92245

AT&T

THE DESIGN OF THE UNIX® OPERATING SYSTEM

Maurice J. Bach

PHI

Prentice/Hall International, Inc.

Published by Prentice-Hall, Inc.
A division of Simon & Schuster
Englewood Cliffs, New Jersey 07632

Prentice-Hall Software Series
Brian W. Kernighan, Advisor

This document was set on an AUTOLOGIC, Inc. APS-5 phototypesetter driven by the TROFF formatter operating under the UNIX system on an AT&T 3B20 computer.

Printed in the United States of America

10 9 8 7

ISBN 0-13-201757-1 025

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*
Prentice-Hall, Inc., *Englewood Cliffs, New Jersey*

To my parents, for their patience and devotion,
to my daughters, Sarah and Rachel, for their laughter,
to my son, Joseph, who arrived after the first printing,
and to my wife, Debby, for her love and understanding.

CONTENTS

PREFACE

The UNIX system was first described in a 1974 paper in the Communications of the ACM [Thompson 74] by Ken Thompson and Dennis Ritchie. Since that time, it has become increasingly widespread and popular throughout the computer industry where more and more vendors are offering support for it on their machines. It is especially popular in universities where it is frequently used for operating systems research and case studies.

Many books and papers have described parts of the system, among them, two special issues of the Bell System Technical Journal in 1978 [BSTJ 78] and 1984 [BLTJ 84]. Many books describe the user level interface, particularly how to use electronic mail, how to prepare documents, or how to use the command interpreter called the shell; some books such as *The UNIX Programming Environment* [Kernighan 84] and *Advanced UNIX Programming* [Rochkind 85] describe the programming interface. This book describes the internal algorithms and structures that form the basis of the operating system (called the kernel) and their relationship to the programmer interface. It is thus applicable to several environments. First, it can be used as a textbook for an operating systems course at either the advanced undergraduate or first-year graduate level. It is most beneficial to reference the system source code when using the book, but the book can be read independently, too. Second, system programmers can use the book as a reference to gain better understanding of how the kernel works and to compare algorithms used in the UNIX system to algorithms used in other operating systems.

Finally, programmers on UNIX systems can gain a deeper understanding of how their programs interact with the system and thereby code more-efficient, sophisticated programs.

The material and organization for the book grew out of a course that I prepared and taught at AT&T Bell Laboratories during 1983 and 1984. While the course centered on reading the source code for the system, I found that understanding the code was easier once the concepts of the algorithms had been mastered. I have attempted to keep the descriptions of algorithms in this book as simple as possible, reflecting in a small way the simplicity and elegance of the system it describes. Thus, the book is not a line-by-line rendition of the system written in English; it is a description of the general flow of the various algorithms, and most important, a description of how they interact with each other. Algorithms are presented in a C-like pseudo-code to aid the reader in understanding the natural language description, and their names correspond to the procedure names in the kernel. Figures depict the relationship between various data structures as the system manipulates them. In later chapters, small C programs illustrate many system concepts as they manifest themselves to users. In the interests of space and clarity, these examples do not usually check for error conditions, something that should always be done when writing programs. I have run them on System V; except for programs that exercise features specific to System V, they should run on other versions of the system, too.

Many exercises originally prepared for the course have been included at the end of each chapter, and they are a key part of the book. Some exercises are straightforward, designed to illustrate concepts brought out in the text. Others are more difficult, designed to help the reader understand the system at a deeper level. Finally, some are exploratory in nature, designed for investigation as a research problem. Difficult exercises are marked with asterisks.

The system description is based on UNIX System V Release 2 supported by AT&T, with some new features from Release 3. This is the system with which I am most familiar, but I have tried to portray interesting contributions of other variations to the operating system, particularly those of Berkeley Software Distribution (BSD). I have avoided issues that assume particular hardware characteristics, trying to cover the kernel-hardware interface in general terms and ignoring particular machine idiosyncrasies. Where machine-specific issues are important to understand implementation of the kernel, however, I delve into the relevant detail. At the very least, examination of these topics will highlight the parts of the operating system that are the most machine dependent.

The reader must have programming experience with a high-level language and, preferably, with an assembly language as a prerequisite for understanding this book. It is recommended that the reader have experience working with the UNIX system and that the reader knows the C language [Kernighan 78]. However, I have attempted to write this book in such a way that the reader should still be able to absorb the material without such background. The appendix contains a simplified description of the system calls, sufficient to understand the presentation in the book, but not a complete reference manual.

