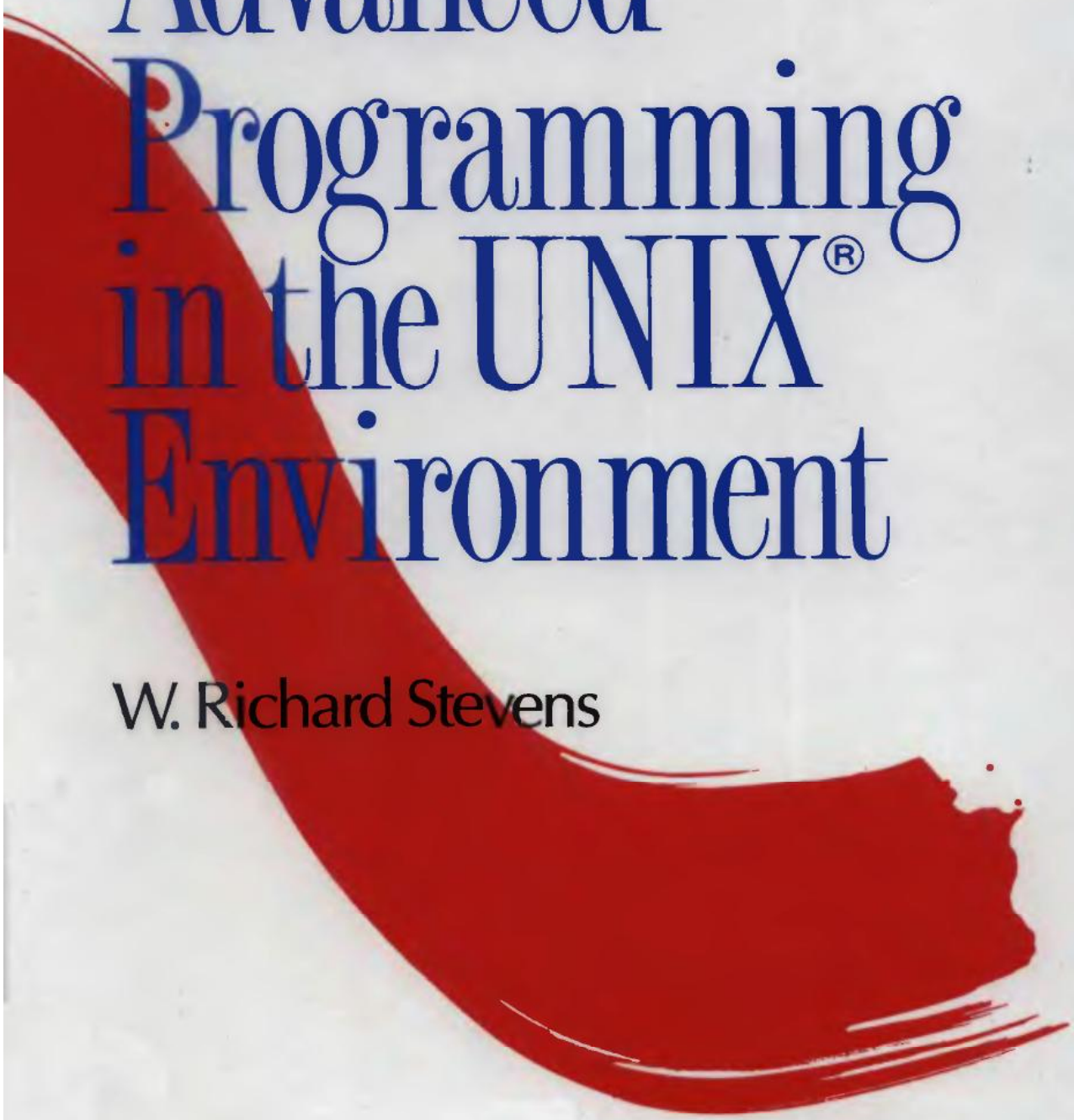
Advanced Programming in the UNIX® Environment

W. Richard Stevens



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

UNIX/C Programming

Advanced Programming in the UNIX® Environment

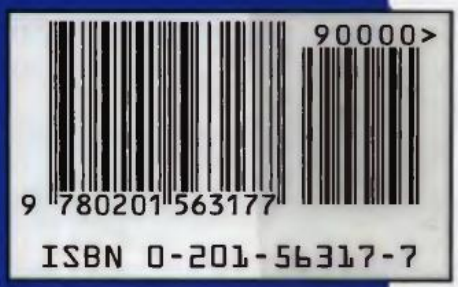
If you are an experienced C programmer with a working knowledge of UNIX, you cannot afford to be without this up-to-date tutorial on the system call interface and the most important functions found in the ANSI C library. Rich Stevens describes more than 200 system calls and functions; since he believes the best way to learn code is to read code, a brief example accompanies each description.

Building upon information presented in the first 15 chapters, the author offers chapter-long examples teaching you how to create a database library, a PostScript printer driver, a modem dialer, and a program that runs other programs under a pseudo terminal. To make your analysis and understanding of this code even easier, and to allow you to modify it, all of the code in the book is available via UUNET.

A 20-page appendix provides detailed function prototypes for all the UNIX, POSIX, and ANSI C functions that are described in the book, and lists the page on which each prototype function is described in detail. Additional tables throughout the text and a thorough index make *Advanced Programming in the UNIX Environment* an invaluable reference tool that all UNIX programmers—beginners to experts—will want on their bookshelves.

Advanced Programming in the UNIX Environment is applicable to all major UNIX releases, especially System V Release 4 and the latest release of 4.3BSD, including 386BSD. These real-world implementations allow you to more clearly understand the status of the current and future standards, including IEEE POSIX and XPG3.

W. Richard Stevens, author of the popular text, *UNIX Network Programming*, is also author of the *UNIX System Workshop* and *Advanced UNIX Programming* courses from Technology Exchange Company.



Cover design by Joyce C. Weston
Text printed on recycled paper

Corporate & Professional Publishing Group
Addison-Wesley Publishing Company, Inc.

Advanced Programming in the UNIX® Environment

Addison-Wesley Professional Computing Series

Brian W. Kernighan, Consulting Editor

Ken Arnold/John Peyton, *A C User's Guide to ANSI C*
Tom Cargill, *C++ Programming Style*
David Curry, *UNIX System Security: A Guide for Users and System Administrators*
Scott Meyers, *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*
Radia Perlman, *Interconnections: Bridges and Routers*
W. Richard Stevens, *Advanced Programming in the UNIX Environment*

Advanced Programming in the UNIX® Environment

W. Richard Stevens

ADDISON-WESLEY PUBLISHING COMPANY, INC.
Reading, Massachusetts Menlo Park, California New York Don Mills, Ontario
Wokingham, England Amsterdam Bonn Paris Milan Madrid Sydney Singapore Tokyo
Seoul Taipei Mexico City San Juan

The publisher offers discounts on this book when ordered in quantity for special sales.
For more information please contact:

Corporate & Professional Publishing Group
Addison-Wesley Publishing Company
One Jacob Way
Reading, Massachusetts 01867

ISBN 0-201-56317-7
2 3 4 5 6 7 8 9 10-MU-95949392
Second printing September 1992

To MTS, the Michigan Terminal System,
and the 360/67.

Contents

Preface

Introduction

This book describes the programming interface to the Unix system—the system call interface and many of the functions provided in the standard C library. It is intended for anyone writing programs that run under Unix.

Like most operating systems, Unix provides numerous services to the programs that are running—open a file, read a file, start a new program, allocate a region of memory, get the current time-of-day, and so on. This has been termed the *system call interface*. Additionally, the standard C library provides numerous functions that are used by almost every C program (format a variable's value for output, compare two strings, etc.).

The system call interface and the library routines have traditionally been described in Sections 2 and 3 of the *Unix Programmer's Manual*. This book is not a duplication of these sections. Examples and rationale are missing from the *Unix Programmer's Manual*, and that's what this book provides.

Unix Standards

The proliferation of different versions of Unix during the 1980s has been tempered by the various international standards that were started during the late 1980s. These include the ANSI standard for the C programming language, the IEEE POSIX family (still being developed), and the X/Open portability guide.

This book also describes these standards. But instead of just describing the standards by themselves, we describe them in relation to popular implementations of the standards—System V Release 4 and the forthcoming 4.4BSD. This provides a real-world description, which is often lacking from the standard itself and from books that describe only the standard.

Organization of the Book

This book is divided into six parts:

1. An overview and introduction to basic Unix programming concepts and terminology (Chapter 1), with a discussion of the various Unix standardization efforts and different Unix implementations (Chapter 2).
2. I/O—unbuffered I/O (Chapter 3), properties of files and directories (Chapter 4), the standard I/O library (Chapter 5), and the standard system data files (Chapter 6).
3. Processes—the environment of a Unix process (Chapter 7), process control (Chapter 8), the relationships between different processes (Chapter 9), and signals (Chapter 10).
4. More I/O—terminal I/O (Chapter 11), advanced I/O (Chapter 12), and daemon processes (Chapter 13).
5. IPC—Interprocess communication (Chapters 14 and 15).
6. Examples—a database library (Chapter 16), communicating with a PostScript printer (Chapter 17), a modem dialing program (Chapter 18), and using pseudo terminals (Chapter 19).

A reading familiarity with C would be beneficial as would some experience using Unix. No prior programming experience with Unix is assumed. This text is intended for programmers familiar with Unix and programmers familiar with some other operating system who wish to learn the details of the services provided by most Unix systems.

Examples in the Text

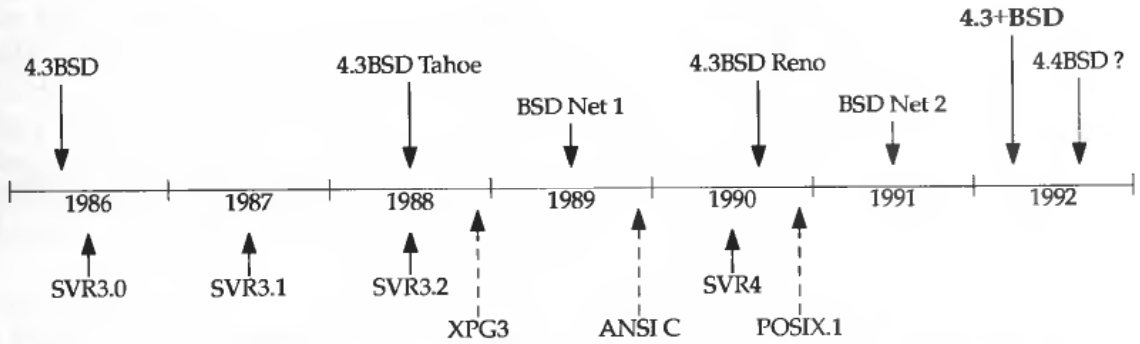
This book contains many examples—approximately 10,000 lines of source code. All the examples are in the C programming language. Furthermore, these examples are in ANSI C. You should have a copy of the *Unix Programmer's Manual* for your system handy while reading this book, since reference is made to it for some of the more esoteric and implementation-dependent features.

Almost every function and system call is demonstrated with a small, complete program. This lets us see the arguments and return values and is often easier to comprehend than the use of the function in a much larger program. But since some of the small programs are contrived examples, a few bigger examples are also included (Chapters 16, 17, 18, and 19). These larger examples demonstrate the programming techniques in larger, real-world examples.

All the examples have been included in the text directly from their source files. A machine-readable copy of all the examples is available via anonymous FTP from the Internet host `ftp.uu.net` in the file `published/books/stevens.advprog.tar.Z`. Obtaining the source code allows you to modify the programs from this text and experiment with them on your system.

Systems Used to Test the Examples

Unfortunately all operating systems are moving targets. Unix is no exception. The following diagram shows the recent evolution of the various versions of System V and 4.xBSD.



4.xBSD are the various systems from the Computer Systems Research Group at the University of California at Berkeley. This group also distributes the BSD Net 1 and BSD Net 2 releases—publicly available source code from the 4.xBSD systems. SVRx refers to System V Release x from AT&T. XPG3 is the X/Open Portability Guide, Issue 3, and ANSI C is the ANSI standard for the C programming language. POSIX.1 is the IEEE and ISO standard for the interface to a Unix-like system. We'll have more to say about these different standards and the various versions of Unix in Sections 2.2 and 2.3.

> **In this text we use the term *4.3+BSD* to refer to the Unix system from Berkeley that is somewhere between the BSD Net 2 release and 4.4BSD.**
>
> At the time of this writing, 4.4BSD was not released, so the system could not be called 4.4BSD. Nevertheless a simple name was needed to refer to this system and *4.3+BSD* is used throughout the text.

Most of the examples in this text have been run on four different versions of Unix:

1. Unix System V/386 Release 4.0 Version 2.0 ("vanilla SVR4") from U.H. Corp. (UHC), on an Intel 80386 processor.
2. 4.3+BSD at the Computer Systems Research Group, Computer Science Division, University of California at Berkeley, on a Hewlett Packard workstation.
3. BSD/386 (a derivative of the BSD Net 2 release) from Berkeley Software Design, Inc., on an Intel 80386 processor. This system is almost identical to what we call 4.3+BSD.
4. SunOS 4.1.1 and 4.1.2 (systems with a strong Berkeley heritage but many System V features) from Sun Microsystems, on a SPARCstation SLC.

Numerous timing tests are provided in the text and the systems used for the test are identified.

Acknowledgments

Once again I am indebted to my family for their love, support, and many lost weekends over the past year and a half. Writing a book is, in many ways, a family affair. Thank you Sally, Bill, Ellen, and David.

I am especially grateful to Brian Kernighan for his help in the book. His numerous thorough reviews of the entire manuscript and his gentle prodding for better prose hopefully show in the final result. Steve Rago was also a great resource, both in reviewing the entire manuscript and answering many questions about the details and history of System V. My thanks to the other technical reviewers used by Addison-Wesley, who provided valuable comments on various portions of the manuscript: Maury Bach, Mark Ellis, Jeff Gitlin, Peter Honeyman, John Linderman, Doug McIlroy, Evi Nemeth, Craig Partridge, Dave Presotto, Gary Wilson, and Gary Wright.

Keith Bostic and Kirk McKusick at the U.C. Berkeley CSRG provided an account that was used to test the examples on the latest BSD system. (Many thanks to Peter Salus too.) Sam Nataros and Joachim Sacksen at UHC provided the copy of SVR4 used to test the examples. Trent Hein helped obtain the alpha and beta copies of BSD/386.

Other friends have helped in many small, but significant ways over the past few years: Paul Lucchina, Joe Godsil, Jim Hogue, Ed Tankus, and Gary Wright. My editor at Addison-Wesley, John Wait, has been a great friend through it all. He never complained when the due date slipped and the page count kept increasing. A special thanks to the National Optical Astronomy Observatories (NOAO), especially Sidney Wolff, Richard Wolff, and Steve Grandi, for providing computer time.

Real Unix books are written using troff and this book follows that time-honored tradition. Camera-ready copy of the book was produced by the author using the groff package written by James Clark. Many thanks to James Clark for providing this excellent system and for his rapid response to bug fixes. Perhaps someday I will really understand troff footer traps.

I welcome electronic mail from any readers with comments, suggestions, or bug fixes: `rstevens@noao.edu`.

Tucson, Arizona
April 1992

W. Richard Stevens

1

Introduction

1.1 Introduction

All operating systems provide services for programs they run. Typical services are execute a new program, open a file, read a file, allocate a region of memory, get the current time-of-day, and so on. The focus of this text is to describe the services provided by various versions of the Unix operating system.

Describing Unix in a strictly stepwise fashion, without any forward references to terms that haven't been described yet, is nearly impossible (and would probably be boring). This chapter is a whirlwind tour of Unix from a programmer's perspective. We'll give some brief descriptions and examples of terms and concepts that will be encountered throughout the text. We describe these features in much more detail in later chapters. This chapter also provides an introduction and overview of the services provided by Unix, for programmers new to Unix.

1.2 Logging In

Login Name

When we log in to a Unix system we enter our login name, followed by our password. Our login name is then looked up in the system's password file, usually the file `/etc/passwd`. If we look at our entry in the password file we see that it's composed of seven colon-separated fields: our login name, encrypted password, numeric user ID (224), numeric group ID (20), a comment field, home directory (`/home/stevens`), and shell program (`/bin/ksh`).

Many newer systems have moved the encrypted password to a different file. In Chapter 6 we'll look at these files and some functions to access them.

Shells

Once we log in, some system information messages are typically displayed, and then we are able to enter commands to the shell program. A *shell* is a command line interpreter that reads user input and executes commands. The user input to a shell is normally from the terminal (an interactive shell) or sometimes from a file (called a *shell script*). The common shells in use are

- the Bourne shell, `/bin/sh`
- the C shell, `/bin/csh`
- the KornShell, `/bin/ksh`

The system knows which shell to execute for us from the final field in our entry in the password file.

The Bourne shell has been in use since Version 7 and is provided with almost every Unix system in existence. The C shell was developed at Berkeley and is provided with all the BSD releases. Additionally the C shell was provided by AT&T with System V/386 Release 3.2 and is also in System V Release 4 (SVR4). (We'll have more to say about these different versions of Unix in the next chapter.) The KornShell is considered to be a successor to the Bourne shell and is provided in SVR4. The KornShell runs on most Unix systems, but before SVR4 it was usually an extra cost add-on, so it is not as widespread as the other two shells.

> The Bourne shell was developed by Steve Bourne at Bell Labs. Its control flow constructs are reminiscent of Algol 68. The C shell was done at Berkeley by Bill Joy. It was built on the 6th Edition shell (not the Bourne shell). Its control flow looks more like the C language, and it supports additional features that weren't provided by the Bourne shell—job control, a history mechanism, and command-line editing. We return to Bell Labs with the KornShell, where it was developed by David Korn. It is upward-compatible from the Bourne shell and includes those features that made the C shell popular—job control, command line editing, etc.
>
> Throughout the text we will use parenthetical notes such as this to describe historical notes and comparisons between different Unix implementations. Often the reason for a particular implementation technique becomes clear when the historical reasons are described.

Throughout this text we'll show shell examples to execute a program that we've developed. This interactive use will use features common to both the Bourne shell and the KornShell.

1.3 Files and Directories

Filesystem

The Unix filesystem is a hierarchical arrangement of directories and files. Everything starts in the directory called *root* whose name is the single character `/`.

A *directory* is a file that contains directory entries. Logically we can think of each directory entry as containing a filename along with a structure of information describing the attributes of the file. The attributes of a file are things such as: type of file, size of the file, owner of the file, permissions for the file (e.g., can other users access this file?), time of last modification of the file, and the like. The `stat` and `fstat` functions return a structure of information containing all the attributes of a file. In Chapter 4 we'll examine all the attributes of a file in great detail.

Filename

The names in a directory are called *filenames*. The only two characters that cannot appear in a filename are the slash character (`/`) and the null character. The slash separates the filenames that form a pathname (described next) and the null character terminates a pathname. Nevertheless, it's good practice to restrict the characters in a filename to a subset of the normal printing characters. (The reason we restrict it to a subset is because if we use some of the shell's special characters in the filename, we have to use the shell's quoting mechanism to reference the filename.)

Two filenames are automatically created whenever a new directory is created: `.` (called *dot*) and `..` (called *dot-dot*). Dot refers to the current directory and dot-dot refers to the parent directory. In the ultimate parent directory, the root, dot-dot is the same as dot.

Some Unix filesystems restrict a filename to 14 characters. BSD versions extended this limit to 255 characters.

Pathname

A sequence of zero or more filenames, separated by slashes, and optionally starting with a slash, forms a *pathname*. A pathname that begins with a slash is called an *absolute pathname*, otherwise it's called a *relative pathname*.

