory.
clearerr	Clears error.
crypt	DES encryption utility.
ctermid	Generates file name for terminal.
ctime	Converts time to ASCII string.
cuserid	Character login name of user utility.
defopen	Opens default parameter file.
defread	Reads default parameters.
ecvt	Format conversion utility.
encrypt	DES encryption utility.
endgrent	Closes group file.
endpwent	Closes password file.
fclose	Closes a stream.
fcvt	Format conversion utility.
fdopen	Reopens a stream.
feof	Tests for end-of-file condition.
ferror	Tests for error.
fflush	Flushes a stream.
fgetc	Gets a character from a stream.
fgets	Gets a string from a stream.
fileno	Converts stream number to file descriptor.
fopen	Opens a stream.
fprintf	Formatted output routine.
fputc	Writes a character to a stream.
fputs	Writes a string to a stream.
fread	Buffered input utility.
free	Free memory function.

freopen	Reopens a stream.
frexp	Returns mantissa.
fscanf	Formatted input conversion utility.
fseek	Seeks within a stream.
ftell	Obtains file pointer position.
fwrite	Buffered output utility.
fxlist	Gets name list entries from a file.
gcvt	Format conversion function.
getc	Gets a character from a stream.
getchar	Gets a character from a stream.
getenv	Gets a value for an environment variable.
getgrent	Gets group file entry.
getgrgid	Gets group file entry.
getgrnam	Gets group file entry.
getlogin	Gets login name.
getopt	Parses command line options.
getpass	Reads a password.
getpw	Gets name from user I.D.
getpwent	Gets password file entry.
getpwnam	Gets password file entry.
getpwuid	Gets password file entry.
gets	Gets a string from a stream.
getw	Gets a word from a stream.
gmtime	Obtains Grenwich Mean Time information.
gsignal	Software signal utility.
isalnum	Tests for alphanumeric character.
isalpha	Tests for alphabetic character.
isascii	Tests for ASCII character.
isatty	Checks for terminal.
iscntrl	Tests for a control character.
isdigit	Tests for a digit.
isgraph	Tests for a printing character.
islower	Tests for lowercase.
isprint	Tests for printing character.
ispunct	Tests for punctuation.
isspace	Test for space character.
isupper	Tests for uppercase.
isxdigit	Tests for hexadecimal digit.
l3tol	Converts a 3-byte integer to long integers.
l64a	Converts a long integer to base-64 ASCII.
ldexp	A useful function.
localtime	Obtains local time information.
logname	Gets login name of user.
longjmp	Nonlocal GOTO statement.
lsearch	Linear search and update function.
ltol3	Converts long integer to 3-byte integer.
malloc	Allocates memory.
mktemp	Makes a temporary file.
modf	Returns fractional part of a division.
monitor	Prepares an execution profile.
nlist	Gets entries from a name list.
pclose	Closes the pipe to a process.

perror	Prints system error messages.
popen	Initiates I/O to/from a process.
printf	Formatted output routine.
putc	Writes a character to a stream.
putchar	Writes a character to a stream.
putpwent	Writes password file entry.
puts	Writes a string to a stream.
putw	Writes a word to a stream.
qsort	Quick sort routine.
rand	Random number generator.
realloc	Reallocates memory.
regcmp	Regulars expression compile function.
regex	Regular expression execute function.
rewind	Seek to zero function.
scanf	Formatted input conversion utility.
setbuf	Assigns buffering to a stream.
setgrent	Rewinds group file pointer.
setjmp	Nonlocal GOTO statement.
setkey	DES encryption.
setpwent	Rewinds password file pointer.
sleep	Suspends execution for an interval.
sprintf	Formatted output routine.
srand	Seeds the random number generator.
sscanf	Formatted input conversion function.
ssignal	Software signal.
strcat	Concatenates strings.
strchr	Finds a character in string.
strcmp	Compares strings.
strcpy	Copies strings.
strcspn	Finds the length of a substring.
strlen	Gets string length.
strncat	Concatenates strings.
strncmp	Compares strings.
strncpy	Copies strings.
strpbrk	Finds a string in another string.
strrchr	Finds a character in a string.
strspn	Finds the length of a substring.
strtok	Finds a token within a string.
swab	Swaps bytes.
system	Executes a shell command.
tmpfile	Creates a temporary file.
tmpnam	Creates a temporary file name.
toascii	Converts to ASCII.
tolower	Converts to lowercase.
toupper	Converts to uppercase.
ttyname	Finds the name of a terminal.
tzset	Sets external time variables.
ungetc	Pushes character back onto stream.
xlist	Gets name list entries from a file.