The book is organized as follows. Chapter 1 is the introduction, giving a brief, general description of system features as perceived by the user and describing the system structure. Chapter 2 describes the general outline of the kernel architecture and presents some basic concepts. The remainder of the book follows the outline presented by the system architecture, describing the various components in a building block fashion. It can be divided into three parts: the file system, process control, and advanced topics. The file system is presented first, because its concepts are easier than those for process control. Thus, Chapter 3 describes the system buffer cache mechanism that is the foundation of the file system. Chapter 4 describes the data structures and algorithms used internally by the file system. These algorithms use the algorithms explained in Chapter 3 and take care of the internal bookkeeping needed for managing user files. Chapter 5 describes the system calls that provide the user interface to the file system; they use the algorithms in Chapter 4 to access user files.

Chapter 6 turns to the control of processes. It defines the context of a process and investigates the internal kernel primitives that manipulate the process context. In particular, it considers the system call interface, interrupt handling, and the context switch. Chapter 7 presents the system calls that control the process context. Chapter 8 deals with process scheduling, and Chapter 9 covers memory management, including swapping and paging systems.

Chapter 10 outlines general driver interfaces, with specific discussion of disk drivers and terminal drivers. Although devices are logically part of the file system, their discussion is deferred until here because of issues in process control that arise in terminal drivers. This chapter also acts as a bridge to the more advanced topics presented in the rest of the book. Chapter 11 covers interprocess communication and networking, including System V messages, shared memory and semaphores, and BSD sockets. Chapter 12 explains tightly coupled multiprocessor UNIX systems, and Chapter 13 investigates loosely coupled distributed systems.

The material in the first nine chapters could be covered in a one-semester course on operating systems, and the material in the remaining chapters could be covered in advanced seminars with various projects being done in parallel.

A few caveats must be made at this time. No attempt has been made to describe system performance in absolute terms, nor is there any attempt to suggest configuration parameters for a system installation. Such data is likely to vary according to machine type, hardware configuration, system version and implementation, and application mix. Similarly, I have made a conscious effort to avoid predicting future development of UNIX operating system features. Discussion of advanced topics does not imply a commitment by AT&T to provide particular features, nor should it even imply that particular areas are under investigation.

It is my pleasure to acknowledge the assistance of many friends and colleagues who encouraged me while I wrote this book and provided constructive criticism of the manuscript. My deepest appreciation goes to Ian Johnstone, who suggested that I write this book, gave me early encouragement, and reviewed the earliest draft of the first chapters. Ian taught me many tricks of the trade, and I will always be indebted to him. Doris Ryan also had a hand in encouraging me from the very beginning, and I will always appreciate her kindness and thoughtfulness. Dennis Ritchie freely answered numerous questions on the historical and technical background of the system. Many people gave freely of their time and energy to review drafts of the manuscript, and this book owes a lot to their detailed comments. They are Debby Bach, Doug Bayer, Lenny Brandwein, Steve Buroff, Tom Butler, Ron Gomes, Mesut Gunduc, Laura Israel, Dean Jagels, Keith Kelleman, Brian Kernighan, Bob Martin, Bob Mitze, Dave Nowitz, Michael Poppers, Marilyn Safran, Curt Schimmel, Zvi Spitz, Tom Vaden, Bill Weber, Larry Wehr, and Bob Zarrow. Mary Fruhstuck provided help in preparing the manuscript for typesetting. I would like to thank my management for their continued support throughout this project and my colleagues, for providing such a stimulating atmosphere and wonderful work environment at AT&T Bell Laboratories. John Wait and the staff at Prentice-Hall provided much valuable assitance and advice to get the book into its final form. Last, but not least, my wife, Debby, gave me lots of emotional support, without which I could never have succeeded.

1

GENERAL OVERVIEW OF THE SYSTEM

The UNIX system has become quite popular since its inception in 1969, running on machines of varying processing power from microprocessors to mainframes and providing a common execution environment across them. The system is divided into two parts. The first part consists of programs and services that have made the UNIX system environment so popular; it is the part readily apparent to users, including such programs as the shell, mail, text processing packages, and source code control systems. The second part consists of the operating system that supports these programs and services. This book gives a detailed description of the operating system. It concentrates on a description of UNIX System V produced by AT&T but considers interesting features provided by other versions too. It examines the major data structures and algorithms used in the operating system that ultimately provide users with the standard user interface.