Example

Listing the names of all the files in a directory is not hard. Program 1.1 is a bare bones implementation of the `ls`(1) command.

```
#include    <sys/types.h>
#include    <dirent.h>
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    DIR             *dp;
    struct dirent   *dirp;

    if (argc != 2)
        err_quit("a single argument (the directory name) is required");

    if ( (dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);

    while ( (dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

Program 1.1 List all the files in a directory.

The notation `ls`(1) is the normal way to reference a particular entry in the Unix manual set. It refers to the entry for `ls` in Section 1. The sections are normally numbered 1 through 8, and all the entries within each section are arranged alphabetically. We assume throughout this text that you have a copy of the Unix manuals for your system.

> Older Unix systems lumped all eight sections together into what was called the *Unix Programmer's Manual*. The trend today is to distribute the sections among separate manuals: one for the users, one for the programmers, and one for the system administrators, for example.
>
> Some Unix systems further divide the manual pages within a given section using an uppercase letter. For example, all the standard I/O functions in AT&T [1990e] are indicated as being in Section 3S, as in `fopen`(3S).
>
> Some Unix systems, notably Xenix-based systems, don't number the manual sections numerically. Instead they use the notation C for commands (Section 1), S for services (normally Sections 2 and 3), and so on.

If your manuals are on-line, the way to see the manual pages for the `ls` command would be something like

```
man 1 ls
```

Program 1.1 just prints the name of every file in a directory, and nothing else. If the source file is named `myls.c`, we compile it into the default `a.out` executable file by

```
cc myls.c
```

Some sample output is

```
$ a.out /dev
.
..
MAKEDEV
console
tty
mem
kmem
null
                    many more lines that aren't shown
printer
$ a.out /var/spool/mqueue
can't open /var/spool/mqueue: Permission denied
$ a.out /dev/tty
can't open /dev/tty: Not a directory
```

Throughout this text we'll show commands that we enter and the resulting output in this fashion: characters that we enter are shown in **`this font`** while output from programs is shown `like this`. If we need to add comments to this output we'll show the comments in *italics*. The dollar sign that precedes our input is the prompt that is printed by the shell. We'll always show the shell prompt as a dollar sign.

Note that the directory listing is not in alphabetical order. The ordering that we are familiar with is done by the `ls` command itself.

There are many details to consider in this 20-line program:

- First, we include a header of our own, `ourhdr.h`. We include this header in almost every program in this text. It includes some standard system headers and defines numerous constants and function prototypes that we use throughout the examples in the text. A listing of this header is in Appendix B.
- The declaration of the `main` function uses the new style supported by the ANSI C standard. (We'll have more to say about the ANSI C standard in the next chapter.)
- We take an argument from the command line, `argv[1]`, as the name of the directory to list. In Chapter 7 we'll look at how the `main` function is called, and how the command-line arguments and environment variables are accessible to the program.
- Since the actual format of directory entries varies from one Unix system to another, we use the functions `opendir`, `readdir`, and `closedir` to manipulate the directory.
- The `opendir` function returns a pointer to a `DIR` structure, and we pass this pointer to the `readdir` function. We don't care what's in the `DIR` structure. We then call `readdir` in a loop, to read each directory entry. It returns a pointer to a `dirent` structure, or a null pointer when it's finished with the directory. All we examine in

the `dirent` structure is the name of each directory entry (`d_name`). Using this name we could then call the `stat` function (Section 4.2) to determine all the attributes of the file.

- We call two functions of our own to handle the errors: `err_sys` and `err_quit`. We can see from the output above that the `err_sys` function prints an informative message describing what type of error was encountered ("Permission denied" or "Not a directory"). These two error functions are shown and described in Appendix B. We also talk more about error handling in Section 1.7.
- When the program is done it calls the function `exit` with an argument of 0. The function `exit` terminates a program. By convention an argument of 0 means OK, and an argument between 1 and 255 means an error occurred. In Section 8.5 we show how any program (such as a shell or a program that we write) can obtain the `exit` status of a program that it executes. □

Working Directory

Every process has a *working directory* (sometimes called the *current working directory*). This is the directory from which all relative pathnames are interpreted. A process can change its working directory with the `chdir` function.

For example, the relative pathname `doc/memo/joe` refers to the file (or directory) `joe`, in the directory `memo`, in the directory `doc`, which must be a directory within the working directory. From looking just at this pathname we know that both `doc` and `memo` have to be directories, but we can't tell if `joe` is a file or directory. The pathname `/usr/lib/lint` is an absolute pathname that refers to the file (or directory) `lint` in the directory `lib`, in the directory `usr`, which is in the root directory.

Home Directory

When we log in, the working directory is set to our *home directory*. Our home directory is obtained from our entry in the password file (recall Section 1.2).

1.4 Input and Output

File Descriptors

File descriptors are small nonnegative integers that the kernel uses to identify the files being accessed by a particular process. Whenever the kernel opens an existing file, or creates a new file, it returns a file descriptor that we use when we want to read or write the file.

Standard Input, Standard Output, and Standard Error

By convention, all shells open three descriptors whenever a new program is run: the standard input, standard output, and standard error. If nothing special is done, as in the simple command

```
ls
```

then all three are connected to our terminal. Most shells provide a way to redirect any or all of these three descriptors to any file. For example,

```
ls > file.list
```

executes the `ls` command with its standard output redirected to the file named `file.list`.

Unbuffered I/O

Unbuffered I/O is provided by the functions `open`, `read`, `write`, `lseek`, and `close`. These functions all work with file descriptors.

Example

If we're willing to read from the standard input and write to the standard output, then Program 1.2 copies any Unix file.

```
#include    "ourhdr.h"

#define BUFFSIZE    8192

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    while ( (n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

Program 1.2 Copy standard input to standard output.

The <unistd.h> header (that's included by ourhdr.h) and the two constants STDIN_FILENO and STDOUT_FILENO are part of the POSIX standard (about which we'll have a lot more to say in the next chapter). In this header are function prototypes for many of the Unix system services, such as the read and write functions that we call. Function prototypes are part of the ANSI C standard, and we talk more about them later in this chapter.

The two constants STDIN_FILENO and STDOUT_FILENO are defined in the <unistd.h> header, and specify the file descriptors for standard input and standard output. These values are typically 0 and 1, respectively, but we'll use the new names for portability.

In Section 3.9 we'll examine the BUFFSIZE constant in detail, seeing how various values affect the efficiency of the program. Regardless of the value of this constant, however, this program still copies any Unix file.

The read function returns the number of bytes that are read, and this value is used as the number of bytes to write. When the end of the input file is encountered, read returns 0 and the program stops. If a read error occurs, read returns –1. Most of the system functions return –1 when an error occurs.

If we compile the program into the standard a.out file and execute it as

```
a.out > data
```

standard input is the terminal, standard output is redirected to the file data, and standard error is also the terminal. If this output file doesn't exist, the shell creates it by default. □

In Chapter 3 we describe the unbuffered I/O functions in more detail.

Standard I/O

The standard I/O functions provide a buffered interface to the unbuffered I/O functions. Using standard I/O prevents us from having to worry about choosing optimal buffer sizes, such as the BUFFSIZE constant in Program 1.2. Another advantage of using the standard I/O functions is when we're dealing with lines of input (a common occurrence in Unix applications). The fgets function, for example, reads an entire line. The read function, on the other hand, reads a specified number of bytes.

The standard I/O function that we're most familiar with is printf. In the programs that call printf, we'll always include <stdio.h> (normally by including ourhdr.h), since this header contains the function prototypes for all the standard I/O functions.

Example

Program 1.3, which we'll examine in more detail in Section 5.8, is like the previous program that called read and write. It copies standard input to standard output and can copy any Unix file.

```
#include    "ourhdr.h"

int
main(void)
{
    int     c;

    while ( (c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

Program 1.3 Copy standard input to standard output using standard I/O.

The function `getc` reads one character at a time, and this character is written by `putc`. After the last byte of input has been read, `getc` returns the constant `EOF`. The standard I/O constants `stdin` and `stdout` are defined in the `<stdio.h>` header and refer to the standard input and standard output. □

1.5 Programs and Processes

Program

A *program* is an executable file residing in a disk file. A program is read into memory and executed by the kernel as a result of one of the six `exec` functions. We'll cover these functions in Section 8.9.

Processes and Process ID

An executing instance of a program is called a *process*. We'll encounter this term on almost every page of the text. Some operating systems use the term task to refer to a program that is being executed.

Every Unix process is guaranteed to have a unique numeric identifier called the *process ID*. The process ID is always a nonnegative integer.

Example

Program 1.4 prints its process ID.

```
#include    "ourhdr.h"

int
main(void)
{
    printf("hello world from process ID %d\n", getpid());
    exit(0);
}
```

Program 1.4 Print the process ID.

If we compile this program into the file `a.out` and execute it, we have

```
$ a.out
hello world from process ID 851
$ a.out
hello world from process ID 854
```

When this program runs it calls the function `getpid` to obtain its process ID. □

Process Control

There are three primary functions used for process control: `fork`, `exec`, and `waitpid`. (There are six variants of the `exec` function, but we often refer to them collectively as just the `exec` function.)

Example

The process control features of Unix can be demonstrated using a simple program (Program 1.5) that reads commands from standard input and executes the commands. This is a bare bones implementation of a shell-like program. There are several features to consider in this 30-line program.

- We use the standard I/O function `fgets` to read one line at a time from the standard input. When we type the end of file character (often Control-D) as the first character of a line, `fgets` returns a null pointer, the loop stops, and the process terminates. In Chapter 11 we describe all the special terminal characters (end of file, backspace one character, erase entire line, etc.) and how to change them.
- Since each line returned by `fgets` is terminated with a newline character, followed by a null byte, we use the standard C function `strlen` to calculate the length of the string, and then replace the newline with a null byte. We do this because the `execlp` function wants a null terminated argument, not a newline terminated argument.
- We call `fork` to create a new process. The new process is a copy of the caller, and we say the caller is the parent and the newly created process is the child. Then `fork` returns the nonnegative process ID of the new child process to the

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

int
main(void)
{
    char    buf[MAXLINE];
    pid_t   pid;
    int     status;

    printf("%% ");  /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        buf[strlen(buf) - 1] = 0;   /* replace newline with null */

        if ( (pid = fork()) < 0)
            err_sys("fork error");

        else if (pid == 0) {        /* child */
            execlp(buf, buf, (char *) 0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ( (pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

Program 1.5 Read commands from standard input and execute them.

parent, and it returns 0 to the child. Since `fork` creates a new process, we say that it is called once (by the parent) but returns twice (in the parent and in the child).

- In the child we call `execlp` to execute the command that was read from the standard input. This replaces the child process with the new program file. The combination of a `fork`, followed by an `exec`, is what some operating systems call spawning a new process. In Unix the two parts are separated into individual functions. We'll have a lot more to say about these functions in Chapter 8.
- Since the child calls `execlp` to execute the new program file, the parent wants to wait for the child to terminate. This is done by calling `waitpid`, specifying which process we want to wait for (the `pid` argument, which is the process ID of the child). The `waitpid` function also returns the termination status of the child (the `status` variable), but in this simple program we don't do anything with this value. We could examine it to determine exactly how the child terminated.

- The most fundamental limitation of this program is that we can't pass arguments to the command that we execute. We can't, for example, specify the name of a directory to list. We can only execute `ls` on the working directory. To allow arguments would require that we parse the input line, separating the arguments by some convention (probably spaces or tabs) and then pass each argument as a separate argument to the `execlp` function. Nevertheless, this program is still a useful demonstration of the process control functions of Unix.

If we run this program we get the following results. Notice that our program has a different prompt (the percent sign).

```
$ a.out
% date
Fri Jun  7 15:50:36 MST 1991
% who
stevens  console Jun  5 06:01
stevens  ttyp0   Jun  5 06:02
% pwd
/home/stevens/doc/apue/proc
% ls
Makefile
a.out
shell1.c
% ^D                          type our end-of-file character
$                             the regular shell prompt is output
```

□

1.6 ANSI C Features

All the examples in this text are written in the version of the C programming language that is called ANSI C.

Function Prototypes

The header `<unistd.h>` includes function prototypes for many of the Unix system services, such as the `read`, `write`, and `getpid` functions that we've called. Function prototypes are part of the ANSI C standard. These function prototypes probably look like

```
ssize_t  read(int, void *, size_t);
ssize_t  write(int, void *, size_t);
pid_t    getpid(void);
```

The final one says that `getpid` takes no arguments (`void`) and returns a value that has the data type `pid_t`. By providing these function prototypes we are able to let the compiler do additional checking at compile time, to verify that we are calling functions with the correct arguments. In Program 1.4, if we had called `getpid` with an argument, as in `getpid(1)`, we would get an error message of the form

```
line 8: too many arguments to function "getpid"
```

from an ANSI C compiler. Also, since the compiler knows the data types of the arguments, it is able to cast the arguments to their required data types, if possible.

Generic Pointers

Another difference that we'll note in the function prototypes shown previously is that the second argument for `read` and `write` is now of type `void *`. All earlier Unix systems used `char *` for this pointer. This change is because ANSI C uses `void *` as the generic pointer, instead of `char *`.

Combining function prototypes and generic pointers lets us remove many of the explicit type casts that are needed with non-ANSI C compilers. For example, given the prototype for `write` earlier, we can write

```
float  data[100];

write(fd, data, sizeof(data));
```

With a non-ANSI compiler, or without the function prototype, we need to write

```
write(fd, (void *) data, sizeof(data));
```

We'll also use this feature of `void *` pointers with the `malloc` function (Section 7.8). The prototype for `malloc` is now

```
void *malloc(size_t);
```

This lets us write

```
int  *ptr;

ptr = malloc(1000 * sizeof(int));
```

without explicitly casting the returned pointer to an `int *`.

Primitive System Data Types

The prototype for the `getpid` function shown earlier defines its return value as being of type `pid_t`. This is also new with POSIX. Earlier versions of Unix defined this function as returning an integer. Similarly both `read` and `write` return a value of type `ssize_t` and require a third argument of type `size_t`.

These data types that end in `_t` are called the primitive system data types. They are usually defined in the header `<sys/types.h>` (which the header `<unistd.h>` must have included). They are usually defined with the C `typedef` declaration, which has been in C for over 15 years (so it doesn't require ANSI C). Their purpose is to prevent programs from using specific data types (such as `int`, `short`, or `long`, for example) to allow each implementation to choose which data type is required for a particular system. Everywhere we need to store a process ID, we'll allocate a variable of type `pid_t`.

(Notice that we did this for the variable named `pid` in Program 1.5.) While the definition of this data type might differ from one implementation to another, the differences are restricted to one header. All we have to do is recompile the application on another system.

1.7 Error Handling

When an error occurs in one of the Unix functions, a negative value is often returned, and the integer `errno` is usually set to a value that gives additional information. For example, the `open` function returns either a nonnegative file descriptor if all is OK, or –1 if an error occurs. In the case of an error from `open`, there are about 15 different `errno` values (file doesn't exist, permission problem, etc.). Some functions use a convention other than returning a negative value. For example, most functions that return a pointer to an object return a null pointer to indicate an error.

The file `<errno.h>` defines the variable `errno` and constants for each value that `errno` can assume. Each of these constants begins with the character `E`. Also, the first page of Section 2 of the Unix manuals, named `intro(2)`, usually lists all these error constants. For example, if `errno` is equal to the constant `EACCES`, this indicates a permission problem (we don't have permission to open the requested file, for example). POSIX defines `errno` as

```
extern int errno;
```

> This POSIX.1 definition of `errno` is stricter than the definition in the C standard. The C standard allows `errno` to be a macro that expands into a modifiable lvalue of type integer (such as a function that returns a pointer to the error number).

There are two rules to be aware of with respect to `errno`. First, its value is never cleared by a routine if an error does not occur. Therefore, we should examine its value only when the return value from a function indicates that an error occurred. Second, the value of `errno` is never set to 0 by any of the functions, and none of the constants defined in `<errno.h>` have a value of 0.

Two functions are defined by the C standard to help with the printing of error messages.

```
#include <string.h>

char *strerror(int errnum);
```

Returns: pointer to message string

This function maps *errnum* (which is typically the `errno` value) into an error message string and returns a pointer to the string.

The `perror` function produces an error message on the standard error (based on the current value of `errno`) and returns.

```
#include <stdio.h>

void perror(const char *msg);
```

It first outputs the string pointed to by *msg*, followed by a colon and a space, followed by the error message corresponding to the value of `errno`, followed by a newline.

Example

Program 1.6 shows the use of these two error functions.

```
#include    <errno.h>
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    fprintf(stderr, "EACCES: %s\n", strerror(EACCES));

    errno = ENOENT;
    perror(argv[0]);

    exit(0);
}
```

Program 1.6 Demonstrate `strerror` and `perror`.

If this program is compiled into the file `a.out`, we have

```
$ a.out
EACCES: Permission denied
a.out: No such file or directory
```

Note that we pass the name of the program (`argv[0]`, whose value is `a.out`) as the argument to `perror`. This is a standard Unix convention. By doing this, if the program is executed as part of a pipeline, as in

```
prog1 < inputfile | prog2 | prog3 > outputfile
```

we are able to tell which of the three programs generated a particular error message. □

Instead of calling either `strerror` or `perror` directly, all the examples in this text use the error functions shown in Appendix B. The error functions in this appendix let us use the variable argument list facility of ANSI C to handle error conditions with a single C statement.

1.8 User Identification

User ID

The *user ID* from our entry in the password file is a numeric value that identifies us to the system. This user ID is assigned by the system administrator when our login name is assigned and we cannot change it. It is normally assigned so that every user has a unique user ID. We'll see how the user ID is utilized by the kernel to check if we have the appropriate permissions to perform certain operations.

We call the user whose user ID is 0 either *root* or the *superuser*. The entry in the password file normally has a login name of `root` and we refer to the special privileges of this user as superuser privileges. As we'll see in Chapter 4, if a process has superuser privileges, most file permission checks are bypassed. Some operating system functions are restricted to only the superuser. The superuser has free reign over the system.

Example

Program 1.7 prints the user ID and the group ID (described next).

```
#include    "ourhdr.h"

int
main(void)
{
    printf("uid = %d, gid = %d\n", getuid(), getgid());
    exit(0);
}
```

Program 1.7 Print user ID and group ID.

We call the functions `getuid` and `getgid` to return the user ID and group ID. Running the program yields

```
$ a.out
uid = 224, gid = 20
```

□

Group ID

Our entry in the password file also specifies our numeric *group ID*. This group ID is also assigned by the system administrator when our login name is assigned. Typically there are multiple entries in the password file that specify the same group ID. Groups are normally used under Unix to collect users together into projects or departments. This allows the sharing of resources (such as files) between members of the same group. We'll see in Section 4.5 that we can set the permissions on a file so that all members of a group can access the file, while others outside the group cannot.

There is also a group file that maps group names into numeric group IDs. The group file is usually `/etc/group`.

The use of numeric user IDs and numeric group IDs for permissions is historical. The directory entry for every file on the system contains both the user ID and the group ID of the owner of the file. Storing both of these values in the directory entry requires only four bytes, assuming each is stored as a two-byte integer. If the full eight-byte login name and eight-byte group name were used instead, additional disk space would be required. Users, however, work better with names instead of numbers, so the password file maintains the mapping between login names and user IDs, and the group file provides the mapping between group names and group IDs. The Unix `ls -l` command, for example, prints the login name of the owner of a file, using the password file to map the numeric user ID into the corresponding login name.

Supplementary Group IDs

In addition to the group ID specified in the password file for a login name, some versions of Unix allow a user to belong to additional groups. This started with 4.2BSD, which allowed a user to belong to up to 16 additional groups. These *supplementary group IDs* are obtained at login time by reading the file `/etc/group` and finding the first 16 entries that list the user as a member.

1.9 Signals

Signals are a technique used to notify a process that some condition has occurred. For example, if a process divides by zero, the signal whose name is `SIGFPE` is sent to the process. The process has three choices for dealing with the signal:

1. Ignore the signal. This isn't recommended for signals that denote a hardware exception, such as dividing by zero, or referencing memory outside the address space of the process, since the results are undefined.
2. Let the default action occur. For a divide by zero the default is to terminate the process.
3. Provide a function that is called when the signal occurs. By providing a function of our own, we'll know when the signal occurs and we can handle it as we wish.

Many conditions generate signals. There are two terminal keys, called the *interrupt key* (often the DELETE key or Control-C) and the *quit key* (often Control-backslash). These are used to interrupt the currently running process. Another way to generate a signal is by calling the function named `kill`. We can call this function from a process to send a signal to another process. Naturally there are limitations: we have to be the owner of the other process to be able to send it a signal.

Example

Recall the bare bones shell example (Program 1.5). If we invoke this program and type the interrupt key, the process terminates. The reason is that the default action for this signal (named `SIGINT`) is to terminate the process. The process hasn't told the kernel to do anything other than the default with this signal, so the process terminates.

To change this program so that it catches this signal, it needs to call the `signal` function, specifying the name of the function to call when the `SIGINT` signal is generated. The function is named `sig_int` and when it's called it just prints a message and a new prompt. Adding 12 lines to Program 1.5 gives us the version in Program 1.8. (The 12 new lines are indicated with a plus sign at the beginning of the line.)