The Standard Math Library (libm) consists of:

acos	Arc cosine function.
asin	Arc sine function.
atan	Arc tangent function.
atan2	Arc tangent function.
cabs	*Euclidean distance function.*
ceil	Ceiling value function.
cos	Cosine function.
cosh	Hyperbolic cosine function.
exp	Exponentiation operator.
fabs	Absolute value function, returns $\lvert\chi\rvert$.
floor	Absolute value function.
fmod	A useful function.
gamma	Log gamma function.
hypot	Finds the (square root of $(x^2 + y^2)$.
j0	Bessel function.
j1	Bessel function.
jn	Bessel function.
log	Natural logarithm.
log10	Common (base-10) logarithm.
pow	Power function.
sin	Sine function.
sinh	Hyperbolic sine function.
sqrt	Square root function.
tan	Tangent function.
tanh	Hyperbolic tangent function.
y0	Bessel function.
y1	Bessel function.
yn	Bessel function.

The Default Lexical Analysis Library (**libl**) consists of:

main	**lex** program entry.
yyless	**lex** routine.
yywrap	**lex** routine.

The Default **yacc** Library (**liby**) consists of:

main	**yacc** program entry.
yyerror	**yacc** error handler.

The Terminal Capabilities Library (**libtermcap**) consists of the following.

tgetent	Gets terminal capability entry.
tgetflag	Tests for presence of capability.
tgetnum	Gets numeric value of capability.
tgetstr	Gets string value of capability.
tgoto	Gets cursor addressing string.
tputs	Decodes padding information.

The Screen Manipulation Library (**libcurses**) consists of:

curses	Many screen cursor manipulation routines.

The Data Base Management Library (**libdbm**) includes:

dbminit	Opens the data base.
delete	Deletes a key in the data base.
fetch	Accesses a key in the data base.
firstkey	Gets the first key in the data base.
nextkey	Gets the next key in the data base.
store	Store a key in the data base.

Commands. *[Listed below are the commands available in the Version 3.0 Software Development System; an asterisk marks those that are new with this release. Capsule descriptions of these commands may be found in Appendix B.]*

adb	***hdr**	***sccsdiff**
***admin**	**lex**	**size**
ar	**lint**	**spline**
as	**lorder**	**stackuse**
cb	**m4**	**strings**
cc	**make**	**strip**
***cdc**	**mkstr**	**time**
***comb**	**nm**	**tsort**
***cref**	**prof**	***unget**
csh	***prs**	**uucp**
ctags	**ranlib**	**uulog**
***delta**	**ratfor**	**uux**
***get**	***regcmp**	**val**
gets	***rmdel**	***xref**
***help**	***sact**	**xstr**
		yacc

MS-DOS Commands

The following commands are available on all versions of the Xenix System, but will not necessarily be appropriate on some, and so may not be provided.

doscat	Concatenates a file on an MS-DOS floppy disk.
doscp	Copies files to from MS-DOS floppy disks.
dosdir	Lists directory of MS-DOS floppy disk.
dosls	Lists directory of MS-DOS floppy disk.
dosmkdir	Creates an MS-DOS directory on an MS-DOS disk.
dosrm	Deletes an MS-DOS file.
dosrmdir	Deletes an MS-DOS directory.

Unsupported Commands

Some of the Unix code provided to Microsoft is in an undocumented form, which makes it impossible to provide as complete support as we would otherwise wish. In addition, some software produces output for certain devices (e.g., graphics plotters and typesetters) to which Microsoft does not have access. It is not possible for us to verify the operation of this Unix code.

Index

A
alias (of file name), 45
Altos Computer Company, 13
application program, 3, 12, 22
argument, command, 59
ASCII, 29
AT&T Unix, 3
  System III, 20
  System V, 13, 20

B
background process, 68
BASIC, 25, 29
Bell Laboratories, 9
Berkeley C shell, 49
bin (directory), 38
block, disk, 27
Bourne shell, 49
buffer, 66
byte, 28

C
C (programming language), 62
cat (concatenate) command, 182
cd (change directory) command, 42, 95, 181
central processing unit (CPU), 4
chmod (change mode) command, 43, 130, 184
chown (change owner) command, 184
Christian, Kaare, 166
COBOL, 25
command, 5, 55
  arguments, 59, 60
  cat, 182
  cd, 42, 95, 181
  chmod, 43, 130, 184
  chown, 184
  conditional, 62, 76
  cp, 183
  date, 182
  echo, 182
  entry of, 85
  expr, 187
  formats of, 58
  grep, 151, 185
  halting execution of, 86
  ln, 183
  ls, 97, 178
  mkdir, 183
  more, 67, 184
  mv, 182
  options, 59
  publically accessible, 62
  pwd, 94, 181
  redirect data, 68
  redirected input, 144
  redirection, 70
  shell, 62
  sort, 185
  terminology, 59
  types of, 61
  wc, 187
  who am i, 93, 181
command file, 74
command line, 64
  multiple command, 65, 145
computer system, 2
Computer Training Centers, 84
control key, 87
cp (copy) command, 183