This chapter provides an introduction to the UNIX system. It reviews its history and outlines the overall system structure. The next chapter gives a more detailed introduction to the operating system.

1.1 HISTORY

In 1965, Bell Telephone Laboratories joined an effort with the General Electric Company and Project MAC of the Massachusetts Institute of Technology to develop a new operating system called Multics [Organick 72]. The goals of th Multics system were to provide simultaneous computer access to a large communit of users, to supply ample computation power and data storage, and to allow users t share their data easily, if desired. Many people who later took part in the earl development of the UNIX system participated in the Multics work at Bel Laboratories. Although a primitive version of the Multics system was running on GE 645 computer by 1969, it did not provide the general service computing fo which it was intended, nor was it clear when its development goals would be met Consequently, Bell Laboratories ended its participation in the project.

With the end of their work on the Multics project, members of the Computing Science Research Center at Bell Laboratories were left without a "convenient interactive computing service" [Ritchie 84a]. In an attempt to improve their programming environment, Ken Thompson, Dennis Ritchie, and others sketched a paper design of a file system that later evolved into an early version of the UNIX file system. Thompson wrote programs that simulated the behavior of the proposed file system and of programs in a demand-paging environment, and he even encoded a simple kernel for the GE 645 computer. At the same time, he wrote a game program, "Space Travel," in Fortran for a GECOS system (the Honeywell 635), but the program was unsatisfactory because it was difficult to control the "space ship" and the program was expensive to run. Thompson later found a little-used PDP-7 computer that provided good graphic display and cheap executing power. Programming "Space Travel" for the PDP-7 enabled Thompson to learn about the machine, but its environment for program development required cross-assembly of the program on the GECOS machine and carrying paper tape for input to the PDP-7. To create a better development environment, Thompson and Ritchie implemented their system design on the PDP-7, including an early version of the UNIX file system, the process subsystem, and a small set of utility programs. Eventually, the new system no longer needed the GECOS system as a development environment but could support itself. The new system was given the name UNIX, a pun on the name Multics coined by another member of the Computing Science Research Center, Brian Kernighan.

Although this early version of the UNIX system held much promise, it could not realize its potential until it was used in a real project. Thus, while providing a text processing system for the patent department at Bell Laboratories, the UNIX system was moved to a PDP-11 in 1971. The system was characterized by its small size: 16K bytes for the system, 8K bytes for user programs, a disk of 512K bytes, and a limit of 64K bytes per file. After its early success, Thompson set out to implement a Fortran compiler for the new system, but instead came up with the language B, influenced by BCPL [Richards 69]. B was an interpretive language with the performance drawbacks implied by such languages, so Ritchie developed it into one he called C, allowing generation of machine code, declaration of data types, and definition of data structures. In 1973, the operating system was rewritten in C, an unheard of step at the time, but one that was to have tremendous impact on its acceptance among outside users. The number of installations at Bell

Laboratories grew to about 25, and a UNIX Systems Group was formed to provide internal support.

At this time, AT&T could not market computer products because of a 1956 Consent Decree it had signed with the Federal government, but it provided the UNIX system to universities who requested it for educational purposes. AT&T neither advertised, marketed, nor supported the system, in adherence to the terms of the Consent Decree. Nevertheless, the system's popularity steadily increased. In 1974, Thompson and Ritchie published a paper describing the UNIX system in the Communications of the ACM [Thompson 74], giving further impetus to its acceptance. By 1977, the number of UNIX system sites had grown to about 500, of which 125 were in universities. UNIX systems became popular in the operating telephone companies, providing a good environment for program development, network transaction operations services, and real-time services (via MERT [Lycklama 78a]). Licenses of UNIX systems were provided to commercial institutions as well as universities. In 1977, Interactive Systems Corporation became the first Value Added Reseller (VAR)[1] of a UNIX system, enhancing it for use in office automation environments. 1977 also marked the year that the UNIX system was first "ported" to a non-PDP machine (that is, made to run on another machine with few or no changes), the Interdata 8/32.