```
  #include    <sys/types.h>
  #include    <sys/wait.h>
+ #include    <signal.h>
  #include    "ourhdr.h"

+ static void sig_int(int);       /* our signal-catching function */
+
  int
  main(void)
  {
      char    buf[MAXLINE];
      pid_t   pid;
      int     status;

+     if (signal(SIGINT, sig_int) == SIG_ERR)
+         err_sys("signal error");
+
      printf("%% ");  /* print prompt (printf requires %% to print %) */
      while (fgets(buf, MAXLINE, stdin) != NULL) {
          buf[strlen(buf) - 1] = 0;   /* replace newline with null */

          if ( (pid = fork()) < 0)
              err_sys("fork error");

          else if (pid == 0) {        /* child */
              execlp(buf, buf, (char *) 0);
              err_ret("couldn't execute: %s", buf);
              exit(127);
          }

          /* parent */
          if ( (pid = waitpid(pid, &status, 0)) < 0)
              err_sys("waitpid error");
          printf("%% ");
      }
      exit(0);
+ }
+
```

```
+ void
+ sig_int(int signo)
+ {
+     printf("interrupt\n%% ");
  }
```

Program 1.8 Read commands from standard input and execute them.

In Chapter 10 we'll take a long look at signals, since most nontrivial applications deal with them. □

1.10 Unix Time Values

Historically, two different time values have been maintained by Unix systems.

1. Calendar time. This value counts the number of seconds since the Epoch, which is 00:00:00 January 1, 1970, Coordinated Universal Time (UTC). (Older manuals refer to UTC as Greenwich Mean Time.) These time values are used to record the time that a file was last modified, for example.

   The primitive system data type `time_t` holds these time values.

2. Process time. This is also called CPU time and measures the central processor resources used by a process. Process time is measured in clock ticks, which have historically been 50, 60, or 100 ticks per second.

   The primitive system data type `clock_t` holds these time values. Further, POSIX defines the constant `CLK_TCK` to specify the number of ticks per second. (The constant `CLK_TCK` is now obsolete. We'll show how to obtain the number of clock ticks per second with the `sysconf` function in Section 2.5.4.)

When we measure the execution time of a process, as in Section 3.9, we'll see that Unix maintains three values for a process:

- clock time
- user CPU time
- system CPU time

The clock time is sometimes called *wall clock time*. It is the amount of time the process takes to run, and its value depends on the number of other processes being run on the system. Whenever we report the clock time, the measurements are made with no other activities on the system.

The user CPU time is the CPU time that is attributed to user instructions. The system CPU time is the CPU time that can be attributed to the kernel, when it executes on behalf of the process. For example, whenever a process executes a system service, such as `read` or `write`, the time spent within the kernel performing that system service is charged to the process. The sum of the user CPU time and system CPU time is often called the *CPU time*.

It is easy to measure the clock time, user time, and system time of any process—just execute the `time`(1) command with the argument to the `time` command being the command we want to measure. For example,

```
$ cd /usr/include
$ time grep _POSIX_SOURCE */*.h > /dev/null

real    0m19.81s
user    0m0.43s
sys     0m4.53s
```

The output format from the `time` command depends on the shell being used.

In Section 8.15 we see how to obtain these three times from a running process. The general topic of times and dates is covered in Section 6.9.

1.11 System Calls and Library Functions

All operating systems provide service points through which programs request services from the kernel. All variants of Unix provide a well-defined, limited number of entry points directly into the kernel called *system calls*. The system calls are one feature of Unix that we cannot change. Unix Version 7 provided about 50 system calls, 4.3+BSD provides about 110, and SVR4 has around 120.

The system call interface has always been documented in Section 2 of the *Unix Programmer's Manual*. Its definition is in the C language, regardless of the actual implementation technique used on any given system to invoke a system call. This differs from many older operating systems, which traditionally defined the kernel entry points in the assembler language of the machine.

The technique used on Unix systems is for each system call to have a function of the same name in the standard C library. The user process calls this function, using the standard C calling sequence. This function then invokes the appropriate kernel service, using whatever technique is required on the system. For example, the function may put one or more of the C arguments into general registers and then execute some machine instruction that generates a software interrupt in the kernel. For our purposes, we can consider the system calls as being C functions.

Section 3 of the *Unix Programmer's Manual* defines the general purpose functions available to programmers. These functions are not entry points into the kernel, although they may invoke one or more of the kernel's system calls. For example, the `printf` function may invoke the `write` system call to perform the output, but the functions `strcpy` (copy a string) and `atoi` (convert ASCII to integer) don't involve the operating system at all.

From an implementor's point of view, the distinction between a system call and a library function is fundamental. But from a user's perspective, the difference is not as critical. From our perspective in this text, both system calls and library functions appear as normal C functions. Both exist to provide services for application programs. We should realize, however, that we can replace the library functions if desired, whereas the system calls usually cannot be replaced.

Consider the memory allocation function `malloc` as an example. There are many ways to do memory allocation and its associated garbage collection (best fit, first fit, etc.). No single technique is optimal for all programs. The Unix system call that handles memory allocation, `sbrk`(2), is not a general purpose memory manager. It increases or decreases the address space of the process by a specified number of bytes. How that space is managed is up to the process. The memory allocation function, `malloc`(3), implements one particular type of allocation. If we don't like its operation we can define our own `malloc` function, which will probably use the `sbrk` system call. There are, in fact, numerous software packages that implement their own memory allocation algorithms, with the `sbrk` system call. Figure 1.1 shows the relationship between the application, the `malloc` function, and the `sbrk` system call.

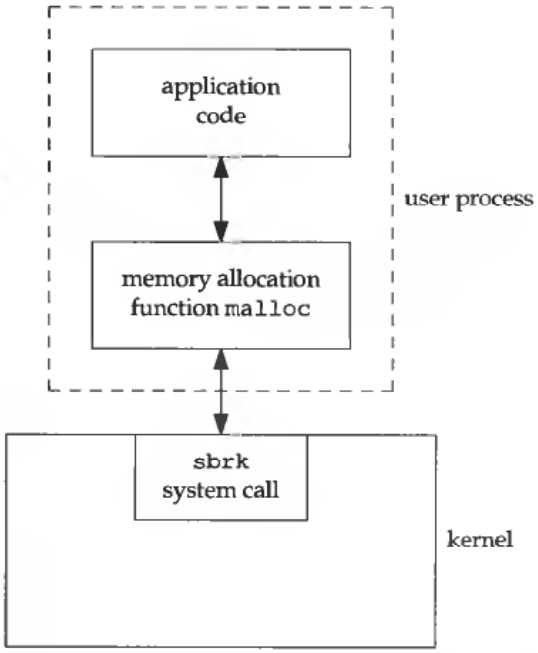


Figure 1.1 Separation of `malloc` function and `sbrk` system call.

Here we have a clean separation of duties—the system call in the kernel allocates an additional chunk of space to the process. The library function `malloc` manages this space.

Another example to illustrate the difference between a system call and a library function is the interface provided by Unix to determine the current time and date. Some operating systems provide one system call to return the time and another to return the date. Any special handling, such as the switch to or from daylight savings time, is handled by the kernel or requires human intervention. Unix, on the other hand, provides a single system call that returns the number of seconds since the Epoch: midnight, January 1, 1970, Coordinated Universal Time. Any interpretation of this value, such as converting it to a human-readable time and date using the local time zone, is left to the user process. Routines are provided in the standard C library to handle most

cases. These library routines handle details such as the various daylight savings time algorithms.

An application can call either a system call or a library routine. Also realize that many library routines invoke a system call. This is shown in Figure 1.2.

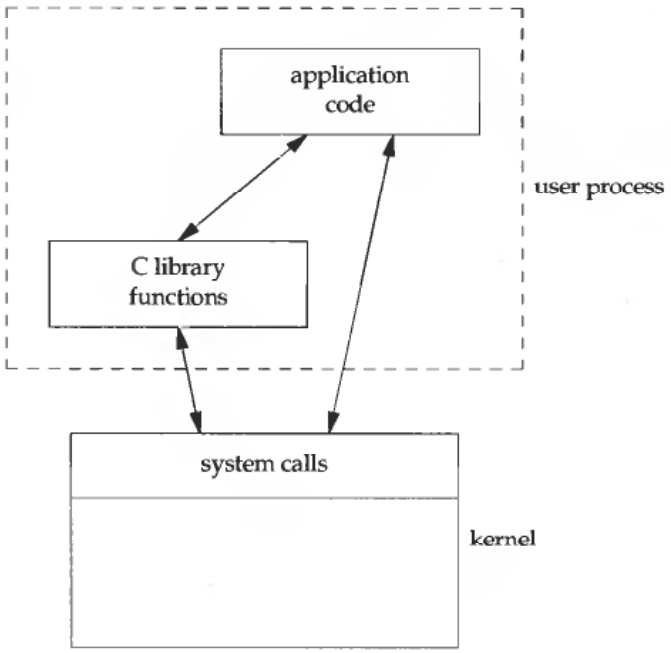


Figure 1.2 Difference between C library functions and system calls.

Another difference between system calls and library functions is that system calls usually provide a minimal interface while library functions often provide more elaborate functionality. We've seen this already in the difference between the `sbrk` system call and the `malloc` library function. We'll see this difference later when we compare the unbuffered I/O functions in Chapter 3 and the standard I/O functions in Chapter 5.

The process control system calls (`fork`, `exec`, and `wait`) are usually invoked by the user's application code directly. (Recall the bare bones shell in Program 1.5.) But some library routines exist to simplify certain common cases: the `system` and `popen` library routines, for example. In Section 8.12 we'll show an implementation of the `system` function that invokes the basic process control system calls. We'll enhance this example in Section 10.18 to handle signals correctly.

To define the interface to the Unix system that most programmers utilize, we have to describe both the system calls and some of the library functions. If we described only the `sbrk` system call, for example, we would skip the `malloc` library function that many applications utilize.

In this text we'll use the term *function* to refer to both system calls and library functions, except when the distinction is necessary.

1.12 Summary

This chapter has been a whirlwind tour of Unix. We've described some of the fundamental terms that we'll encounter over and over again. We've seen numerous small examples of Unix programs to give us a feel for what the remainder of the text talks about.

The next chapter is about Unix standardization and the effect of recent work in this area on current systems. Standards, particularly the ANSI C standard and the POSIX.1 standard, will affect the rest of the text.

Exercises

1.1 Verify on your system that the directories dot and dot-dot are not the same, except in the root directory.

1.2 In the output from Program 1.4, what happened to the processes with process IDs 852 and 853?

1.3 In Section 1.7 the argument to `perror` is defined with the ANSI C attribute `const`, while the integer argument to `strerror` isn't defined with this attribute. Why?

1.4 In the error handling function `err_sys` in Appendix B, why is the value of `errno` saved when the function is called?

1.5 If the calendar time is stored as a signed 32-bit integer, in what year will it overflow?

1.6 If the process time is stored as a signed 32-bit integer, and if the system counts 100 ticks per second, after how many days will the value overflow?

2

Unix Standardization and Implementations

2.1 Introduction

Much work has gone into standardizing the various flavors of Unix and the C programming language. Although Unix applications have always been quite portable between different versions of Unix, the proliferation of versions and differences during the 1980s led many large users (such as the U.S. government) to lead the call for standardization.

In this chapter we first look at the various standardization efforts that are underway. We then discuss the effects of these standards on the actual Unix implementations that are described in this book. An important part of all the standardization efforts is the specification of various limits that each implementation must define, so we look at these limits and the various ways to determine their values.

2.2 Unix Standardization

2.2.1 ANSI C

In late 1989 the ANSI Standard X3.159–1989 for the C programming language was approved [ANSI 1989]. This standard has also been adopted as international standard ISO/IEC 9899:1990. ANSI is the American National Standards Institute. It is a nonprofit organization composed of vendors and users. It is the national clearinghouse for voluntary standards in the United States and is the U.S. member in the International Organization for Standardization (ISO).

The intent of the ANSI C standard is to provide portability of conforming C programs to a wide variety of operating systems, not just Unix. This standard defines not

only the syntax and semantics of the programming language but also a standard library [Chapter 4 of ANSI 1989; Plauger 1992; Appendix B of Kernighan and Ritchie 1988]. This library is important to us because many newer Unix systems (such as the ones described in this book) provide the library routines that are specified in the C standard.

This library can be divided into 15 areas based on the headers defined by the standard. Figure 2.1 lists the headers defined by the C standard, along with the headers defined by the other two standards that we describe in the following sections (POSIX.1 and XPG3). We also list which of these headers are supported by the two implementations (SVR4 and 4.3+BSD) that are described later in this chapter.

2.2.2 IEEE POSIX

POSIX is a family of standards developed by the IEEE (Institute of Electrical and Electronics Engineers). POSIX stands for Portable Operating System Interface for Computer Environments. It originally referred only to the IEEE Standard 1003.1–1988 (the operating system interface), but the IEEE is currently working on other related standards in the POSIX family. For example, 1003.2 will be a standard for shells and utilities, and 1003.7 will be a standard for system administration. There are over 15 other subcommittees in the 1003 working group.

Of specific interest to this book is the 1003.1 operating system interface standard. This standard defines the services that must be provided by an operating system if it is to be "POSIX compliant" and is being adopted by most computer vendors. Although the 1003.1 standard is based on the Unix operating system, the standard is not restricted to Unix and Unix-like systems. Indeed, there are vendors that supply proprietary operating systems who claim that these systems will be made POSIX compliant (while still leaving all their proprietary features in place).

Because the 1003.1 standard specifies an *interface* and not an *implementation*, no distinction is made between system calls and library functions. All the routines in the standard are called *functions*.

Standards are continually evolving, and the 1003.1 standard is no exception. The 1988 version of this standard, IEEE Standard 1003.1–1988, was modified and submitted to the International Organization for Standardization. No new interfaces or features were added but the text was revised. The resulting document was published as IEEE Std 1003.1–1990 [IEEE 1990]. This is also the international standard ISO/IEC 9945–1:1990. This standard is commonly referred to as *POSIX.1*, which we'll use in this text.

The IEEE 1003.1 working group then made more changes, which should be approved by 1993. These changes (currently called 1003.1a) should be published by the IEEE as a supplement to IEEE Standard 1003.1–1990. These changes affect this text, primarily because Berkeley-style symbolic links will probably be added as a required feature. The changes will probably become an addendum to ISO/IEC 9945–1:1990. In this text we describe the 1003.1a version of POSIX.1 with notes specifying which features will probably be added with 1003.1a.

POSIX.1 does not include the notion of a superuser. Instead, certain operations require "appropriate privileges," although POSIX.1 leaves the definition of this term up

Header	Standards			Implementations		Description
	ANSI C	POSIX.1	XPG3	SVR4	4.3+BSD	
<assert.h>	•			•	•	verify program assertion
<cpio.h>		•		•		cpio archive values
<ctype.h>	•			•	•	character types
<dirent.h>		•	•	•	•	directory entries (Section 4.21)
<errno.h>	•			•	•	error codes (Section 1.7)
<fcntl.h>		•	•	•	•	file control (Section 3.13)
<float.h>	•			•	•	floating point constants
<ftw.h>			•	•		file tree walking (Section 4.21)
<grp.h>		•	•	•	•	group file (Section 6.4)
<langinfo.h>			•	•		language information constants
<limits.h>	•			•	•	implementation constants (Section 2.5)
<locale.h>	•			•	•	locale categories
<math.h>	•			•	•	mathematical constants
<nl_types.h>			•	•		message catalogs
<pwd.h>		•	•	•	•	password file (Section 6.2)
<regex.h>			•	•	•	regular expressions
<search.h>			•	•		search tables
<setjmp.h>	•			•	•	nonlocal goto (Section 7.10)
<signal.h>	•			•	•	signals (Chapter 10)
<stdarg.h>	•			•	•	variable argument lists
<stddef.h>	•			•	•	standard definitions
<stdio.h>	•			•	•	standard I/O library (Chapter 5)
<stdlib.h>	•			•	•	utility functions
<string.h>	•			•	•	string operations
<tar.h>		•		•		tar archive values
<termios.h>		•	•	•	•	terminal I/O (Chapter 11)
<time.h>	•			•	•	time and date (Section 6.9)
<ulimit.h>			•	•		user limits
<unistd.h>		•	•	•	•	symbolic constants
<utime.h>		•	•	•	•	file times (Section 4.19)
<sys/ipc.h>			•	•	•	IPC (Section 14.6)
<sys/msg.h>			•	•		message queues (Section 14.7)
<sys/sem.h>			•	•		semaphores (Section 14.8)
<sys/shm.h>			•	•	•	shared memory (Section 14.9)
<sys/stat.h>		•	•	•	•	file status (Chapter 4)
<sys/times.h>		•	•	•	•	process times (Section 8.15)
<sys/types.h>		•	•	•	•	primitive system data types (Section 2.7)
<sys/utsname.h>		•	•	•		system name (Section 6.8)
<sys/wait.h>		•	•	•	•	process control (Section 8.6)

Figure 2.1 Headers defined by the various standards and implementations.

to the implementation. Some newer Unix systems, which conform to the Department of Defense security guidelines, have many different levels of security. In this text, however, we use the traditional Unix terminology and refer to operations that require superuser privilege.

2.2.3 X/Open XPG3

X/Open is an international group of computer vendors. They have produced a seven-volume portability guide called the *X/Open Portability Guide*, Issue 3 [X/Open 1989]. We'll call this *XPG3*. Volume 2 of XPG3 (*XSI System Interface and Headers*) defines an interface to a Unix-like system that is built on the IEEE Std. 1003.1–1988 interface. But XPG3 contains additional features that are not in POSIX.1.

For example, one feature that is in XPG3 but not in POSIX.1 is the X/Open messaging facility. This facility can be used by applications to display text messages in different languages.

One thing to be aware of is that the XPG3 interface was built on the draft ANSI C standard, not the final standard. For this reason a few features in the XPG3 interface specification are out of date. These will probably be fixed in a future release of the XPG specification. (Work is underway on XPG4, and it will probably be completed by 1993.)

2.2.4 FIPS

FIPS stands for Federal Information Processing Standard, and these standards are published by the U.S. government. They are used for procurement of computer systems by the U.S. government. FIPS 151–1 (April 1989) is based on the IEEE Std. 1003.1–1988 and a draft of the ANSI C standard. FIPS 151–1 requires some features that POSIX.1 lists as optional. This FIPS is sometimes called the POSIX.1 FIPS. Section 2.5.5 lists the POSIX.1 options required by the FIPS.

The effect of the POSIX.1 FIPS is to require any vendor who wishes to sell POSIX.1-compliant computer systems to the U.S. government to support some of the optional features of POSIX.1. We won't consider the POSIX.1 FIPS as another standard, since practically it is just a tightening of the POSIX.1 standard.

2.3 Unix Implementations

The previous section described three standards done by independent organizations: ANSI C, IEEE POSIX, and the X/Open XPG3. Standards, however, are interface specifications. How do these standards relate to the real world? These standards are taken by vendors and turned into actual implementations. In this book we are interested in both these standards and their actual implementation.

Section 1.1 of Leffler et al. [1989] gives a detailed history (and a nice picture) of the Unix family tree. Everything starts from the Sixth Edition (1976) and Seventh Edition (1979) of the Unix Time-Sharing System on the PDP-11 (usually called Version 6 and Version 7). These were the first releases widely distributed outside of Bell Labs. Three branches of the tree evolved: (a) one at AT&T that led to System III and System V (the so-called commercial versions of Unix), (b) one at the University of California at Berkeley that led to the 4.xBSD implementations, (c) the research version of Unix, under continuing development at the Computing Science Research Center of AT&T Bell Laboratories, that led to the 8th, 9th, and 10th Editions.

2.3.1 System V Release 4

System V Release 4 (SVR4) is a product of AT&T's Unix System Laboratories. It is a merging of AT&T Unix System V Release 3.2 (SVR3.2), Sun Microsystem's SunOS system, the 4.3BSD release from the University of California, and the Xenix system from Microsoft. (Xenix was originally developed from Version 7, with many features later taken from System V.) The source code was released in late 1989 with the first end-user copies being available during 1990. SVR4 conforms to both the POSIX 1003.1 standard and the X/Open XPG3 standard.

AT&T also publishes the System V Interface Definition (SVID) [AT&T 1989]. Issue 3 of the SVID specifies the functionality that a Unix system must offer to qualify as Unix System V Release 4. As with POSIX.1, the SVID specifies an interface and not an implementation. No distinction is made in the SVID between system calls and library functions. The reference manual for an actual implementation of SVR4 must be consulted to see this distinction [AT&T 1990e].

SVR4 contains a Berkeley compatibility library [AT&T 1990c] that provides functions and commands that operate like their 4.3BSD counterparts. Some of these functions, however, differ from their POSIX counterparts. None of the SVR4 examples in this text use this compatibility library. This library should be used only if you have an older application that you do not want to convert. New applications should not use it.

2.3.2 4.3+BSD

The Berkeley Software Distributions (BSD) are produced and distributed by the Computer Systems Research Group at the University of California at Berkeley. 4.2BSD was released in 1983 and 4.3BSD in 1986. Both of these releases ran on the VAX minicomputer. The next release, 4.3BSD Tahoe in 1988, also ran on a particular minicomputer called the Tahoe. (The book by Leffler et al. [1989] describes the 4.3BSD Tahoe release.) This was followed in 1990 with the 4.3BSD Reno release. 4.3BSD Reno supported many of the POSIX.1 features. The next major release, 4.4BSD, should be released in 1992.

The original BSD systems contained proprietary AT&T source code and were covered by AT&T licenses. To obtain the source code to the BSD system you had to have an AT&T source license for Unix. This has been changing as more and more of the AT&T source code has been replaced over the years with non-AT&T source code, and many of the new features added to the Berkeley system were derived from non-AT&T sources.

In 1989 Berkeley identified much of the non-AT&T source code in the 4.3BSD Tahoe release and made it publicly available as the BSD Networking Software, Release 1.0. This was followed in 1991 with Release 2.0 of the BSD Networking Software, which was derived from the 4.3BSD Reno release. The intent is that most, if not all, of the 4.4BSD system will be free of any AT&T license restrictions. This will make the source code available to all.

> As we mentioned in the preface, throughout the text we use the term 4.3+BSD to refer to the BSD system being described. This system is between the BSD Networking Software Release 2.0 and the forthcoming 4.4BSD.

The Unix development done at Berkeley started with PDP-11s, then moved to the VAX minicomputer, and has since moved to other so-called workstations. During the early 1990s support was provided to Berkeley for the popular 80386-based personal computers, leading to what is called 386BSD. This was done by Bill Jolitz and is documented in a series of monthly articles in *Dr. Dobb's Journal* throughout 1991. Much of this code appears in the BSD Networking Software, Release 2.0.

2.4 Relationship of Standards and Implementations

The standards that we've mentioned define a subset of any actual system. Although the IEEE POSIX efforts plan to define standards in other required areas (such as the networking interface, communication between different processes, and system administration), these additional standards don't exist at the time of this writing.

The focus of this book is to describe two real Unix systems: SVR4 and 4.3+BSD. Since both claim to be POSIX compliant we will also concentrate on the features that are required by the POSIX.1 standard, noting any differences between POSIX and the actual implementations of these two systems. Those features and routines that are specific only to SVR4 or 4.3+BSD are clearly marked. Since XPG3 is a superset of POSIX.1, we'll also note any features that are part of XPG3 but not part of POSIX.1.

Be aware that both of the implementations (SVR4 and 4.3+BSD) provide backward compatibility for features in earlier releases (such as SVR3.2 and 4.3BSD). For example, SVR4 supports both the POSIX.1 specification for nonblocking I/O (`O_NONBLOCK`) and the traditional System V method (`O_NDELAY`). In this text we'll use only the POSIX.1 feature, although we'll mention the nonstandard feature that it replaces. Similarly, both SVR3.2 and 4.3BSD provided reliable signals in a way that differs from the POSIX.1 standard. In Chapter 10 we describe only the POSIX.1 signal mechanism.

2.5 Limits

There are many magic numbers and constants that are defined by the implementation. Many of these have been hard coded into programs or were determined using ad hoc techniques. With the various standardization efforts that we've described, more portable methods are now provided to determine these magic numbers and implementation-defined limits. This can greatly aid the portability of our software.

Three types of features are needed:

- compile-time options (does the system support job control?)
- compile-time limits (what's the largest value of a short integer?)
- run-time limits (how many characters in a filename?)

The first two features, compile-time options and compile-time limits, can be defined in headers that any program can include at compile time. But run-time limits require the process to call a function to obtain the value of the limit.

Additionally, some limits can be fixed on a given implementation (and could therefore be defined statically in a header) yet vary on another implementation (and would require a run-time function call). An example of this type of limit is the maximum number of characters in a filename. System V has historically allowed only 14 characters in a filename while Berkeley-derived systems increased this to 255. SVR4 allows us to specify, for each filesystem that we create, whether it is a System V filesystem or a BSD filesystem, and each has a different limit. This is the case of a run-time limit that depends where in the filesystem the file in question is located. A filename in the root filesystem, for example, could have a 14-character limit, while a filename in some other filesystem could have a 255-character limit.

To solve these problems, three types of limits are provided:

1. compile-time options and limits (headers);
2. run-time limits that are not associated with a file or directory (the `sysconf` function);
3. run-time limits that are associated with a file or directory (the `pathconf` and `fpathconf` functions).

To further confuse things, if a particular run-time limit does not vary on a given system, it can be defined statically in a header. If it is not defined in a header, however, the application must call one of the three `conf` functions (which we describe shortly) to determine its value at run time.

2.5.1 ANSI C Limits

All the limits defined by ANSI C are compile-time limits. Figure 2.2 shows the limits from the C standard that are defined in the file `<limits.h>`. These constants are always defined in the header and don't change in a given system. The third column shows the minimum acceptable values from the ANSI C standard. This allows for a system with 16-bit integers using 1's-complement arithmetic. The fourth column shows the values from a current system with 32-bit integers using 2's-complement arithmetic. Note that none of the unsigned data types has a minimum value, as this value must be 0 for an unsigned data type.

One difference that we will encounter is whether a system provides signed or unsigned character values. From the fourth column in Figure 2.2 we see that this particular system uses signed characters. We see that `CHAR_MIN` equals `SCHAR_MIN` and `CHAR_MAX` equals `SCHAR_MAX`. If the system uses unsigned characters we would have `CHAR_MIN` equal to 0 and `CHAR_MAX` equal to `UCHAR_MAX`.

There is a similar set of definitions for the floating point data types in the header `<float.h>`. Anyone doing serious floating point work should examine this file.

Another constant from ANSI C that we'll encounter is `FOPEN_MAX`, the minimum number of standard I/O streams that can be open at once. This value is in the `<stdio.h>` header, and its minimum value is 8. The POSIX.1 value `STREAM_MAX`, if defined, must have the same value as `FOPEN_MAX`.

Name	Description	Minimum acceptable value	Typical value
CHAR_BIT	bits in a char	8	8
CHAR_MAX	max value of char	(see later)	127
CHAR_MIN	min value of char	(see later)	–128
SCHAR_MAX	max value of signed char	127	127
SCHAR_MIN	min value of signed char	–127	–128
UCHAR_MAX	max value of unsigned char	255	255
INT_MAX	max value of int	32,767	2,147,483,647
INT_MIN	min value of int	–32,767	–2,147,483,648
UINT_MAX	max value of unsigned int	65,535	4,294,967,295
SHRT_MIN	min value of short	–32,767	–32,768
SHRT_MAX	max value of short	32,767	32,767
USHRT_MAX	max value of unsigned short	65,535	65,535
LONG_MAX	max value of long	2,147,483,647	2,147,483,647
LONG_MIN	min value of long	–2,147,483,647	–2,147,483,648
ULONG_MAX	max value of unsigned long	4,294,967,295	4,294,967,295
MB_LEN_MAX	max number of bytes in a multibyte character constant	1	1

Figure 2.2 Sizes of integral values from <limits.h>.

ANSI C also defines the constant TMP_MAX in <stdio.h>. It is the maximum number of unique filenames generated by the tmpnam function. We'll have more to say about this constant in Section 5.13.

2.5.2 POSIX Limits

POSIX.1 defines numerous constants that deal with implementation limits of the operating system. Unfortunately, this is one of the more confusing aspects of POSIX.1. (Reading and comprehending Sections 2.8 and 2.9 of the POSIX.1 standard are an exercise in deciphering "standardese.")

There are 33 different limits and constants. These are divided into the following eight categories:

1. Invariant minimum values (the 13 constants in Figure 2.3).
2. Invariant value: SSIZE_MAX.
3. Run-time increasable value: NGROUPS_MAX.
4. Run-time invariant values (possibly indeterminate): ARG_MAX, CHILD_MAX, OPEN_MAX, STREAM_MAX, and TZNAME_MAX.
5. Pathname variable values (possibly indeterminate): LINK_MAX, MAX_CANON, MAX_INPUT, NAME_MAX, PATH_MAX, and PIPE_BUF.
6. Compile-time symbolic constants: _POSIX_SAVED_IDS, _POSIX_VERSION, and _POSIX_JOB_CONTROL.

7. Execution-time symbolic constants: _POSIX_NO_TRUNC, _POSIX_VDISABLE, and _POSIX_CHOWN_RESTRICTED.
8. Obsolete constant: CLK_TCK.

Of these 33 limits and constants, 15 are always defined and others may or may not be defined, depending on certain conditions. We describe the limits and constants that may or may not be defined (items 4–8) in Section 2.5.4, when we describe the sysconf, pathconf, and fpathconf functions. We summarize all the limits and constants in Figure 2.7. The 13 invariant minimum values are shown in Figure 2.3.

Name	Description: minimum acceptable value for	Value
_POSIX_ARG_MAX	length of arguments to exec functions	4096
_POSIX_CHILD_MAX	number of child processes per real user ID	6
_POSIX_LINK_MAX	number of links to a file	8
_POSIX_MAX_CANON	number of bytes on a terminal's canonical input queue	255
_POSIX_MAX_INPUT	space available on a terminal's input queue	255
_POSIX_NAME_MAX	number of bytes in a filename	14
_POSIX_NGROUPS_MAX	number of simultaneous supplementary group IDs per process	0
_POSIX_OPEN_MAX	number of open files per process	16
_POSIX_PATH_MAX	number of bytes in a pathname	255
_POSIX_PIPE_BUF	number of bytes that can be written atomically to a pipe	512
_POSIX_SSIZE_MAX	value that can be stored in ssize_t object	32767
_POSIX_STREAM_MAX	number of standard I/O streams a process can have open at once	8
_POSIX_TZNAME_MAX	number of bytes for the name of a time zone	3

Figure 2.3 POSIX.1 invariant minimum values from <limits.h>.

These values are invariant—they do not change from one system to another. They specify the most restrictive values for these features. A conforming POSIX.1 implementation must provide values that are at least this large. This is why they are called minimums, although their names all contain MAX. Also, a portable application must not require a larger value. We describe what each of these constants refers to as we proceed through the text.

Unfortunately, some of these invariant minimum values are too small to be of practical use. For example, most Unix systems today provide far more than 16 open files per process. Even Version 7 in 1978 provided 20 open files per process! Also, the minimum limit of 255 for _POSIX_PATH_MAX is too small. Pathnames can exceed this limit. This means that we can't use the two constants _POSIX_OPEN_MAX and _POSIX_PATH_MAX as array sizes at compile time, for example.

Each of the 13 invariant minimum values in Figure 2.3 has an associated implementation value whose name is formed by removing the _POSIX_ prefix from the name in Figure 2.3. The names without the leading _POSIX_ were intended to be the actual values that a given implementation supports. (These 13 implementation values are items 2–5 from our list earlier in this section: the invariant value, the run-time increasable value, the run-time invariant values, and the pathname variable values.) The problem is that not all of the 13 implementation values are guaranteed to be defined in the <limits.h> header. The reason a particular value may not be defined in the header is

because its actual value for a given process may depend on the amount of memory on the system, for example. If they're not defined in the header, we can't use them as array bounds at compile time. So, POSIX.1 decided to provide three run-time functions for us to call, `sysconf`, `pathconf`, and `fpathconf`, to determine the actual implementation value at run time. There is still a problem, however, because some of the values are defined by POSIX.1 as being possibly "indeterminate" (logically infinite). This means that the value has no practical upper bound. For example, the limit on the number of open files per process in SVR4 is virtually unlimited, so `OPEN_MAX` is considered indeterminate under SVR4. We'll return to this problem of indeterminate run-time limits in Section 2.5.7.

2.5.3 XPG3 Limits

XPG3 defines seven constants that always appear in the `<limits.h>` header. POSIX.1 would call these invariant minimum values. They are listed in Figure 2.4. Most of these values deal with message catalogs.

Name	Description	Minimum acceptable value	Typical value
`NL_ARGMAX`	maximum value of digit in calls to `printf` and `scanf`	9	9
`NL_LANGMAX`	maximum number of bytes in `LANG` environment variable	14	14
`NL_MSGMAX`	maximum message number	32,767	32,767
`NL_NMAX`	maximum number of bytes in N-to-1 mapping characters		1
`NL_SETMAX`	maximum set number	255	255
`NL_TEXTMAX`	maximum number of bytes in a message string	255	255
`NZERO`	default process priority	20	20

Figure 2.4 XPG3 invariant minimum values from `<limits.h>`.

XPG3 also defines the value `PASS_MAX`, which can appear in `<limits.h>` as the maximum number of significant characters in a password (not including the terminating null byte). POSIX.1 would call this value a run-time invariant value (possibly indeterminate). The minimum acceptable value is 8. The value of `PASS_MAX` can also be obtained at run time with the `sysconf` function, as described in Section 2.5.4.

2.5.4 `sysconf`, `pathconf`, and `fpathconf` Functions

We've listed various minimum values that an implementation must support, but how do we find out the limits that a particular system actually supports? As we mentioned earlier, some of these limits might be available at compile time and others must be determined at run time. We've also mentioned that some don't change in a given system, while others are associated with a file or directory. The run-time limits are obtained by calling one of the following three functions.