D
"daisy chain", 30
date command, 182
dev (directory), 39
Digital Equipment Corporation, 12
directory
  alphanumeric indexed, 33
  bin, 38
  creation of, 120
  dev, 39
  etc, 39
  home, 33, 94
  in hierarchical structure, 35
  information in, 36
  removal of, 170
  root, 38
  tmp, 39
  top-level, 38
  usr, 38
  working, 94
disk
  block structure, 27
  configuration of, 28
  data storage on, 27
  i-node, 27
  track structure, 27

E
echo command, 182
etc (directory), 39

expr (expression) command, 187

F
file, 29
categories of, 29
changing ownership, 135
contents of, 30
copy permission, 133
creation of, 109
creation with pathname, 124
directory, 29
executable, 29
group overlay, 44
linking, 45, 139
mountable, 47
naming conventions, 31
ordinary, 29
"public" access to, 44
registered owner, 43
removal of, 170
search procedure, 57
security, 42
size limit, 31
special, 30
system, 29
text/data, 29
with appended redirection, 115
with move command, 117
with redirection command, 110
file structure, hierarchical, 7, 32
filter, 65, 67, 146, 151
flag, 59
FORTRAN, 25, 29, 62
Fortune Computer Company, 78
free list (disk), 18

G
grep command, 185
Groff, James, 78

H
Halamka, John, 165
hardware, system, 2, 14

I
i-node, disk, 27
input/output, redirection of, 7

K
kernel, 16, 17, 18
Khaw, Jack, 84
Korn shell, 49
Korn, David, 49

L
ln (link) command, 183
log off, 92
login acknowledgment, 91
login name, 88
syntax rules, 89
ls (list) command, 97, 178

M
mass storage device, 26
memory
auxiliary, 4, 26
random access, 4, 27
metacharacter, 62, 63, 74, 144
changing directory with, 105
in file name, 32
table of, 75
mkdir (make directory) command, 183
mode bit, 42
more command, 184
multi-tasking, 7, 66
mv (move) command, 182

O
operating system, 2, 3, 15
functions, 4
list of, 4

P
pagination, with more command, 67
Pascal, 25, 29
passwd command, 39, 90
password, 88
syntax rules, 90
pathname, 35, 40, 94
relative, 42
PC-DOS, 12
PC/IX, 9, 50
permission bit, 42
pipe, 9, 65, 66, 144, 146
pipe line, 65, 67, 146, 148
porting, 9, 16
primate, 17
process, background, 68
PWB/UNIX, 50
pwd (print working directory) command, 94, 181

R
Real World Unix, 165
redirect input command, 144
root directory, 38

S
security
file-level, 129
system-level, 88
semicolon, as command, 73
shell, 20, 48
application program, 51
command language, 63
file search procedure, 57
interaction with commands, 58
menu, 50
operation of, 51
restricted, 49
selection of, 55
system, 49
shell program, 13, 16, 17, 157
using pipe, 166
shell programming, 9
shell prompt, 85, 91
shell script, 74
shell variable, 159
non-numeric, 162
software, 2
levels of, 22
sort command, 185
subcommand, 66, 70
subroutine library, 62
super-user, 85, 91, 138
system call, 16, 18, 20, 62

T
tee, 65, 68, 144, 156
timesharing, 51
tmp (directory), 39
track, disk, 27

U
Understanding Unix, 78
UniPlus+, 50, 55
Unix for Users, 6
Unix system
add-ons, 13
components of, 16
features, 7
generic, 7
nomenclature, 13
size of, 12
utility categories, 23
Unix, AT&T, 3, 9
System III, 9
System VI, 9
Version 7, 9
Unix, Berkeley, 9
Unix-based system, 9
differences with AT&T Unix, 11
Unix-like system, 12
user-friendliness, 9
usr (directory), 38
utility, 5
categories of, 23
Unix, 21, 22
user-created, 13

V
variable, shell, 159

W
wc (word count) command, 187
Weinberg, Paul, 78
who am i command, 93, 181
wild card, 151, 170

X
Xenix, 9, 13, 55

Edited by Stephen Moore

Document generated by Anna's Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):
{
  "filename": "NDAxNjc5NjMuemlw",
  "filename_decoded": "40167963.zip",
  "filesize": 14815184,
  "md5": "753e4dac7d3f99a52a0a64ad7067cc7b",
  "header_md5": "b21c888fed89573c1de9b44be99bea49",
  "sha1": "4a4b77b814acccfd3387d5c25faad49b118100a4",
  "sha256": "f210bb41a8081928d731ffa39f381d160213e8197b9823e2208f0cb706a1bf96",
  "crc32": 2917723362,
  "zip_password": "6622Ee",
  "uncompressed_size": 15514599,
  "pdg_dir_name": "40167963_UNIX AND XENIX DEMYSTIFIED_p243",
  "pdg_main_pages_found": 243,
  "pdg_main_pages_max": 243,
  "total_pages": 257,
  "total_pixels": 1487773441,
  "pdf_generation_missing_pages": false
}