With the growing popularity of microprocessors, other companies ported the UNIX system to new machines, but its simplicity and clarity tempted many developers to enhance it in their own way, resulting in several variants of the basic system. In the period from 1977 to 1982, Bell Laboratories combined several AT&T variants into a single system, known commercially as UNIX System III. Bell Laboratories later added several features to UNIX System III, calling the new product UNIX System V,[2] and AT&T announced official support for System V in January 1983. However, people at the University of California at Berkeley had developed a variant to the UNIX system, the most recent version of which is called 4.3 BSD for VAX machines, providing some new, interesting features. This book will concentrate on the description of UNIX System V and will occasionally talk about features provided in the BSD system.

By the beginning of 1984, there were about 100,000 UNIX system installations in the world, running on machines with a wide range of computing power from microprocessors to mainframes and on machines across different manufacturers' product lines. No other operating system can make that claim. Several reasons have been suggested for the popularity and success of the UNIX system.

---

1. Value Added Resellers add specific applications to a computer system to satisfy a particular market. They market the applications rather than the operating system upon which they run.
2. What happened to System IV? An internal version of the system evolved into System V.

- The system is written in a high-level language, making it easy to read, understand, change, and move to other machines. Ritchie estimates that the first system in C was 20 to 40 percent larger and slower because it was not written in assembly language, but the advantages of using a higher-level language far outweigh the disadvantages (see page 1965 of [Ritchie 78b]).
- It has a simple user interface that has the power to provide the services that users want.
- It provides primitives that permit complex programs to be built from simpler programs.
- It uses a hierarchical file system that allows easy maintenance and efficient implementation.
- It uses a consistent format for files, the byte stream, making application programs easier to write.
- It provides a simple, consistent interface to peripheral devices.
- It is a multi-user, multiprocess system; each user can execute several processes simultaneously.
- It hides the machine architecture from the user, making it easier to write programs that run on different hardware implementations.

The philosophy of simplicity and consistency underscores the UNIX system and accounts for many of the reasons cited above.

Although the operating system and many of the command programs are written in C, UNIX systems support other languages, including Fortran, Basic, Pascal, Ada, Cobol, Lisp, and Prolog. The UNIX system can support any language that has a compiler or interpreter and a system interface that maps user requests for operating system services to the standard set of requests used on UNIX systems.

1.2 SYSTEM STRUCTURE

Figure 1.1 depicts the high-level architecture of the UNIX system. The hardware at the center of the diagram provides the operating system with basic services that will be described in Section 1.5. The operating system interacts directly[3] with the hardware, providing common services to programs and insulating them from hardware idiosyncrasies. Viewing the system as a set of layers, the operating system is commonly called the *system kernel*, or just the kernel, emphasizing its

3. In some implementations of the UNIX system, the operating system interacts with a native operating system that, in turn, interacts with the underlying hardware and provides necessary services to the system. Such configurations allow installations to run other operating systems and their applications in parallel to the UNIX system. The classic example of such a configuration is the MERT system [Lycklama 78a]. More recent configurations include implementations for IBM System/370 computers [Felton 84] and for UNIVAC 1100 Series computers [Bodenstab 84].