```
#include <unistd.h>

long sysconf(int name);

long pathconf(const char *pathname, int name);

long fpathconf(int filedes, int name);
```

All three return: corresponding value if OK, –1 on error (see later)

The difference between the last two functions is that one takes a pathname as its argument and the other takes a file descriptor argument.

Figure 2.5 lists the *name* arguments that are used by these three functions. Constants beginning with `_SC_` are used as arguments to `sysconf` and arguments beginning with `_PC_` are used as arguments to either `pathconf` or `fpathconf`.

There are some restrictions for the *pathname* argument to `pathconf` and the *filedes* argument to `fpathconf`. If any of these restrictions aren't met, the results are undefined.

1. The referenced file for `_PC_MAX_CANON`, `_PC_MAX_INPUT`, and `_PC_VDISABLE` must be a terminal file.
2. The referenced file for `_PC_LINK_MAX` can be either a file or directory. If it is a directory, the return value applies to the directory itself (not the filename entries within the directory).
3. The referenced file for `_PC_NAME_MAX` and `_PC_NO_TRUNC` must be a directory. The return value applies to filenames within the directory.
4. The referenced file for `_PC_PATH_MAX` must be a directory. The value returned is the maximum length of a relative pathname when the specified directory is the working directory. (Unfortunately this isn't the real maximum length of an absolute pathname, which is what we want to know. We'll return to this problem in Section 2.5.7.)
5. The referenced file for `_PC_PIPE_BUF` must be a pipe, FIFO, or directory. In the first two cases (pipe or FIFO) the return value is the limit for the referenced pipe or FIFO. For the other case (a directory) the return value is the limit for any FIFO created in that directory.
6. The referenced file for `_PC_CHOWN_RESTRICTED` must be either a file or directory. If it is a directory, the return value indicates whether this option applies to files within that directory.

We need to specify in more detail the different return values from these three functions.

1. All three functions return –1 and set `errno` to `EINVAL` if the *name* isn't one of the appropriate constants from the third column of Figure 2.5.

Name of limit	Description	*name* argument
ARG_MAX	maximum length of arguments to the exec functions (in bytes)	_SC_ARG_MAX
CHILD_MAX	maximum number of processes per real user ID	_SC_CHILD_MAX
clock ticks/second	number of clock ticks per second	_SC_CLK_TCK
NGROUPS_MAX	maximum number of simultaneous supplementary process group IDs per process	_SC_NGROUPS_MAX
OPEN_MAX	maximum number of open files per process	_SC_OPEN_MAX
PASS_MAX	maximum number of significant characters in a password (XPG3 and SVR4, not POSIX.1)	_SC_PASS_MAX
STREAM_MAX	maximum number of standard I/O streams per process at any given time—if defined, it must have the same value as FOPEN_MAX	_SC_STREAM_MAX
TZNAME_MAX	maximum number of bytes for the name of a time zone	_SC_TZNAME_MAX
_POSIX_JOB_CONTROL	indicates if the implementation supports job control	_SC_JOB_CONTROL
_POSIX_SAVED_IDS	indicates if the implementation supports the saved set-user-ID and the saved set-group-ID	_SC_SAVED_IDS
_POSIX_VERSION	indicates the POSIX.1 version	_SC_VERSION
_XOPEN_VERSION	indicates the XPG version (not POSIX.1)	_SC_XOPEN_VERSION
LINK_MAX	maximum value of a file's link count	_PC_LINK_MAX
MAX_CANON	maximum number of bytes on a terminal's canonical input queue	_PC_MAX_CANON
MAX_INPUT	number of bytes for which space is available on terminal's input queue	_PC_MAX_INPUT
NAME_MAX	maximum number of bytes in a filename (does not include a null at end)	_PC_NAME_MAX
PATH_MAX	maximum number of bytes in a relative pathname (does not include a null at end)	_PC_PATH_MAX
PIPE_BUF	maximum number of bytes that can be written atomically to a pipe	_PC_PIPE_BUF
_POSIX_CHOWN_RESTRICTED	indicates if use of chown is restricted	_PC_CHOWN_RESTRICTED
_POSIX_NO_TRUNC	indicates if pathnames longer than NAME_MAX generate an error	_PC_NO_TRUNC
_POSIX_VDISABLE	if defined, terminal special characters can be disabled with this value	_PC_VDISABLE

Figure 2.5 Limits and *name* arguments to `sysconf`, `pathconf`, and `fpathconf`.

2. The 12 *names* that contain MAX and the *name* _PC_PIPE_BUF can return either the value of the variable (a return value ≥ 0) or an indication that the value is indeterminate. An indeterminate value is indicated by returning -1 and not changing the value of `errno`.
3. The value returned for _SC_CLK_TCK is the number of clock ticks per second, for use with the return values from the `times` function (Section 8.15).

4. The value returned for _SC_VERSION indicates the four-digit year and two-digit month of the standard. This can be either 198808L or 199009L, or some other value for a later version of the standard.
5. The value returned for _SC_XOPEN_VERSION indicates the version of the XPG that the system complies with. Its current value is 3.
6. The two values _SC_JOB_CONTROL and _SC_SAVED_IDS represent optional features. If sysconf returns –1 without changing errno for either of these, the feature isn't supported. Both of these features can also be determined at compile time from the <unistd.h> header.
7. _PC_CHOWN_RESTRICTED and _PC_NO_TRUNC, return –1 without changing errno if the feature is not supported for the specified *pathname* or *filedes*.
8. _PC_VDISABLE returns –1 without changing errno if the feature is not supported for the specified *pathname* or *filedes*. If the feature is supported, the return value is the character value to be used to disable the special terminal input characters (Figure 11.6).

Example

Program 2.1 prints all these limits, handling the case where a limit is not defined.

```
#include    <errno.h>
#include    "ourhdr.h"

static void pr_sysconf(char *, int);
static void pr_pathconf(char *, char *, int);

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <dirname>");

    pr_sysconf("ARG_MAX              =", _SC_ARG_MAX);
    pr_sysconf("CHILD_MAX            =", _SC_CHILD_MAX);
    pr_sysconf("clock ticks/second =", _SC_CLK_TCK);
    pr_sysconf("NGROUPS_MAX          =", _SC_NGROUPS_MAX);
    pr_sysconf("OPEN_MAX             =", _SC_OPEN_MAX);
#ifdef  _SC_STREAM_MAX
    pr_sysconf("STREAM_MAX           =", _SC_STREAM_MAX);
#endif
#ifdef  _SC_TZNAME_MAX
    pr_sysconf("TZNAME_MAX           =", _SC_TZNAME_MAX);
#endif
    pr_sysconf("_POSIX_JOB_CONTROL =", _SC_JOB_CONTROL);
    pr_sysconf("_POSIX_SAVED_IDS   =", _SC_SAVED_IDS);
    pr_sysconf("_POSIX_VERSION     =", _SC_VERSION);
```

```
    pr_pathconf("MAX_CANON        =", "/dev/tty", _PC_MAX_CANON);
    pr_pathconf("MAX_INPUT        =", "/dev/tty", _PC_MAX_INPUT);
    pr_pathconf("_POSIX_VDISABLE =", "/dev/tty", _PC_VDISABLE);
    pr_pathconf("LINK_MAX         =", argv[1], _PC_LINK_MAX);
    pr_pathconf("NAME_MAX         =", argv[1], _PC_NAME_MAX);
    pr_pathconf("PATH_MAX         =", argv[1], _PC_PATH_MAX);
    pr_pathconf("PIPE_BUF         =", argv[1], _PC_PIPE_BUF);
    pr_pathconf("_POSIX_NO_TRUNC =", argv[1], _PC_NO_TRUNC);
    pr_pathconf("_POSIX_CHOWN_RESTRICTED =",
                            argv[1], _PC_CHOWN_RESTRICTED);
    exit(0);
}

static void
pr_sysconf(char *mesg, int name)
{
    long    val;

    fputs(mesg, stdout);
    errno = 0;
    if ( (val = sysconf(name)) < 0) {
        if (errno != 0)
            err_sys("sysconf error");
        fputs(" (not defined)\n", stdout);
    } else
        printf(" %ld\n", val);
}

static void
pr_pathconf(char *mesg, char *path, int name)
{
    long    val;

    fputs(mesg, stdout);
    errno = 0;
    if ( (val = pathconf(path, name)) < 0) {
        if (errno != 0)
            err_sys("pathconf error, path = %s", path);
        fputs(" (no limit)\n", stdout);
    } else
        printf(" %ld\n", val);
}
```

Program 2.1 Print all possible sysconf and pathconf values.

We have conditionally included two constants that were added to POSIX.1 but were not part of the IEEE Std 1003.1–1988 version of the standard. Figure 2.6 shows sample output from Program 2.1 for some different systems. The entries "not def" mean the constant is not defined. We'll see in Section 4.14 that the SVR4 S5 filesystem is the

Limit	SunOS 4.1.1	SVR4		4.3+BSD
		S5 filesys	UFS filesys	
ARG_MAX	1048576	5120	5120	20480
CHILD_MAX	133	30	30	40
clock ticks/second	60	100	100	60
NGROUPS_MAX	16	16	16	16
OPEN_MAX	64	64	64	64
_POSIX_JOB_CONTROL	1	1	1	1
_POSIX_SAVED_IDS	1	1	1	not def
_POSIX_VERSION	198808	198808	198808	198808
MAX_CANON	256	256	256	255
MAX_INPUT	256	512	512	255
_POSIX_VDISABLE	0	0	0	255
LINK_MAX	32767	1000	1000	32767
NAME_MAX	255	14	255	255
PATH_MAX	1024	1024	1024	1024
PIPE_BUF	4096	5120	5120	512
_POSIX_NO_TRUNC	1	not def	1	1
_POSIX_CHOWN_RESTRICTED	1	not def	not def	1

Figure 2.6 Examples of configuration limits.

traditional System V filesystem, that dates back to Version 7. UFS is the SVR4 implementation of the Berkeley fast filesystem. □

2.5.5 FIPS 151-1 Requirements

The FIPS 151-1 standard that we mentioned in Section 2.2.4 tightens the POSIX.1 standard by requiring the following features.

- The following POSIX.1 optional features are required: `_POSIX_JOB_CONTROL`, `_POSIX_SAVED_IDS`, `_POSIX_NO_TRUNC`, `_POSIX_CHOWN_RESTRICTED`, and `_POSIX_VDISABLE`.
- The minimum value of `NGROUPS_MAX` is 8.
- The group ID of a newly created file or directory must be set to the group ID of the directory in which the file is created. (We describe this feature in Section 4.6.)
- If a `read` or `write` is interrupted by a signal that is caught after some data has been transferred, the function must return the number of bytes that have been transferred. (We discuss interrupted system calls in Section 10.5.)
- A login shell must define the environment variables `HOME` and `LOGNAME`.

Since the U.S. government purchases many computer systems, most vendors of POSIX systems will support these added FIPS requirements.

Constant name	Compile-time Header	Compile-time Required?	Run-time name	Minimum value
ARG_MAX	<limits.h>	*optional*	_SC_ARG_MAX	_POSIX_ARG_MAX = 4096
CHAR_BIT	<limits.h>	required		8
CHAR_MAX	<limits.h>	required		127
CHAR_MIN	<limits.h>	required		0
CHILD_MAX	<limits.h>	*optional*	_SC_CHILD_MAX	_POSIX_CHILD_MAX = 6
clock ticks/second			_SC_CLK_TCK	
FOPEN_MAX	<stdio.h>	required		8
INT_MAX	<limits.h>	required		32,767
INT_MIN	<limits.h>	required		–32,768
LINK_MAX	<limits.h>	*optional*	_PC_LINK_MAX	_POSIX_LINK_MAX = 8
LONG_MAX	<limits.h>	required		2,147,483,647
LONG_MIN	<limits.h>	required		–2,147,483,648
MAX_CANON	<limits.h>	*optional*	_PC_MAX_CANON	_POSIX_MAX_CANON = 255
MAX_INPUT	<limits.h>	*optional*	_PC_MAX_INPUT	_POSIX_MAX_INPUT = 255
MB_LEN_MAX	<limits.h>	required		
NAME_MAX	<limits.h>	*optional*	_PC_NAME_MAX	_POSIX_NAME_MAX = 14
NGROUPS_MAX	<limits.h>	required	_SC_NGROUPS_MAX	_POSIX_NGROUPS_MAX = 0
NL_ARGMAX	<limits.h>	required		9
NL_LANGMAX	<limits.h>	required		14
NL_MSGMAX	<limits.h>	required		32,767
NL_NMAX	<limits.h>	required		
NL_SETMAX	<limits.h>	required		255
NL_TEXTMAX	<limits.h>	required		255
NZERO	<limits.h>	required		20
OPEN_MAX	<limits.h>	*optional*	_SC_OPEN_MAX	_POSIX_OPEN_MAX = 16
PASS_MAX	<limits.h>	*optional*	_SC_PASS_MAX	8
PATH_MAX	<limits.h>	*optional*	_PC_PATH_MAX	_POSIX_PATH_MAX = 255
PIPE_BUF	<limits.h>	*optional*	_PC_PIPE_BUF	_POSIX_PIPE_BUF = 512
SCHAR_MAX	<limits.h>	required		127
SCHAR_MIN	<limits.h>	required		–127
SHRT_MAX	<limits.h>	required		32,767
SHRT_MIN	<limits.h>	required		–32,768
SSIZE_MAX	<limits.h>	required		_POSIX_SSIZE_MAX = 32,767
STREAM_MAX	<limits.h>	*optional*	_SC_STREAM_MAX	_POSIX_STREAM_MAX = 8
TMP_MAX	<stdio.h>	required		10,000
TZNAME_MAX	<limits.h>	*optional*	_SC_TZNAME_MAX	_POSIX_TZNAME_MAX = 3
UCHAR_MAX	<limits.h>	required		255
UINT_MAX	<limits.h>	required		65,535
ULONG_MAX	<limits.h>	required		4,294,967,295
USHRT_MAX	<limits.h>	required		65,535
_POSIX_CHOWN_RESTRICTED	<unistd.h>	*optional*	_PC_CHOWN_RESTRICTED	
_POSIX_JOB_CONTROL	<unistd.h>	*optional*	_SC_JOB_CONTROL	
_POSIX_NO_TRUNC	<unistd.h>	*optional*	_PC_NO_TRUNC	
_POSIX_SAVED_IDS	<unistd.h>	*optional*	_SC_SAVED_IDS	
_POSIX_VDISABLE	<unistd.h>	*optional*	_PC_VDISABLE	
_POSIX_VERSION	<unistd.h>	required	_SC_VERSION	
_XOPEN_VERSION	<unistd.h>	required	_SC_XOPEN_VERSION	

Figure 2.7 Summary of compile-time and run-time limits.

2.5.6 Summary of Limits

We've described various limits and magic constants, some of which always appear in a header, some of which may optionally appear in a header, and others that can be determined at run time. Figure 2.7 summarizes all these constants alphabetically and the various ways to obtain their values. A run-time name that begins with `_SC_` is an argument to the `sysconf` function, and a name that begins with `_PC_` is an argument to the `pathconf` and `fpathconf` functions. If the constant has a minimum value, it is listed. Note that the 13 POSIX.1 invariant minimum values from Figure 2.3 appear in the rightmost column of Figure 2.7.

2.5.7 Indeterminate Run-Time Limits

We mentioned that some of the values from Figure 2.7 can be indeterminate. These values are the ones with a third column of *optional*, whose names contain `MAX`, and the value `PIPE_BUF`. The problem we encounter is that if these aren't defined in the `<limits.h>` header we can't use them at compile time. But they might not be defined at run time if their value is indeterminate! Let's look at two specific cases—allocating storage for a pathname and determining the number of file descriptors.

Pathnames

Lots of programs need to allocate storage for a pathname. Typically the storage has been allocated at compile time and various magic numbers (few of which are the correct value) have been used by different programs as the array size: 256, 512, 1024, or the standard I/O constant `BUFSIZ`. The 4.3BSD constant `MAXPATHLEN` in the header `<sys/param.h>` is the correct value, but many 4.3BSD applications didn't use it.

POSIX.1 tries to help with the `PATH_MAX` value, but if this value is indeterminate, we're still out of luck. Program 2.2 is a function that we'll use throughout this text to allocate storage dynamically for a pathname.

If the constant `PATH_MAX` is defined in `<limits.h>` then we're all set. If it's not, we need to call `pathconf`. Since the value returned by `pathconf` is the maximum size of a relative pathname when the first argument is the working directory, we specify the root as the first argument and add 1 to the result. If `pathconf` indicates that `PATH_MAX` is indeterminate, we have to punt and just guess some value. The +1 in the call to `malloc` is for the null byte at the end (which `PATH_MAX` doesn't account for).

The correct way to handle the case of an indeterminate result depends on how the allocated space is being used. If we were allocating space for a call to `getcwd`, for example (to return the absolute pathname of the current working directory, see Section 4.22) and if the allocated space is too small, an error is returned and `errno` is set to `ERANGE`. We could then increase the allocated space by calling `realloc` (see Section 7.8 and Exercise 4.18) and try again. We could keep doing this until the call to `getcwd` succeeded.

```
#include    <errno.h>
#include    <limits.h>
#include    "ourhdr.h"

#ifdef  PATH_MAX
static int  pathmax = PATH_MAX;
#else
static int  pathmax = 0;
#endif

#define PATH_MAX_GUESS  1024    /* if PATH_MAX is indeterminate */
                        /* we're not guaranteed this is adequate */
char *
path_alloc(int *size)
                /* also return allocated size, if nonnull */
{
    char    *ptr;

    if (pathmax == 0) {     /* first time through */
        errno = 0;
        if ( (pathmax = pathconf("/", _PC_PATH_MAX)) < 0) {
            if (errno == 0)
                pathmax = PATH_MAX_GUESS;   /* it's indeterminate */
            else
                err_sys("pathconf error for _PC_PATH_MAX");
        } else
            pathmax++;      /* add one since it's relative to root */
    }

    if ( (ptr = malloc(pathmax + 1)) == NULL)
        err_sys("malloc error for pathname");

    if (size != NULL)
        *size = pathmax + 1;
    return(ptr);
}
```

Program 2.2 Dynamically allocate space for a pathname.

Maximum Number of Open Files

A common sequence of code in a daemon process (a process that runs in the background, not connected to a terminal) is one that closes all open files. Some programs have the code sequence

```
#include  <sys/param.h>

for (i = 0; i < NOFILE; i++)
    close(i);
```

assuming the constant NOFILE was defined in the <sys/param.h> header. Other programs use the constant _NFILE that some versions of <stdio.h> provide as the upper limit. Some hard code the upper limit as 20.

We would hope to use the POSIX.1 value OPEN_MAX to determine this value portably, but if the value is indeterminate we still have a problem. If we wrote

```
#include  <unistd.h>

for (i = 0; i < sysconf(_SC_OPEN_MAX); i++)
    close(i);
```

and if OPEN_MAX was indeterminate, the loop would never execute, since sysconf would return –1. Our best option in this case is just to close all descriptors up to some arbitrary limit (say 256). As with our pathname example, this is not guaranteed to work for all cases, but it's the best we can do. We show this technique in Program 2.3.

```
#include    <errno.h>
#include    <limits.h>
#include    "ourhdr.h"

#ifdef  OPEN_MAX
static int  openmax = OPEN_MAX;
#else
static int  openmax = 0;
#endif

#define OPEN_MAX_GUESS  256     /* if OPEN_MAX is indeterminate */
                        /* we're not guaranteed this is adequate */
int
open_max(void)
{
    if (openmax == 0) {     /* first time through */
        errno = 0;
        if ( (openmax = sysconf(_SC_OPEN_MAX)) < 0) {
            if (errno == 0)
                openmax = OPEN_MAX_GUESS;   /* it's indeterminate */
            else
                err_sys("sysconf error for _SC_OPEN_MAX");
        }
    }
    return(openmax);
}
```

Program 2.3 Determine the number of file descriptors.

We might be tempted to call close until we get an error return, but the error return from close (EBADF) doesn't distinguish between an invalid descriptor and a descriptor that wasn't open. If we tried this technique and descriptor 9 was not open but descriptor 10 was, we would stop on 9 and never close 10. The dup function (Section 3.12) does return a specific error when OPEN_MAX is exceeded, but duplicating a descriptor a couple of hundred times is an extreme way to determine this value.

The SVR4 and 4.3+BSD `getrlimit`(2) function (Section 7.11) and the 4.3+BSD function `getdtablesize`(2) return the maximum number of descriptors that a process can have open. Calling these two functions, however, isn't portable.

The `OPEN_MAX` value is called run-time invariant by POSIX, meaning its value should not change during the lifetime of a process. But under SVR4 and 4.3+BSD we can call the `setrlimit`(2) function (Section 7.11) to change this value from a running process. (This value can also be changed from the C Shell with the `limit` command, and from the Bourne shell and KornShell with the `ulimit` command.) If our system supports this functionality we could change Program 2.3 to call `sysconf` every time it is called, not just the first time.

2.6 Feature Test Macros

The headers define numerous POSIX.1 and XPG3 symbols, as we've described. But most implementations can add their own definitions to these headers, in addition to the POSIX.1 and XPG3 definitions. If we want to compile a program so that it depends only on the POSIX definitions and doesn't use any implementation-defined limits, we need to define the constant `_POSIX_SOURCE`. All the POSIX.1 headers use this constant to exclude any implementation-defined definitions when `_POSIX_SOURCE` is defined.

The constant `_POSIX_SOURCE`, and its corresponding constant `_XOPEN_SOURCE`, are called *feature test macros*. All feature test macros begin with an underscore. When used, they are typically defined in the `cc` command as in

```
cc -D_POSIX_SOURCE file.c
```

This causes the feature test macro to be defined before any header files are included by the C program. We can also set the first line of a source file to

```
#define _POSIX_SOURCE  1
```

if we want to use only the POSIX.1 definitions.

Another feature test macro is `__STDC__`, which is automatically defined by the C compiler if the compiler conforms to the ANSI C standard. This allows us to write C programs that compile under both ANSI C compilers and non-ANSI C compilers. For example, a header could look like

```
#ifdef __STDC__
void  *myfunc(const char *, int);
#else
void  *myfunc();
#endif
```

to take advantage of the ANSI C prototype feature, if supported. Be aware that the two consecutive underscores at the beginning and end of the name `__STDC__`, often print as one long underscore (as in the preceding sample source code).

2.7 Primitive System Data Types

Historically certain C data types have been associated with certain Unix variables. For example, the major and minor device numbers have historically been stored in a 16-bit

short integer, with 8 bits for the major device number and 8 bits for the minor device number. But many larger systems need more than 256 values for these device numbers, so a different technique is needed. (Indeed, SVR4 uses 32 bits for the device number: 14 bits for the major and 18 bits for the minor.)

The header `<sys/types.h>` defines some implementation-dependent data types, called the *primitive system data types*. More of these data types are defined in other headers also. These data types are defined in the headers with the C `typedef` facility. Most end in `_t`. Figure 2.8 lists the primitive system data types that we'll encounter in this text.

Type	Description
`caddr_t`	core address (Section 12.9)
`clock_t`	counter of clock ticks (process time) (Section 1.10)
`comp_t`	compressed clock ticks (Section 8.13)
`dev_t`	device numbers (major and minor) (Section 4.23)
`fd_set`	file descriptor sets (Section 12.5.1)
`fpos_t`	file position (Section 5.10)
`gid_t`	numeric group IDs
`ino_t`	i-node numbers (Section 4.14)
`mode_t`	file type, file creation mode (Section 4.5)
`nlink_t`	link counts for directory entries (Section 4.14)
`off_t`	file sizes and offsets (signed) (`lseek`, Section 3.6)
`pid_t`	process IDs and process group IDs (signed) (Sections 8.2 and 9.4)
`ptrdiff_t`	result of subtracting two pointers (signed)
`rlim_t`	resource limits (Section 7.11)
`sig_atomic_t`	data type that can be accessed atomically (Section 10.15)
`sigset_t`	signal set (Section 10.11)
`size_t`	sizes of objects (such as strings) (unsigned) (Section 3.7)
`ssize_t`	functions that return a count of bytes (signed) (`read`, `write`, Section 3.7)
`time_t`	counter of seconds of calendar time (Section 1.10)
`uid_t`	numeric user IDs
`wchar_t`	can represent all distinct character codes

Figure 2.8 Primitive system data types.

By defining these data types this way, we do not build into our programs implementation details that can change from one system to another. We describe what each of these data types is used for when we encounter them later in the text.

2.8 Conflicts Between Standards

All in all these different standards fit together nicely. Our main concern is any differences between the ANSI C standard and POSIX.1, since XPG3 is an older standard (and is being revised) and FIPS is a tightening of POSIX.1. There are some differences.

ANSI C defines the function `clock` to return the amount of CPU time used by the process. The value returned is a `clock_t` value. To convert this value to seconds we divide it by `CLOCKS_PER_SEC`, which is defined in the `<time.h>` header. POSIX.1 defines the function `times` that returns both the CPU time (for the caller and all its

terminated children) and the clock time. All these time values are `clock_t` values. The IEEE Std. 1003.1–1988 defined the symbol `CLK_TCK` as the number of ticks per second that these `clock_t` values were measured in. With the 1990 POSIX.1 standard the symbol `CLK_TCK` is declared obsolete and we should use the `sysconf` function instead, to obtain the number of clock ticks per second for use with the return values from the `times` function. What we have is the same term, clock ticks per second, defined differently by ANSI C and POSIX.1. Both standards also use the same data type (`clock_t`) to hold these different values. The difference can be seen in SVR4, where `clock` returns microseconds (hence `CLOCKS_PER_SEC` is 1 million) while `CLK_TCK` is usually 50, 60, or 100, depending on the CPU type.

Another area of potential conflict is when the ANSI C standard specifies a function, but doesn't specify it as strongly as POSIX.1. This is the case for functions that require a different implementation in a POSIX environment (with multiple processes) than an ANSI C environment (where very little can be assumed about the host operating system). Nevertheless, many POSIX-compliant systems implement the ANSI C function, for compatibility. The `signal` function is an example. If we unknowingly use the `signal` function provided by SVR4 (hoping to write portable code that can be run in ANSI C environments and under older Unix systems), it'll provide different semantics than the POSIX.1 `sigaction` function. We'll have more to say about the `signal` function in Chapter 10.

2.9 Summary

Much has been happening over the past few years with the standardization of the different versions of Unix. We've described the three dominant standards—ANSI C, POSIX, and XPG3—and the effect of these standards on the two implementations that we'll examine in this text: SVR4 and 4.3+BSD. These standards try to define certain parameters that can change with each implementation, but we've seen that these limits are imperfect. We'll encounter all these limits and magic constants as we proceed through the text.

The bibliography lists how one can order copies of these standards that we've discussed.

Exercises

2.1 We mentioned in Section 2.7 that some of the primitive system data types are defined in more than one header. For example, `size_t` is defined in six different headers. Since a program could `#include` all six of these different headers and since ANSI C does not allow multiple `typedefs` for the same name, how must the headers be written?

2.2 Examine your system's headers and list the actual data types used to implement the primitive system data types.

3

File I/O

3.1 Introduction

We'll start our discussion of the Unix system by describing the functions available for file I/O—open a file, read a file, write a file, and so on. Most Unix file I/O can be performed using only five functions: `open`, `read`, `write`, `lseek`, and `close`. We then examine the effect of different buffer sizes on the `read` and `write` functions.

The functions described in this chapter are often referred to as *unbuffered I/O* (in contrast to the standard I/O routines, which we describe in Chapter 5). The term unbuffered refers to the fact that each `read` or `write` invokes a system call in the kernel. These unbuffered I/O functions are not part of ANSI C, but are part of POSIX.1 and XPG3.

Whenever we describe the sharing of resources between multiple processes, the concept of an atomic operation becomes important. We examine this concept with regard to file I/O and the arguments to the `open` function. This leads to a discussion of how files are shared between multiple processes and the kernel data structures involved. Once we've described these features, we describe the `dup`, `fcntl`, and `ioctl` functions.

3.2 File Descriptors

To the kernel all open files are referred to by file descriptors. A file descriptor is a nonnegative integer. When we open an existing file or create a new file, the kernel returns a file descriptor to the process. When we want to read or write a file, we identify the file with the file descriptor that was returned by `open` or `creat` as an argument to either `read` or `write`.

By convention the Unix shells associate file descriptor 0 with the standard input of a process, file descriptor 1 with the standard output, and file descriptor 2 with the standard error. This is a convention employed by the Unix shells and many Unix applications—it is not a feature of the kernel. Nevertheless, many Unix applications would break if these associations weren't followed.

The magic numbers 0, 1, and 2 should be replaced in POSIX.1 applications with the symbolic constants `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`. These are defined in the `<unistd.h>` header.

File descriptors range from 0 through `OPEN_MAX`. (Recall Figure 2.7.) Older versions of Unix had an upper limit of 19 (allowing a maximum of 20 open files per process) but this was increased to 63 by many systems.

> With SVR4 and 4.3+BSD the limit is essentially infinite, bounded by the amount of memory on the system, the size of an integer, and any hard and soft limits configured by the system administrator.

3.3 open Function

A file is opened or created by calling the open function.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int oflag, ... /* , mode_t mode */ );
```