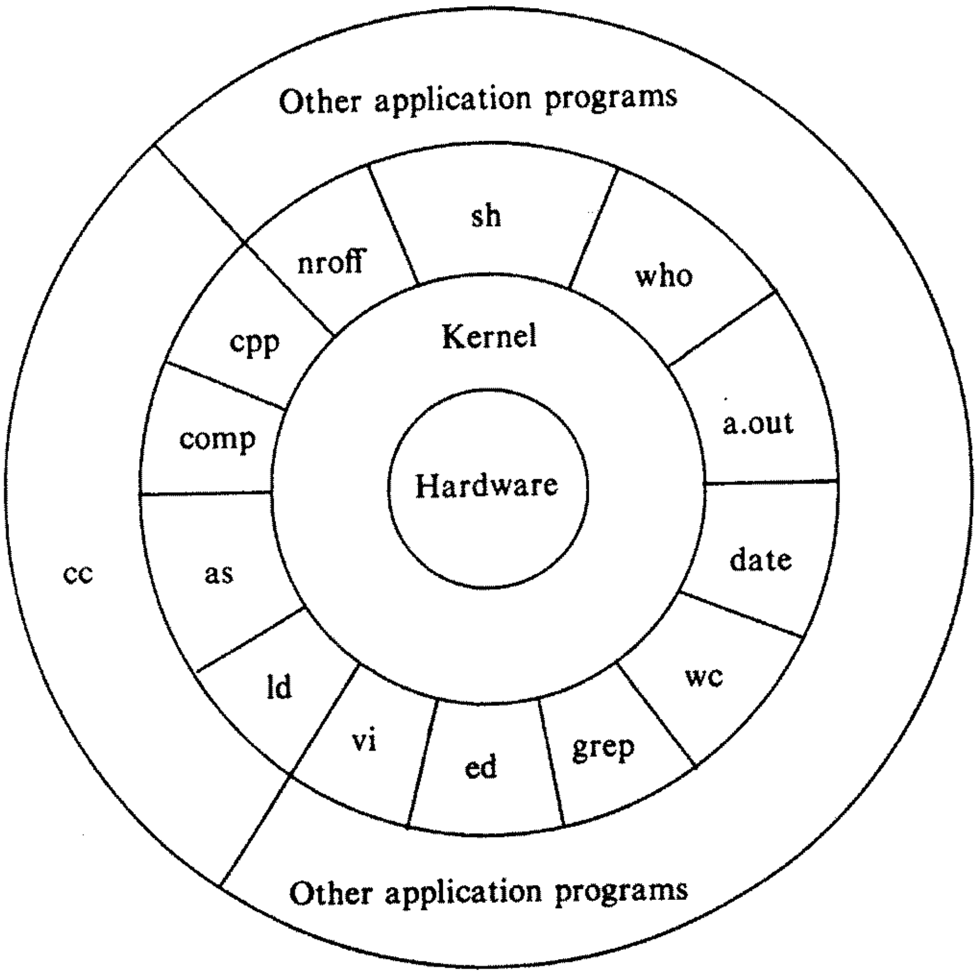

Figure 1.1. Architecture of UNIX Systems

isolation from user programs. Because programs are independent of the underlying hardware, it is easy to move them between UNIX systems running on different hardware if the programs do not make assumptions about the underlying hardware. For instance, programs that assume the size of a machine word are more difficult to move to other machines than programs that do not make this assumption.

Programs such as the shell and editors (*ed* and *vi*) shown in the outer layers interact with the kernel by invoking a well defined set of *system calls*. The system calls instruct the kernel to do various operations for the calling program and exchange data between the kernel and the program. Several programs shown in the figure are in standard system configurations and are known as *commands*, but private user programs may also exist in this layer as indicated by the program whose name is *a.out*, the standard name for executable files produced by the C compiler. Other application programs can build on top of lower-level programs, hence the existence of the outermost layer in the figure. For example, the standard C compiler, *cc*, is in the outermost layer of the figure: it invokes a C preprocessor, two-pass compiler, assembler, and loader (link-editor), all separate lower-l programs. Although the figure depicts a two-level hierarchy of applica programs, users can extend the hierarchy to whatever levels are appropri Indeed, the style of programming favored by the UNIX system encourages combination of existing programs to accomplish a task.

Many application subsystems and programs that provide a high-level view of system such as the shell, editors, SCCS (Source Code Control System), document preparation packages, have gradually become synonymous with the na "UNIX system." However, they all use lower-level services ultimately provided the kernel, and they avail themselves of these services via the set of system ca There are about 64 system calls in System V, of which fewer than 32 are us frequently. They have simple options that make them easy to use but provide t user with a lot of power. The set of system calls and the internal algorithms th implement them form the body of the kernel, and the study of the UNIX operati system presented in this book reduces to a detailed study and analysis of the syste calls and their interaction with one another. In short, the kernel provides t services upon which all application programs in the UNIX system rely, and defines those services. This book will frequently use the terms "UNIX system "kernel," or "system," but the intent is to refer to the kernel of the UNI operating system and should be clear in context.

1.3 USER PERSPECTIVE

This section briefly reviews high-level features of the UNIX system such as the fil system, the processing environment, and building block primitives (for example *pipes*). Later chapters will explore kernel support of these features in detail.

1.3.1 The File System

The UNIX file system is characterized by

- a hierarchical structure,
- consistent treatment of file data,
- the ability to create and delete files,
- dynamic growth of files,
- the protection of file data,
- the treatment of peripheral devices (such as terminals and tape units) as files.

The file system is organized as a tree with a single root node called *root* (written "/"); every non-leaf node of the file system structure is a *directory* of files, and files at the leaf nodes of the tree are either directories, *regular files*, or *special* device files. The name of a file is given by a *path name* that describes how to locate the file in the file system hierarchy. A path name is a sequence of component names separated by slash characters; a component is a sequence of characters that

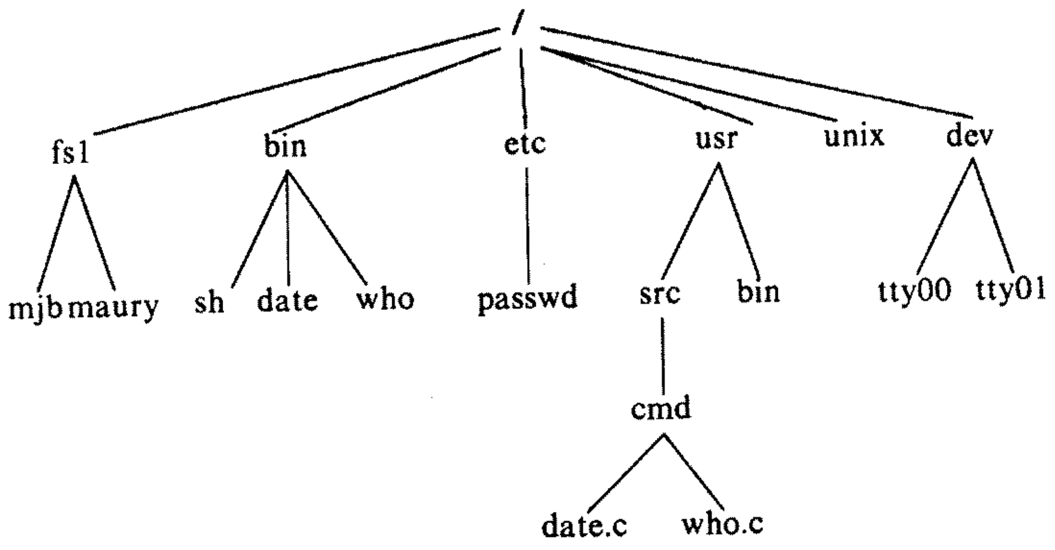

Figure 1.2. Sample File System Tree

designates a file name that is uniquely contained in the previous (directory) component. A full path name starts with a slash character and specifies a file that can be found by starting at the file system root and traversing the file tree, following the branches that lead to successive component names of the path name. Thus, the path names "/etc/passwd", "/bin/who", and "/usr/src/cmd/who.c" designate files in the tree shown in Figure 1.2, but "/bin/passwd" and "/usr/src/date.c" do not. A path name does not have to start from root but can be designated relative to the *current directory* of an executing process, by omitting the initial slash in the path name. Thus, starting from directory "/dev", the path name "tty01" designates the file whose full path name is "/dev/tty01".

Programs in the UNIX system have no knowledge of the internal format in which the kernel stores file data, treating the data as an unformatted stream of bytes. Programs may interpret the byte stream as they wish, but the interpretation has no bearing on how the operating system stores the data. Thus, the syntax of accessing the data in a file is defined by the system and is identical for all programs, but the semantics of the data are imposed by the program. For example, the text formatting program *troff* expects to find "new-line" characters at the end of each line of text, and the system accounting program *acctcom* expects to find fixed length records. Both programs use the same system services to access the data in the file as a byte stream, and internally, they parse the stream into a suitable format. If either program discovers that the format is incorrect, it is responsible for taking the appropriate action.

Directories are like regular files in this respect; the system treats the data in a directory as a byte stream, but the data contains the names of the files in the directory in a predictable format so that the operating system and programs such as *ls* (list the names and attributes of files) can discover the files in a directory.

Permission to access a file is controlled by *access permissions* associated wi the file. Access permissions can be set independently to control read, write, an execute permission for three classes of users: the file owner, a file group, an everyone else. Users may create files if directory access permissions allow it. Th newly created files are leaf nodes of the file system directory structure.