Returns: file descriptor if OK, –1 on error

We show the third argument as `...`, which is the ANSI C way to specify that the number and types of the remaining arguments may vary. For this function the third argument is only used when a new file is being created, as we describe later. We show this argument as a comment in the prototype.

The *pathname* is the name of the file to open or create. There are a multitude of options for this function, which are specified by the *oflag* argument. This argument is formed by OR'ing together one or more of the following constants (from the `<fcntl.h>` header).

`O_RDONLY`	Open for reading only.
`O_WRONLY`	Open for writing only.
`O_RDWR`	Open for reading and writing.

> Most implementations define `O_RDONLY` as 0, `O_WRONLY` as 1, and `O_RDWR` as 2, for compatibility with older programs.

One and only one of these three constants must be specified. The following constants are optional:

`O_APPEND` Append to the end of file on each write. We describe this option in detail in Section 3.11.

`O_CREAT` Create the file if it doesn't exist. This option requires a third argument to the open function, the *mode*, which specifies the access permission bits of the new file. (When we describe a file's access permission bits in Section 4.5, we'll see how to specify the *mode*, and how it can be modified by the `umask` value of a process.)

`O_EXCL` Generate an error if `O_CREAT` is also specified and the file already exists. This test for whether the file already exists and the creation of the file if it doesn't exist is an atomic operation. We describe atomic operations in more detail in Section 3.11.

`O_TRUNC` If the file exists, and if the file is successfully opened for either write-only or read–write, truncate its length to 0.

`O_NOCTTY` If the *pathname* refers to a terminal device, do not allocate the device as the controlling terminal for this process. We talk about controlling terminals in Section 9.6.

`O_NONBLOCK` If the *pathname* refers to a FIFO, a block special file, or a character special file, this option sets the nonblocking mode for both the opening of the file and for subsequent I/O. We describe this mode in Section 12.2.

> In earlier releases of System V the `O_NDELAY` (no delay) flag was introduced. This option is similar to the `O_NONBLOCK` (nonblocking) option, but an ambiguity was introduced in the return value from a read operation. The no-delay option causes a read to return 0 if there is no data to be read from a pipe, FIFO, or device, but this conflicts with a return value of 0 indicating an end of file. SVR4 still supports the no-delay option, with the old semantics, but new applications should use the nonblocking option instead.

`O_SYNC` Have each `write` wait for physical I/O to complete. We use this option in Section 3.13.

> The `O_SYNC` option is not part of POSIX.1. It is supported by SVR4.

The file descriptor returned by open is guaranteed to be the lowest numbered unused descriptor. This is used by some applications to open a new file on standard input, standard output, or standard error. For example, an application might close standard output (normally file descriptor 1) and then open some other file, knowing that it will be opened on file descriptor 1. We'll see a better way to guarantee that a file is open on a given descriptor in Section 3.12 with the dup2 function.

Filename and Pathname Truncation

What happens if `NAME_MAX` is 14 and we try to create a new file in the current directory with a filename containing 15 characters? Traditionally, earlier releases of System V allowed this to happen, silently truncating the filename beyond the 14th character, while Berkeley-derived systems return the error `ENAMETOOLONG`. This problem does

not apply just to the creation of new files. If NAME_MAX is 14 and a file exists whose name is exactly 14 characters, then any function that accepts a *pathname* argument (open, stat, etc.) has to deal with this problem.

With POSIX.1 the constant _POSIX_NO_TRUNC determines whether long filenames and long pathnames are truncated or whether an error is returned. As we saw in Chapter 2, this value can vary on a per-filesystem basis.

> FIPS 151-1 requires that an error be returned.
>
> SVR4 does not generate an error for a traditional System V filesystem (S5). (See Figure 2.6.) For a Berkeley-style filesystem (UFS), however, SVR4 does generate an error.
>
> 4.3+BSD always returns an error.

If _POSIX_NO_TRUNC is in effect, then the error ENAMETOOLONG is returned if either the entire pathname exceeds PATH_MAX, or if any filename component of the pathname exceeds NAME_MAX.

3.4 creat Function

A new file can also be created by

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat(const char *pathname, mode_t mode);
```

Returns: file descriptor opened for write-only if OK, −1 on error

Note that this function is equivalent to

```
open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

> In earlier versions of Unix the second argument to open could only be 0, 1, or 2. There was no way to open a file that didn't already exist. Therefore a separate system call, creat, was needed to create new files. With the O_CREAT and O_TRUNC options now provided by open, a separate creat function is no longer needed.

We'll show how to specify *mode* in Section 4.5 when we describe a file's access permissions in detail.

One deficiency with creat is that the file is opened only for writing. Before the new version of open was provided, if we were creating a temporary file that we wanted to write and then read back, we had to call creat, close, and then open. A better way is to use the new open function, as in

```
open(pathname, O_RDWR | O_CREAT | O_TRUNC, mode);
```

3.5 close Function

An open file is closed by

```
#include <unistd.h>

int close(int filedes);
```

Returns: 0 if OK, –1 on error

Closing a file also releases any record locks that the process may have on the file. We'll discuss this in Section 12.3.

When a process terminates, all open files are automatically closed by the kernel. Many programs take advantage of this fact and don't explicitly `close` open files. See Program 1.2, for example.

3.6 lseek Function

Every open file has an associated "current file offset." This is a nonnegative integer that measures the number of bytes from the beginning of the file. (We describe some exceptions to the "nonnegative" qualifier later in this section.) Read and write operations normally start at the current file offset and cause the offset to be incremented by the number of bytes read or written. By default, this offset is initialized to 0 when a file is opened, unless the `O_APPEND` option is specified.

An open file can be explicitly positioned by calling `lseek`.

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int filedes, off_t offset, int whence);
```

Returns: new file offset if OK, –1 on error

The interpretation of the *offset* depends on the value of the *whence* argument.

- If *whence* is `SEEK_SET`, the file's offset is set to *offset* bytes from the beginning of the file.
- If *whence* is `SEEK_CUR`, the file's offset is set to its current value plus the *offset*. The *offset* can be positive or negative.
- If *whence* is `SEEK_END`, the file's offset is set to the size of the file plus the *offset*. The *offset* can be positive or negative.

Since a successful call to `lseek` returns the new file offset, we can seek zero bytes from the current position to determine the current offset.

```
off_t    currpos;

currpos = lseek(fd, 0, SEEK_CUR);
```

This technique can also be used to determine if the referenced file is capable of seeking: if the file descriptor refers to a pipe or FIFO, `lseek` returns –1 and sets `errno` to `EPIPE`.

> The three symbolic constants, `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`, were introduced with System V. Before System V *whence* was specified as 0 (absolute), 1 (relative to current offset), or 2 (relative to end of file). Much software still exists with these numbers hard coded.
>
> The character `l` in the name `lseek` means "long integer." Before the introduction of the `off_t` data type, the *offset* argument and the return value were long integers. `lseek` was introduced with Version 7 when long integers were added to C. (Similar functionality was provided in Version 6 by the functions `seek` and `tell`.)

Example

Program 3.1 tests its standard input to see if it is capable of seeking.

```
#include    <sys/types.h>
#include    "ourhdr.h"

int
main(void)
{
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("cannot seek\n");
    else
        printf("seek OK\n");
    exit(0);
}
```

Program 3.1 Test if standard input is capable of seeking.

If we invoke this program interactively, we get

```
$ a.out < /etc/motd
seek OK
$ cat < /etc/motd | a.out
cannot seek
$ a.out < /var/spool/cron/FIFO
cannot seek
```

□

Normally a file's current offset must be a nonnegative integer. It is possible, however, that certain devices could allow negative offsets. But for regular files the offset

must be nonnegative. Since negative offsets are possible, we should be careful to compare the return value from lseek as being equal to or not equal to –1 and not test if it's less than 0.

> The /dev/kmem device on SVR4 for the 80386 supports negative offsets.
>
> Since the offset (off_t) is a signed data type (Figure 2.8), we lose a factor of 2 in the maximum file size. For example, if off_t is a 32-bit integer, the maximum file size is 2^{31} bytes.

lseek only records the current file offset within the kernel—it does not cause any I/O to take place. This offset is then used by the next read or write operation.

The file's offset can be greater than the file's current size, in which case the next write to the file will extend the file. This is referred to as creating a hole in a file and is allowed. Any bytes in a file that have not been written are read back as 0.

Example

Program 3.2 creates a file with a hole in it.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>
#include    "ourhdr.h"

char    buf1[] = "abcdefghij";
char    buf2[] = "ABCDEFGHIJ";

int
main(void)
{
    int     fd;

    if ( (fd = creat("file.hole", FILE_MODE)) < 0)
        err_sys("creat error");

    if (write(fd, buf1, 10) != 10)
        err_sys("buf1 write error");
    /* offset now = 10 */

    if (lseek(fd, 40, SEEK_SET) == -1)
        err_sys("lseek error");
    /* offset now = 40 */

    if (write(fd, buf2, 10) != 10)
        err_sys("buf2 write error");
    /* offset now = 50 */

    exit(0);
}
```

Program 3.2 Create a file with a hole in it.

Running this program gives us

```
$ a.out
$ ls -l file.hole                          check its size
-rw-r--r--  1 stevens     50 Jul 31 05:50 file.hole
$ od -c file.hole                          let's look at the actual contents
0000000   a   b   c   d   e   f   g   h   i   j \0 \0 \0 \0 \0 \0
0000020  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000040  \0 \0 \0 \0 \0 \0 \0 \0   A   B   C   D   E   F   G   H
0000060   I   J
0000062
```

We use the od(1) command to look at the actual contents of the file. The -c flag tells it to print the contents as characters. We can see that the 30 unwritten bytes in the middle are read back as zero. The seven-digit number at the beginning of each line is the byte offset in octal. In this example we call the `write` function (Section 3.8). We'll have more to say about files with holes in Section 4.12. □

3.7 `read` Function

Data is read from an open file with the `read` function.

```
#include <unistd.h>

ssize_t read(int filedes, void *buff, size_t nbytes);

                                Returns: number of bytes read, 0 if end of file, –1 on error
```

If the `read` is successful, the number of bytes read is returned. If the end of file is encountered, 0 is returned.

There are several cases in which the number of bytes actually read is less than the amount requested:

- When reading from a regular file, if the end of file is reached before the requested number of bytes has been read. For example, if there are 30 bytes remaining until the end of file and we try to read 100 bytes, `read` returns 30. The next time we call `read` it will return 0 (end of file).
- When reading from a terminal device, normally up to one line is read at a time (we'll see how to change this in Chapter 11).
- When reading from a network, buffering within the network may cause less than the requested amount to be returned.
- Some record-oriented devices, such as a magnetic tape, return up to a single record at a time.

The read operation starts at the file's current offset. Before a successful return, the offset is incremented by the number of bytes actually read.

POSIX.1 changed the prototype for this function in several ways. The classic definition is

```
int read(int filedes, char *buff, unsigned nbytes);
```

First, the second argument was changed from a `char *` to a `void *` to be consistent with ANSI C: the type `void *` is used for generic pointers. Next, the return value must be a signed integer (`ssize_t`) to return either a positive byte count, 0 (for end of file), or –1 (for an error). Finally, the third argument historically has been an unsigned integer, to allow a 16-bit implementation to read or write up to 65534 bytes at a time. With the 1990 POSIX.1 standard the new primitive system data type `ssize_t` was introduced to provide the signed return value, and the unsigned `size_t` was used for the third argument. (Recall the `SSIZE_MAX` constant from Figure 2.7.)

3.8 `write` Function

Data is written to an open file with the `write` function.

```
#include <unistd.h>

ssize_t write(int filedes, const void *buff, size_t nbytes);
```

Returns: number of bytes written if OK, –1 on error

The return value is usually equal to the *nbytes* argument, otherwise an error has occurred. A common cause for a `write` error is either filling up a disk or exceeding the file size limit for a given process (Section 7.11 and Exercise 10.11).

For a regular file, the write starts at the file's current offset. If the `O_APPEND` option was specified in the open, the file's offset is set to the current end of file before each write operation. After a successful write, the file's offset is incremented by the number of bytes actually written.

3.9 I/O Efficiency

Using only the `read` and `write` functions, Program 3.3 copies a file. The following caveats apply to Program 3.3:

- It reads from standard input and writes to standard output. This assumes that these have been set up by the shell before this program is executed. Indeed, all normal Unix shells provide a way to open a file for reading on standard input and to create (or rewrite) a file on standard output. This prevents the program from having to open the input and output files.
- Many applications assume that standard input is file descriptor 0 and standard output is file descriptor 1. In this example we use the two defined names `STDIN_FILENO` and `STDOUT_FILENO` from `<unistd.h>`.

```
#include    "ourhdr.h"

#define BUFFSIZE    8192

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    while ( (n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

Program 3.3 Copy standard input to standard output.

- The program doesn't `close` the input file or output file. Instead it uses the fact that whenever a process terminates, Unix closes all open file descriptors.
- This example works for both text file and binary files, since there is no difference between the two to the Unix kernel.

One question we haven't answered, however, is how we chose the `BUFFSIZE` value. Before answering that, let's run the program using different values for `BUFFSIZE`. In Figure 3.1 we show the results for reading a 1,468,802 byte file, using 18 different buffer sizes.

The file was read using Program 3.3 with standard output redirected to `/dev/null`. The filesystem used for this test was a Berkeley fast filesystem with 8192-byte blocks. (The `st_blksize`, which we describe in Section 4.12, is 8192.) This accounts for the minimum in the system time occurring at a `BUFFSIZE` of 8192. Increasing the buffer size beyond this has no effect.

We'll return to this timing example later in the text. In Section 3.13 we show the effect of synchronous writes, and in Section 5.8 we compare these unbuffered I/O times with the standard I/O library.

3.10 File Sharing

Unix supports the sharing of open files between different processes. Before describing the dup function, we need to describe this sharing. To do this we'll examine the data structures used by the kernel for all I/O.

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	#loops
1	23.8	397.9	423.4	1468802
2	12.3	202.0	215.2	734401
4	6.1	100.6	107.2	367201
8	3.0	50.7	54.0	183601
16	1.5	25.3	27.0	91801
32	0.7	12.8	13.7	45901
64	0.3	6.6	7.0	22950
128	0.2	3.3	3.6	11475
256	0.1	1.8	1.9	5738
512	0.0	1.0	1.1	2869
1024	0.0	0.6	0.6	1435
2048	0.0	0.4	0.4	718
4096	0.0	0.4	0.4	359
8192	0.0	0.3	0.3	180
16384	0.0	0.3	0.3	90
32768	0.0	0.3	0.3	45
65536	0.0	0.3	0.3	23
131072	0.0	0.3	0.3	12

Figure 3.1 Timing results for reading with different buffer sizes.

Three data structures are used by the kernel, and the relationships among them determines the effect one process has on another with regard to file sharing.

1. Every process has an entry in the process table. Within each process table entry is a table of open file descriptors, which we can think of as a vector, with one entry per descriptor. Associated with each file descriptor are
   (a) the file descriptor flags,
   (b) a pointer to a file table entry.

2. The kernel maintains a file table for all open files. Each file table entry contains
   (a) the file status flags for the file (read, write, append, sync, nonblocking, etc.),
   (b) the current file offset,
   (c) a pointer to the v-node table entry for the file.

3. Each open file (or device) has a v-node structure. The v-node contains information about the type of file and pointers to functions that operate on the file. For most files the v-node also contains the i-node for the file. This information is read from disk when the file is opened, so that all the pertinent information about the file is readily available. For example, the i-node contains the owner of the file, the size of the file, the device the file is located on, pointers to where the actual data blocks for the file are located on disk, and so on. (We talk more about i-nodes in Section 4.14 when we describe the typical Unix filesystem in more detail.)

We're ignoring some implementation details that don't affect our discussion. For example, the table of open file descriptors is usually in the user area and not the process table. In SVR4 this data structure is a linked list of structures. The file table can be implemented in numerous ways—it need not be an array of file table entries. In 4.3+BSD the v-node contains the actual i-node, as we've shown. SVR4 stores the v-node in the i-node for most of its filesystem types. These implementation details don't affect our discussion of file sharing.

Figure 3.2 shows a pictorial arrangement of these three tables for a single process that has two different files open—one file is open on standard input (file descriptor 0) and the other is open on standard output (file descriptor 1).

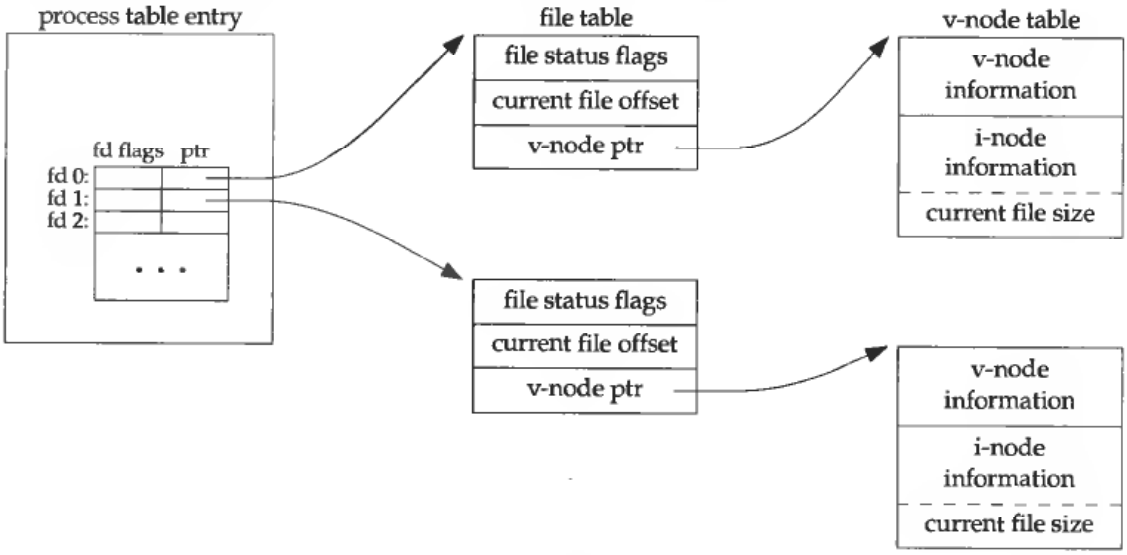


Figure 3.2 Kernel data structures for open files.

The arrangement of these three tables has existed since the early versions of Unix [Thompson 1978], and this arrangement is critical to the way files are shared between different processes. We'll return to this figure in later chapters, as we describe additional ways that files are shared.

> The v-node structure is a recent addition. It evolved when support was provided for multiple filesystem types on a given system. This work was done independently by Peter Weinberger (Bell Laboratories) and Bill Joy (Sun Microsystems). Sun called this the Virtual File System and called the filesystem independent portion of the i-node the v-node [Kleiman 1986]. The v-node propagated through various vendor implementations as support for Sun's Network File System (NFS) was added. The first release from Berkeley to provide v-nodes was the 4.3BSD Reno release, when NFS was added.
>
> In SVR4 the v-node replaced the filesystem independent i-node of SVR3.

If two independent processes have the same file open we could have the arrangement shown in Figure 3.3. We assume here that the first process has the file open on descriptor 3, and the second process has that same file open on descriptor 4. Each process that opens the file gets its own file table entry, but only a single v-node table entry

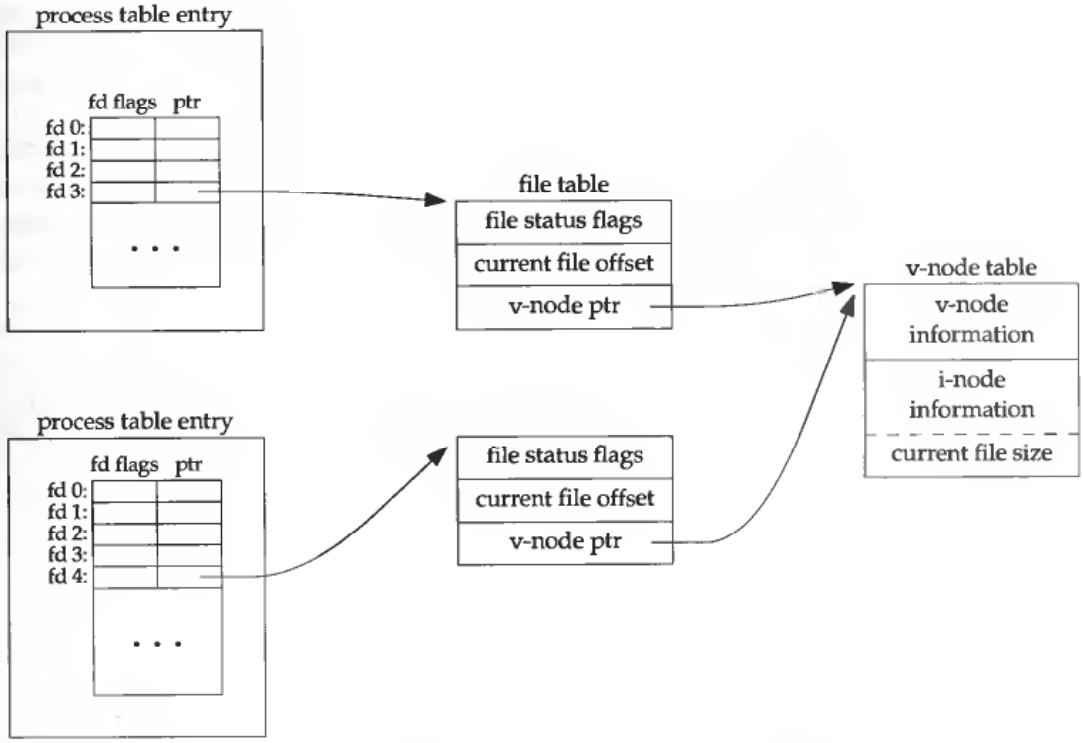


Figure 3.3 Two independent processes with the same file open.

is required for a given file. One reason each process gets its own file table entry is so that each process has its own current offset for the file.

Given these data structures we now need to be more specific about what happens with certain operations that we've already described.

- After each `write` is complete, the current file offset in the file table entry is incremented by the number of bytes written. If this causes the current file offset to exceed the current file size, the current file size in the i-node table entry is set to the current file offset (e.g., the file is extended).
- If a file is opened with the `O_APPEND` flag, a corresponding flag is set in the file status flags of the file table entry. Each time a `write` is performed for a file with this append flag set, the current file offset in the file table entry is first set to the current file size from the i-node table entry. This forces every `write` to be appended to the current end of file.
- The `lseek` function only modifies the current file offset in the file table entry. No I/O takes place.
- If a file is positioned to its current end of file using `lseek`, all that happens is the current file offset in the file table entry is set to the current file size from the i-node table entry.

It is possible for more than one file descriptor entry to point to the same file table entry. We'll see this when we discuss the dup function in Section 3.12. This also happens after a `fork` when the parent and child share the same file table entry for each open descriptor (Section 8.3).

Note the difference in scope between the file descriptor flags and the file status flags. The former apply only to a single descriptor in a single process, while the latter apply to all descriptors in any process that point to the given file table entry. When we describe the `fcntl` function in Section 3.13 we'll see how to fetch and modify both the file descriptor flags and the file status flags.

Everything that we've described so far in this section works fine for multiple processes that are reading the same file. Each process has its own file table entry with its own current file offset. Unexpected results can arise, however, when multiple processes write to the same file. To see how to avoid some surprises, we need to understand the concept of atomic operations.

3.11 Atomic Operations

Appending to a File

Consider a single process that wants to append to the end of a file. Older versions of Unix didn't support the `O_APPEND` option to open, so the program was coded as

```
if (lseek(fd, 0L, 2) < 0)          /* position to EOF */
   err_sys("lseek error");
if (write(fd, buff, 100) != 100)   /* and write */
   err_sys("write error");
```

This works fine for a single process, but problems arise if multiple processes use this technique to append to the same file. (This scenario can arise if multiple instances of the same program are appending messages to a log file, for example.)

Assume two independent processes, A and B, are appending to the same file. Each have opened the file but *without* the `O_APPEND` flag. This gives us the same picture as Figure 3.3. Each process has its own file table entry, but they share a single v-node table entry. Assume process A does the `lseek` and this sets the current offset for the file for process A to byte offset 1500 (the current end of file). Then the kernel switches processes and B continues running. It then does the `lseek`, which sets the current offset for the file for process B to byte offset 1500 also (the current end of file). Then B calls `write`, which increments B's current file offset for the file to 1600. Since the file's size has been extended, the kernel also updates the current file size in the v-node to 1600. Then the kernel switches processes and A resumes. When A calls `write`, the data is written starting at the current file offset for A, which is byte offset 1500. This overwrites the data that B wrote to the file.

The problem here is that our logical operation of "position to the end of file and write" requires two separate function calls (as we've shown it). The solution is to have the positioning to the current end of file and the write be an atomic operation with

regard to other processes. Any operation that requires more than one function call cannot be atomic, as there is always the possibility that the kernel can temporarily suspend the process between the two function calls (as we assumed previously).

Unix provides an atomic way to do this operation if we set the `O_APPEND` flag when a file is opened. As we described in the previous section, this causes the kernel to position the file to its current end of file before each `write`. We no longer have to call `lseek` before each `write`.

Creating a File

We saw another example of an atomic operation when we described the `O_CREAT` and `O_EXCL` options for the open function. When both of these options are specified, the open will fail if the file already exists. We also said that the check for the existence of the file and the creation of the file was performed as an atomic operation. If we didn't have this atomic operation we might try

```
if ( (fd = open(pathname, O_WRONLY)) < 0)
   if (errno == ENOENT) {
      if ( (fd = creat(pathname, mode)) < 0)
         err_sys("creat error");
   } else
      err_sys("open error");
```

The problem occurs if the file is created by another process between the `open` and the `creat`. If the file is created by another process between these two function calls, and if that other process writes something to the file, that data is erased when this `creat` is executed. By making the test for existence and the creation an atomic operation, this problem is avoided.

In general, the term *atomic operation* refers to an operation that is composed of multiple steps. If the operation is performed atomically, either all the steps are performed, or none is performed. It must not be possible for a subset of the steps to be performed. We'll return to the topic of atomic operations when we describe the `link` function in Section 4.15 and record locking in Section 12.3.

3.12 dup and dup2 Functions

An existing file descriptor is duplicated by either of the following functions:

```
#include <unistd.h>

int dup(int filedes);

int dup2(int filedes, int filedes2);
```

Both return: new file descriptor if OK, −1 on error

The new file descriptor returned by dup is guaranteed to be the lowest numbered available file descriptor. With dup2 we specify the value of the new descriptor with the *filedes2* argument. If *filedes2* is already open, it is first closed. If *filedes* equals *filedes2*, then dup2 returns *filedes2* without closing it.

The new file descriptor that is returned as the value of the functions shares the same file table entry as the *filedes* argument. We show this in Figure 3.4.

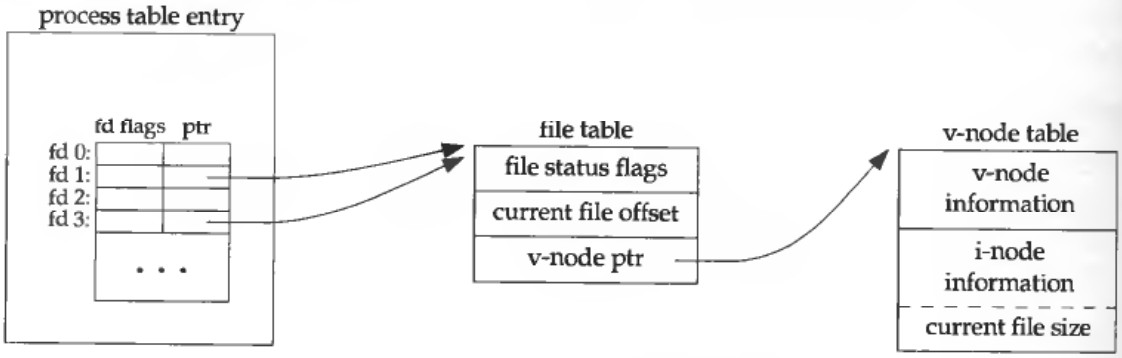


Figure 3.4 Kernel data structures after dup(1).

In this figure we're assuming that the process executes

```
newfd = dup(1);
```

when it's started. We assume the next available descriptor is 3 (which it probably is, since 0, 1, and 2 are opened by the shell). Since both descriptors point to the same file table entry they share the same file status flags (read, write, append, etc.) and the same current file offset.

Each descriptor has its own set of file descriptor flags. As we describe in the next section, the close-on-exec file descriptor flag for the new descriptor is always cleared by the dup functions.

Another way to duplicate a descriptor is with the `fcntl` function, which we describe in the next section. Indeed, the call

```
dup(filedes);
```

is equivalent to

```
fcntl(filedes, F_DUPFD, 0);
```

and the call

```
dup2(filedes, filedes2);
```

is equivalent to

```
close(filedes2);
fcntl(filedes, F_DUPFD, filedes2);
```

In this last case, the dup2 is not exactly the same as a `close` followed by an `fcntl`.

The differences are

1. dup2 is an atomic operation, while the alternate form involves two function calls. It is possible in the latter case to have a signal catcher called between the `close` and `fcntl` that could modify the file descriptors. (We describe signals in Chapter 10.)
2. There are some `errno` differences between dup2 and `fcntl`.

The dup2 system call originated with Version 7 and propagated through the BSD releases. The `fcntl` method for duplicating file descriptors appeared with System III and continued with System V. SVR3.2 picked up the dup2 function and 4.2BSD picked up the `fcntl` function and the `F_DUPFD` functionality. POSIX.1 requires both dup2 and the `F_DUPFD` feature of `fcntl`.

3.13 `fcntl` Function

The `fcntl` function can change the properties of a file that is already open.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl(int filedes, int cmd, ... /* int arg */ );
```

Returns: depends on *cmd* if OK (see following), –1 on error

In the examples we show in this section, the third argument is always an integer, corresponding to the comment in the function prototype just shown. But when we describe record locking in Section 12.3, the third argument becomes a pointer to a structure.

The `fcntl` function is used for five different purposes:

- duplicate an existing descriptor (*cmd* = `F_DUPFD`),
- get/set file descriptor flags (*cmd* = `F_GETFD` or `F_SETFD`),
- get/set file status flags (*cmd* = `F_GETFL` or `F_SETFL`),
- get/set asynchronous I/O ownership (*cmd* = `F_GETOWN` or `F_SETOWN`),
- get/set record locks (*cmd* = `F_GETLK`, `F_SETLK`, or `F_SETLKW`).

We'll now describe the first seven of these 10 *cmd* values. (We'll wait until Section 12.3 to describe the last three, which deal with record locking.) Refer to Figure 3.2 since we'll be referring to both the file descriptor flags associated with each file descriptor in the process table entry and the file status flags associated with each file table entry.

F_DUPFD Duplicate the file descriptor *filedes*. The new file descriptor is returned as the value of the function. It is the lowest numbered descriptor that is not already open, that is greater than or equal to the third argument (taken as an integer). The new descriptor shares the same file table entry as *filedes*. (Refer to Figure 3.4.) But the new descriptor has its own set of file descriptor flags and its FD_CLOEXEC file descriptor flag is cleared. (This means that the descriptor is left open across an exec, which we discuss in Chapter 8.)

F_GETFD Return the file descriptor flags for *filedes* as the value of the function. Currently only one file descriptor flag is defined: the FD_CLOEXEC flag.

F_SETFD Set the file descriptor flags for *filedes*. The new flag value is set from the third argument (taken as an integer).

> Be aware that many existing programs that deal with the file descriptor flags don't use the constant FD_CLOEXEC. Instead the programs set the flag to either 0 (don't close-on-exec, the default) or 1 (do close-on-exec).

F_GETFL Return the file status flags for *filedes* as the value of the function. We described the file status flags when we described the open function. They are listed in Figure 3.5.

File status flag	Description
O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_APPEND	append on each write
O_NONBLOCK	nonblocking mode
O_SYNC	wait for writes to complete
O_ASYNC	asynchronous I/O (4.3+BSD only)

Figure 3.5 File status flags for `fcntl`.

Unfortunately, the three access mode flags (O_RDONLY, O_WRONLY, and O_RDWR) are not separate bits that can be tested. (As we mentioned earlier, these three often have the values 0, 1, and 2, respectively, for historical reasons; also these three values are mutually exclusive—a file can have only one of the three enabled.) Therefore we must first use the O_ACCMODE mask to obtain the access mode bits and then compare the result against any of the three values.

F_SETFL Set the file status flags to the value of the third argument (taken as an integer). The only flags that can be changed are O_APPEND, O_NONBLOCK, O_SYNC, and O_ASYNC.

F_GETOWN Get the process ID or process group ID currently receiving the SIGIO and SIGURG signals. We describe these 4.3+BSD asynchronous I/O signals in Section 12.6.2.

F_SETOWN Set the process ID or process group ID to receive the SIGIO and SIGURG signals. A positive *arg* specifies a process ID. A negative *arg* implies a process group ID equal to the absolute value of *arg*.

The return value from fcntl depends on the command. All commands return –1 on an error or some other value if OK. The following three commands have special return values: F_DUPFD, F_GETFD, F_GETFL, and F_GETOWN. The first returns the new file descriptor, the next two return the corresponding flags, and the final one returns a positive process ID or a negative process group ID.

Example

Program 3.4 takes a single command-line argument that specifies a file descriptor and prints a description of the file flags for that descriptor.

```
#include    <sys/types.h>
#include    <fcntl.h>
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    int     accmode, val;

    if (argc != 2)
        err_quit("usage: a.out <descriptor#>");

    if ( (val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
        err_sys("fcntl error for fd %d", atoi(argv[1]));

    accmode = val & O_ACCMODE;
    if      (accmode == O_RDONLY)   printf("read only");
    else if (accmode == O_WRONLY)   printf("write only");
    else if (accmode == O_RDWR)     printf("read write");
    else err_dump("unknown access mode");

    if (val & O_APPEND)         printf(", append");
    if (val & O_NONBLOCK)       printf(", nonblocking");
#if !defined(_POSIX_SOURCE) && defined(O_SYNC)
    if (val & O_SYNC)           printf(", synchronous writes");
#endif
    putchar('\n');
    exit(0);
}
```

Program 3.4 Print file flags for specified descriptor.

Notice that we use the feature test macro _POSIX_SOURCE and conditionally compile the file access flags that are not part of POSIX.1. The following script shows the operation of the program, when invoked from a KornShell.

```
$ a.out 0 < /dev/tty
read only
$ a.out 1 > temp.foo
$ cat temp.foo
write only
$ a.out 2 2>>temp.foo
write only, append
$ a.out 5 5<>temp.foo
read write
```

The KornShell clause `5<>temp.foo` opens the file `temp.foo` for reading and writing on file descriptor 5. □

Example

When we modify either the file descriptor flags or the file status flags we must be careful to fetch the existing flag value, modify it as desired, and then set the new flag value. We can't just do an `F_SETFD` or an `F_SETFL`, as this could turn off flag bits that were previously set.

Program 3.5 shows a function that sets one or more of the file status flags for a descriptor.

```
#include    <fcntl.h>
#include    "ourhdr.h"

void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
    int     val;

    if ( (val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("fcntl F_GETFL error");

    val |= flags;       /* turn on flags */

    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("fcntl F_SETFL error");
}
```

Program 3.5 Turn on one or more of the file status flags for a descriptor.

If we change the middle statement to

```
val &= ~flags;      /* turn flags off */
```

we have a function named `clr_fl` that we'll use in some later examples. This statement logically ANDs the 1's-complement of `flags` with the current `val`.

If we call `set_fl` from Program 3.3 by adding the line

```
set_fl(STDOUT_FILENO, O_SYNC);
```

at the beginning of the program, we'll turn on the synchronous-write flag. This causes each `write` to wait for the data to be written to disk before returning. Normally in Unix, a `write` only queues the data for writing, and the actual I/O operation can take place sometime later. A database system is a likely candidate for using `O_SYNC`, so that it knows on return from a `write` that the data is actually on the disk, in case of a system crash.

We expect the `O_SYNC` flag to increase the clock time when the program runs. To test this we can run Program 3.3, copying a 1.5 Mbyte file from one file on disk to another and compare this with a version that does the same thing with the `O_SYNC` flag set. The results are in Figure 3.6.

Operation	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)
read time from Figure 3.1 for `BUFFSIZE` = 8192	0.0	0.3	0.3
normal Unix `write` to disk file	0.0	1.0	2.3
`write` to disk file with `O_SYNC` set	0.0	1.4	13.4

Figure 3.6 Timing results using synchronous writes (`O_SYNC`).

The three rows in Figure 3.6 were all measured with a `BUFFSIZE` of 8192. The results in Figure 3.1 were measured reading a disk file and writing to `/dev/null`, so there was no disk output. The second row in Figure 3.6 corresponds to reading a disk file and writing to another disk file. This is why the first and second rows in Figure 3.6 are different. The system time increases when we write to a disk file because the kernel now copies the data from our process and queues the data to for writing by the disk driver. The clock time increases also when we write to a disk file. When we enable synchronous writes, the system time increases slightly and the clock time increases by a factor of 6. □

With this example we see the need for `fcntl`. Our program operates on a descriptor (standard output), never knowing name of the file that was opened by the shell on that descriptor. We can't set the `O_SYNC` flag when the file is opened, since the shell opened the file. `fcntl` allows us to modify the properties of a descriptor, knowing only the descriptor for the open file. We'll see another need for `fcntl` when we describe nonblocking pipes (Section 14.2), since all we have with a pipe is a descriptor.