To the user, the UNIX system treats devices as if they were files. Device designated by special device files, occupy node positions in the file system director structure. Programs access devices with the same syntax they use when accessin regular files; the semantics of reading and writing devices are to a large degree th same as reading and writing regular files. Devices are protected in the same wa that regular files are protected: by proper setting of their (file) access permissions Because device names look like the names of regular files and because the sam operations work for devices and regular files, most programs do not have to kno internally the types of files they manipulate.

For example, consider the C program in Figure 1.3, which makes a new copy o an existing file. Suppose the name of the executable version of the program i *copy*. A user at a terminal invokes the program by typing

```
copy oldfile newfile
```

where *oldfile* is the name of the existing file and *newfile* is the name of the new file. The system invokes *main*, supplying *argc* as the number of parameters in the list *argv*, and initializing each member of the array *argv* to point to a user-supplied parameter. In the example above, *argc* is 3, *argv[0]* points to the character string *copy* (the program name is conventionally the 0th parameter), *argv[1]* points to the character string *oldfile*, and *argv[2]* points to the character string *newfile*. The program then checks that it has been invoked with the proper number of parameters. If so, it invokes the *open* system call "read-only" for the file *oldfile*, and if the system call succeeds, invokes the *creat* system call to create *newfile*. The permission modes on the newly created file will be 0666 (octal), allowing all users access to the file for reading and writing. All system calls return −1 on failure; if the *open* or *creat* calls fail, the program prints a message and calls the *exit* system call with return status 1, terminating its execution and indicating that something went wrong.

The *open* and *creat* system calls return an integer called a *file descriptor*, which the program uses for subsequent references to the files. The program then calls the subroutine *copy*, which goes into a loop, invoking the *read* system call to read a buffer's worth of characters from the existing file, and invoking the *write* system call to write the data to the new file. The *read* system call returns the number of bytes read, returning 0 when it reaches the end of file. The program finishes the loop when it encounters the end of file, or when there is some error on the *read* system call (it does not check for *write* errors). Then it returns from *copy* and *exits* with return status 0, indicating that the program completed successfully.

```
#include  <fcntl.h>
char buffer[2048];
int version = 1;        /* Chapter 2 explains this */

main(argc, argv)
        int argc;
        char *argv[];
{
        int fdold, fdnew;

        if (argc != 3)
        {
                printf("need 2 arguments for copy program\n");
                exit(1);
        }
        fdold = open(argv[1], O_RDONLY);   /* open source file read only */
        if (fdold == -1)
        {
                printf("cannot open file %s\n", argv[1]);
                exit(1);
        }
        fdnew = creat(argv[2], 0666);   /* create target file rw for all */
        if (fdnew == -1)
        {
                printf("cannot create file %s\n", argv[2]);
                exit(1);
        }
        copy(fdold, fdnew);
        exit(0);
}

copy(old, new)
        int old, new;
{
        int count;

        while ((count = read(old, buffer, sizeof(buffer))) > 0)
                write(new, buffer, count);
}
```

Figure 1.3. Program to Copy a File

The program copies any files supplied to it as arguments, provided it has permission to *open* the existing file and permission to create the new file. The file can be a file of printable characters, such as the source code for the program, or it can contain unprintable characters, even the program itself. Thus, the two invocations

```
copy copy.c newcopy.c
copy copy newcopy
```

both work. The old file can also be a directory. For instance,

```
copy . dircontents
```

copies the contents of the current directory, denoted by the name ".", to a regular file, "dircontents"; the data in the new file is identical, byte for byte, to the contents of the directory, but the file is a regular file. (The system call *mknod* creates a new directory.) Finally, either file can be a device special file. For example,

```
copy /dev/tty terminalread
```

reads the characters typed at the terminal (the special file */dev/tty* is the user's terminal) and copies them to the file *terminalread*, terminating only when the user types the character control-d. Similarly,

```
copy /dev/tty /dev/tty
```

reads characters typed at the terminal and copies them back.

1.3.2 Processing Environment

A *program* is an executable file, and a *process* is an instance of the program in execution. Many processes can execute simultaneously on UNIX systems (this feature is sometimes called multiprogramming or multitasking) with no logical limit to their number, and many instances of a program (such as *copy*) can exist simultaneously in the system. Various system calls allow processes to create new processes, terminate processes, synchronize stages of process execution, and control reaction to various events. Subject to their use of system calls, processes execute independently of each other.

For example, a process executing the program in Figure 1.4 executes the *fork* system call to create a new process. The new process, called the *child* process, gets a 0 return value from *fork* and invokes *execl* to execute the program *copy* (the program in Figure 1.3). The *execl* call overlays the address space of the child process with the file "copy", assumed to be in the current directory, and runs the program with the user-supplied parameters. If the *execl* call succeeds, it never returns because the process executes in a new address space, as will be seen in Chapter 7. Meanwhile, the process that had invoked *fork* (the parent) receives a non-0 return from the call, calls *wait*, suspending its execution until *copy* finishes, prints the message "copy done," and *exit*s (every program *exit*s at the end of its *main* function, as arranged by standard C program libraries that are linked during the compilation process). For example, if the name of the executable program is *run*, and a user invokes the program by

```
main(argc, argv)
        int argc;
        char *argv[];
{
        /* assume 2 args: source file and target file */
        if (fork() == 0)
                execl("copy", "copy", argv[1], argv[2], 0);
        wait((int *) 0);
        printf("copy done\n");
}
```

Figure 1.4. Program that Creates a New Process to Copy Files

```
run oldfile newfile
```

the process copies "oldfile" to "newfile" and prints out the message. Although this program adds little to the "copy" program, it exhibits four major system calls used for process control: *fork*, *exec*, *wait*, and, discreetly, *exit*.

Generally, the system calls allow users to write programs that do sophisticated operations, and as a result, the kernel of the UNIX system does not contain many functions that are part of the "kernel" in other systems. Such functions, including compilers and editors, are user-level programs in the UNIX system. The prime example of such a program is the *shell*, the command interpreter program that users typically execute after logging into the system. The shell interprets the first word of a *command line* as a *command* name: for many commands, the shell *fork*s and the child process *exec*s the command associated with the name, treating the remaining words on the command line as parameters to the command.

The shell allows three types of commands. First, a command can be an executable file that contains object code produced by compilation of source code (a C program for example). Second, a command can be an executable file that contains a sequence of shell command lines. Finally, a command can be an internal shell command (instead of an executable file). The internal commands make the shell a programming language in addition to a command interpreter and include commands for looping (*for-in-do-done* and *while-do-done*), commands for conditional execution (*if-then-else-fi*), a "case" statement command, a command to change the current directory of a process (*cd*), and several others. The shell syntax allows for pattern matching and parameter processing. Users execute commands without having to know their types.

The shell searches for commands in a given sequence of directories, changeable by user request per invocation of the shell. The shell usually executes a command synchronously, waiting for the command to terminate before reading the next command line. However, it also allows asynchronous execution, where it reads the next command line and executes it without waiting for the prior command to terminate. Commands executed asynchronously are said to execute in the *background*. For example, typing the command

```
who
```

causes the system to execute the program stored in the file */bin/who*,[4] which prints a list of people who are currently logged in to the system. While *who* executes, the shell waits for it to finish and then prompts the user for another command. By typing

```
who &
```

the system executes the program *who* in the background, and the shell is ready to accept another command immediately.

Every process executing in the UNIX system has an execution environment that includes a current directory. The current directory of a process is the start directory used for all path names that do not begin with the slash character. The user may execute the shell command *cd*, change directory, to move around the file system tree and change the current directory. The command line

```
cd /usr/src/uts
```

changes the shell's current directory to the directory "/usr/src/uts". The command line

```
cd ../..
```

changes the shell's current directory to the directory that is two nodes "closer" to the root node: the component ".." refers to the *parent directory* of the current directory.

Because the shell is a user program and not part of the kernel, it is easy to modify it and tailor it to a particular environment. For instance, users can use the C shell to provide a history mechanism and avoid retyping recently used commands, instead of the Bourne shell (named after its inventor, Steve Bourne), provided as part of the standard System V release. Or some users may be granted use only of a restricted shell, providing a scaled down version of the regular shell. The system can execute the various shells simultaneously. Users have the capability to execute many processes simultaneously, and processes can create other processes dynamically and synchronize their execution, if desired. These features provide users with a powerful execution environment. Although much of the power of the shell derives from its capabilities as a programming language and from its capabilities for pattern matching of arguments, this section concentrates on the process environment provided by the system via the shell. Other important shell

4. The directory "/bin" contains many useful commands and is usually included in the sequence of directories the shell searches.