3.14 `ioctl` Function

The `ioctl` function has always been the catchall for I/O operations. Anything that couldn't be expressed using one of the other functions in this chapter usually ended up being specified with an `ioctl`. Terminal I/O was the biggest user of this function. (When we get to Chapter 11 we'll see that POSIX.1 has replaced the terminal I/O operations with new functions.)

```
#include <unistd.h>      /* SVR4 */
#include <sys/ioctl.h>   /* 4.3+BSD */

int ioctl(int filedes, int request, ...);
```

Returns: –1 on error, something else if OK

The `ioctl` function is not part of POSIX.1. Both SVR4 and 4.3+BSD, however, use it for many miscellaneous device operations.

The prototype that we show corresponds to SVR4. 4.3+BSD and earlier Berkeley systems declare the second argument as an `unsigned long`. This detail doesn't matter, since the second argument is always a `#defined` name from a header.

For the ANSI C prototype an ellipsis is used for the remaining arguments. Normally, however, there is just one more argument, and it's usually a pointer to a variable or a structure.

In this prototype we show only the headers required for the function itself. Normally additional device-specific headers are required. For example, the `ioctl`s for terminal I/O, beyond the basic operations specified by POSIX.1, all require the `<termios.h>` header.

What are `ioctl`s used for today? We can divide the 4.3+BSD operations into the categories shown in Figure 3.7.

Category	Constant Names	Header	Number of `ioctl`s
disk labels	`DIOxxx`	`<disklabel.h>`	10
file I/O	`FIOxxx`	`<ioctl.h>`	7
mag tape I/O	`MTIOxxx`	`<mtio.h>`	4
socket I/O	`SIOxxx`	`<ioctl.h>`	25
terminal I/O	`TIOxxx`	`<ioctl.h>`	35

Figure 3.7 4.3+BSD `ioctl` operations.

The mag tape operations allow us to write end-of-file marks on a tape, rewind a tape, space forward over a specified number of files or records, and the like. None of these operations is easily expressed in terms of the other functions in the chapter (`read`, `write`, `lseek`, etc.) so the easiest way to handle these devices has always been to access their operations using `ioctl`.

We use the `ioctl` function in Section 11.12 to fetch and set the size of a terminal's window, in Section 12.4 when we describe the streams system, and in Section 19.7 when we access the advanced features of pseudo terminals.

3.15 /dev/fd

Newer systems provide a directory named /dev/fd whose entries are files named 0, 1, 2, and so on. Opening the file /dev/fd/*n* is equivalent to duplicating descriptor *n* (assuming that descriptor *n* is open).

> The /dev/fd feature was developed by Tom Duff and appeared in the 8th Edition of the Research Unix System. It is supported by SVR4 and 4.3+BSD. It is not part of POSIX.1.

In the function call

```
fd = open("/dev/fd/0", mode);
```

most systems ignore the specified mode, while others require that it be a subset of the mode used when the referenced file (standard input in this case) was originally opened. Since the open above is equivalent to

```
fd = dup(0);
```

the descriptors 0 and fd share the same file table entry (Figure 3.4). For example, if descriptor 0 was opened read-only, we can only read on fd. Even if the system ignores the open mode, and the call

```
fd = open("/dev/fd/0", O_RDWR);
```

succeeds, we still can't write to fd.

We can also call creat with a /dev/fd pathname argument, as well as specifying O_CREAT in a call to open. This allows a program that calls creat to still work if the pathname argument is /dev/fd/1, for example.

Some systems provide the pathnames /dev/stdin, /dev/stdout, and /dev/stderr. These are equivalent to /dev/fd/0, /dev/fd/1, and /dev/fd/2.

The main use of the /dev/fd files is from the shell. It allows programs that use pathname arguments to handle standard input and standard output in the same manner as other pathnames. For example, the cat(1) program specifically looks for an input filename of - and uses this to mean standard input. The command

```
filter file2 | cat file1 - file3 | lpr
```

is an example. First cat reads file1, next its standard input (the output of the filter program on file2), then file3. If /dev/fd is supported, the special handling of - can be removed from cat, and we can enter

```
filter file2 | cat file1 /dev/fd/0 file3 | lpr
```

The special meaning of - as a command-line argument to refer to the standard input or standard output is a kludge that has crept into many programs. There are also problems if we specify - as the first file, since it looks like the start of another command-line option. /dev/fd is a step toward uniformity and cleanliness.

3.16 Summary

This chapter has described the traditional Unix I/O functions. These are often called the unbuffered I/O functions because each `read` or `write` invokes a system call into the kernel. Using only `read` and `write` we looked at the effect of different I/O sizes on the amount of time required to read a file.

Atomic operations were introduced when multiple processes append to the same file and when multiple processes create the same file. We also looked at the data structures used by the kernel to share information about open files. We'll return to these data structures later in the text.

We also described the `ioctl` and `fcntl` functions. We return to both of these functions in Chapter 12—we'll use `ioctl` with the streams I/O system, and `fcntl` is used for record locking.

Exercises

3.1 When reading or writing a disk file, are the functions described in this chapter really unbuffered? Explain.

3.2 Write your own function called `dup2` that performs the same service as the `dup2` function we described in Section 3.12, without calling the `fcntl` function. Be sure to handle errors correctly.

3.3 Assume a process executes the following three function calls:

```
fd1 = open(pathname, oflags);
fd2 = dup(fd1);
fd3 = open(pathname, oflags);
```

Draw the resulting picture, similar to Figure 3.4. Which descriptors are affected by an `fcntl` on `fd1` with a command of `F_SETFD`? Which descriptors are affected by an `fcntl` on `fd1` with a command of `F_SETFL`?

3.4 The following sequence of code has been observed in various programs:

```
dup2(fd, 0);
dup2(fd, 1);
dup2(fd, 2);
if (fd > 2)
    close(fd);
```

To see why the `if` test is needed, assume `fd` is 1 and draw a picture of what happens to the three descriptor entries and the corresponding file table entry with each call to `dup2`. Then assume `fd` is 3 and draw the same picture.

3.5 The Bourne shell and KornShell notation

digit1`>&`*digit2*

says to redirect descriptor *digit1* to the same file as descriptor *digit2*. What is the difference between the two commands

```
a.out > outfile 2>&1
```

```
a.out 2>&1 > outfile
```

(Hint: the shells process their command lines from left to right.)

3.6 If you open a file for read–write with the append flag, can you still `read` from anywhere in the file using `lseek`? Can you use `lseek` to replace existing data in the file? Write a program to verify this.

4

Files and Directories

4.1 Introduction

In the previous chapter we covered the basic functions that perform I/O. The discussion centered around I/O for regular files—opening a file, and reading or writing a file. We'll now look at additional features of the filesystem and the properties of a file. We'll start with the `stat` functions and go through each member of the `stat` structure, looking at all the attributes of a file. In this process we'll also describe each of the functions that modify these attributes (change the owner, change the permissions, etc.). We'll also look in more detail at the structure of a Unix filesystem and symbolic links. We finish this chapter with the functions that operate on directories and develop a function that descends through a directory hierarchy.

4.2 `stat`, `fstat`, and `lstat` Functions

The discussion in this chapter is centered around the three `stat` functions and the information they return.

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(const char *pathname, struct stat *buf);

int fstat(int filedes, struct stat *buf);

int lstat(const char *pathname, struct stat *buf);
```

All three return: 0 if OK, –1 on error

Given a *pathname*, the `stat` function returns a structure of information about the named file. The `fstat` function obtains information about the file that is already open on the descriptor *filedes*. The `lstat` function is similar to `stat`, but when the named file is a symbolic link, `lstat` returns information about the symbolic link, not the file referenced by the symbolic link. (We'll need `lstat` in Section 4.21 when we walk down a directory hierarchy. We describe symbolic links in more detail in Section 4.16.)

> The `lstat` function is not in the POSIX 1003.1–1990 standard, but will probably be added to 1003.1a. It is supported by SVR4 and 4.3+BSD.

The second argument is a pointer to a structure that we must supply. The function fills in the structure pointed to by *buf*. The actual definition of the structure can differ among implementations, but it could look like

```
struct stat {
  mode_t  st_mode;    /* file type & mode (permissions) */
  ino_t   st_ino;     /* i-node number (serial number) */
  dev_t   st_dev;     /* device number (filesystem) */
  dev_t   st_rdev;    /* device number for special files */
  nlink_t st_nlink;   /* number of links */
  uid_t   st_uid;     /* user ID of owner */
  gid_t   st_gid;     /* group ID of owner */
  off_t   st_size;    /* size in bytes, for regular files */
  time_t  st_atime;   /* time of last access */
  time_t  st_mtime;   /* time of last modification */
  time_t  st_ctime;   /* time of last file status change */
  long    st_blksize; /* best I/O block size */
  long    st_blocks;  /* number of 512-byte blocks allocated */
};
```

> The fields `st_rdev`, `st_blksize`, and `st_blocks` are not defined by POSIX.1. These fields are in SVR4 and 4.3+BSD.

Note that each member, other than the last two, is specified by a primitive system data type (see Section 2.7). We'll go through each member of this structure, to examine the attributes of a file.

The biggest user of the `stat` functions is probably the `ls -l` command, to learn all the information about a file.

4.3 File Types

We've talked about two different types of files so far—regular files and directories. Most files on a Unix system are either regular files or directories, but there are additional types of files:

1. Regular file. The most common type of file, which contains data of some form. There is no distinction to the Unix kernel whether this data is text or binary.

Any interpretation of the contents of a regular file is left to the application processing the file.

2. Directory file. A file that contains the names of other files and pointers to information on these files. Any process that has read permission for a directory file can read the contents of the directory, but only the kernel can write to a directory file.
3. Character special file. A type of file used for certain types of devices on a system.
4. Block special file. A type of file typically used for disk devices. All devices on a system are either character special files or block special files.
5. FIFO. A type of file used for interprocess communication between processes. It's sometimes called a named pipe. We describe FIFOs in Section 14.5.
6. Socket. A type of file used for network communication between processes. A socket can also be used for nonnetwork communication between processes on a single host. We use sockets for interprocess communication in Chapter 15.

   A file type of socket is returned only by 4.3+BSD. Although SVR4 supports sockets for interprocess communication, this is currently done through a library of socket functions, not through a file type of socket within the kernel. Future versions of SVR4 may support the socket type.

7. Symbolic link. A type of file that points to another file. We talk more about symbolic links in Section 4.16.

The type of a file is encoded in the `st_mode` member of the `stat` structure. We can determine the file type with the macros shown in Figure 4.1. The argument to each of these macros is the `st_mode` member from the `stat` structure.

Macro	Type of file
`S_ISREG()`	regular file
`S_ISDIR()`	directory file
`S_ISCHR()`	character special file
`S_ISBLK()`	block special file
`S_ISFIFO()`	pipe or FIFO
`S_ISLNK()`	symbolic link (not in POSIX.1 or SVR4)
`S_ISSOCK()`	socket (not in POSIX.1 or SVR4)

Figure 4.1 File type macros in `<sys/stat.h>`.

Example

Program 4.1 takes its command-line arguments and prints the type of file for each command-line argument.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    int         i;
    struct stat buf;
    char        *ptr;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            err_ret("lstat error");
            continue;
        }

        if      (S_ISREG(buf.st_mode))  ptr = "regular";
        else if (S_ISDIR(buf.st_mode))  ptr = "directory";
        else if (S_ISCHR(buf.st_mode))  ptr = "character special";
        else if (S_ISBLK(buf.st_mode))  ptr = "block special";
        else if (S_ISFIFO(buf.st_mode)) ptr = "fifo";
#ifdef  S_ISLNK
        else if (S_ISLNK(buf.st_mode))  ptr = "symbolic link";
#endif
#ifdef  S_ISSOCK
        else if (S_ISSOCK(buf.st_mode)) ptr = "socket";
#endif
        else                    ptr = "** unknown mode **";
        printf("%s\n", ptr);
    }
    exit(0);
}
```

Program 4.1 Print type of file for each command-line argument.

Sample output from Program 4.1 is

```
$ a.out /vmunix /etc /dev/ttya /dev/sd0a /var/spool/cron/FIFO \
> /bin /dev/printer
/vmunix: regular
/etc: directory
/dev/ttya: character special
/dev/sd0a: block special
/var/spool/cron/FIFO: fifo
/bin: symbolic link
/dev/printer: socket
```

(Here we have explicitly entered a backslash at the end of the first command line, telling the shell that we want to continue entering the command on another line. The shell

then prompts us with its secondary prompt, >, on the next line.) We have specifically used the `lstat` function instead of the `stat` function, to detect symbolic links. If we used the `stat` function, we would never see symbolic links. □

Earlier versions of Unix didn't provide the `S_ISxxx` macros. Instead we had to logically AND the `st_mode` value with the mask `S_IFMT` and then compare the result with the constants whose names are `S_IFxxx`. SVR4 and 4.3+BSD define this mask and the related constants in the file `<sys/stat.h>`. If we examine this file we'll find the `S_ISDIR` macro defined as

```
#define  S_ISDIR(mode)  (((mode) & S_IFMT) == S_IFDIR)
```

We've said that regular files are predominant, but it is interesting to see what percentage of the files on a given system are of each file type. Figure 4.2 shows the counts and percentages for a medium-sized system. This data was obtained from the program that we show in Section 4.21.

File type	Count	Percentage
regular file	30,369	91.7 %
directory	1,901	5.7
symbolic link	416	1.3
character special	373	1.1
block special	61	0.2
socket	5	0.0
FIFO	1	0.0

Figure 4.2 Counts and percentages of different file types.

4.4 Set-User-ID and Set-Group-ID

Every process has six or more IDs associated with it. These are shown in Figure 4.3.

real user ID real group ID	who we really are
effective user ID effective group ID supplementary group IDs	used for file access permission checks
saved set-user-ID saved set-group-ID	saved by exec functions

Figure 4.3 User IDs and group IDs associated with each process.

- The real user ID and real group ID identify who we really are. These two fields are taken from our entry in the password file when we log in. Normally these values don't change during a login session, although there are ways for a superuser process to change them, which we describe in Section 8.10.

- The effective user ID, effective group ID, and supplementary group IDs determine our file access permissions, as we describe in the next section. (We defined supplementary group IDs in Section 1.8.)
- The saved set-user-ID and saved set-group-ID contain copies of the effective user ID and the effective group ID when a program is executed. We describe the function of these two saved values when we describe the `setuid` function in Section 8.10.

> The saved IDs are optional with POSIX.1. An application can test for the constant `_POSIX_SAVED_IDS` at compile time, or call `sysconf` with the `_SC_SAVED_IDS` argument at run time, to see if the implementation supports this feature. SVR4 supports this feature.
>
> FIPS 151-1 requires this optional POSIX.1 feature.

Normally the effective user ID equals the real user ID, and the effective group ID equals the real group ID.

Every file has an owner and a group owner. The owner is specified by the `st_uid` member of the `stat` structure, and the group owner by the `st_gid` member.

When we execute a program file the effective user ID of the process is usually the real user ID, and the effective group ID is usually the real group ID. But the capability exists to set a special flag in the file's mode word (`st_mode`) that says "when this file is executed, set the effective user ID of the process to be the owner of the file (`st_uid`)." Similarly, another bit can be set in the file's mode word that causes the effective group ID to be the group owner of the file (`st_gid`). These two bits in the file's mode word are called the *set-user-ID* bit and the *set-group-ID* bit.

For example, if the owner of the file is the superuser and if the file's set-user-ID bit is set, then while that program file is running as a process, it has superuser privileges. This happens regardless of the real user ID of the process that executes the file. As an example, the Unix program that allows anyone to change his or her password, `passwd`(1), is a set-user-ID program. This is required so that the program can write the new password to the password file, typically either `/etc/passwd` or `/etc/shadow`, files that should be writable only by the superuser. Since a process that is running set-user-ID to some other user usually assumes extra permissions, it must be written carefully. We'll discuss these types of programs in more detail in Chapter 8.

Returning to the `stat` function, the set-user-ID bit and the set-group-ID bit are contained in the file's `st_mode` value. These two bits can be tested against the constants `S_ISUID` and `S_ISGID`.

4.5 File Access Permissions

The `st_mode` value also encodes the access permission bits for the file. When we say file we mean any of the file types that we described earlier. All the file types (directories, character special files, etc.) have permissions. Many people think only of regular files as having access permissions.

There are nine permission bits for each file, divided into three categories. These are shown in Figure 4.4.

st_mode mask	Meaning
S_IRUSR	user-read
S_IWUSR	user-write
S_IXUSR	user-execute
S_IRGRP	group-read
S_IWGRP	group-write
S_IXGRP	group-execute
S_IROTH	other-read
S_IWOTH	other-write
S_IXOTH	other-execute

Figure 4.4 The nine file access permission bits, from <sys/stat.h>.

The term user in the first three rows in Figure 4.4 refers to the owner of the file. The chmod(1) command, which is typically used to modify these nine permission bits, allows us to specify u for user (owner), g for group, and o for other. Some books refer to these three as owner, group, and world; this is confusing since the chmod command uses o to mean other, not owner. We'll use the terms user, group, and other, to be consistent with the chmod command.

The three categories in Figure 4.4—read, write, and execute—are used in various ways by different functions. We'll summarize them here, and return to them when we describe the actual functions.

- The first rule is that *whenever* we want to open any type of file by name we must have execute permission in each directory mentioned in the name, including the current directory if it is implied. This is why the execute permission bit for a directory is often called the search bit.

  For example, to open the file /usr/dict/words we need execute permission in the directory /, execute permission in the directory /usr, and execute permission in the directory /usr/dict. We then need appropriate permission for the file itself, depending on how we're trying to open it (read-only, read–write, etc.).

  If the current directory is /usr/dict then we need execute permission in the current directory to open the file words. This is an example of the current directory being implied, not specifically mentioned. It is identical to our opening the file ./words.

  Note that read permission for a directory and execute permission for a directory mean different things. Read permission lets us read the directory, obtaining a list of all the filenames in the directory. Execute permission lets us pass through the directory when it is a component of a pathname that we are trying to access (i.e., search the directory looking for a specific filename).

Another example of an implicit directory reference is if the PATH environment variable (described in Section 8.9) specifies a directory that does not have execute permission enabled. In this case the shell will never find executable files in that directory.

- The read permission for a file determines if we can open an existing file for reading: the O_RDONLY and O_RDWR flags for the open function.
- The write permission for a file determines if we can open an existing file for writing: the O_WRONLY and O_RDWR flags for the open function.
- We must have write permission for a file to specify the O_TRUNC flag in the open function.
- We cannot create a new file in a directory unless we have write permission and execute permission in the directory.
- To delete an existing file, we need write permission and execute permission in the directory containing the file. We do not need read permission or write permission for the file itself.
- Execute permission for a file must be on if we want to execute the file using any of the six exec functions (Section 8.9). The file also has to be a regular file.

The file access tests that the kernel performs each time a process opens, creates, or deletes a file depend on the owners of the file (st_uid and st_gid), the effective IDs of the process (effective user ID and effective group ID), and the supplementary group IDs of the process (if supported). The two owner IDs are properties of the file, while the two effective IDs and the supplementary group IDs are properties of the process. The tests performed by the kernel are

1. If the effective user ID of the process is 0 (the superuser), access is allowed. This gives the superuser free reign throughout the entire filesystem.
2. If the effective user ID of the process equals the owner ID of the file (i.e., the process owns the file):
   a. if the appropriate user access permission bit is set, access is allowed,
   b. else permission is denied.

   By *appropriate access permission bit* we mean if the process is opening the file for reading, the user-read bit must be on. If the process is opening the file for writing, the user-write bit must be on. If the process is executing the file, the user-execute bit must be on.
3. If the effective group ID of the process or one of the supplementary group IDs of the process equals the group ID of the file:
   a. if the appropriate group access permission bit is set, access is allowed,
   b. else permission is denied.

4. If the appropriate other access permission bit is set, access is allowed, else permission is denied.

These four steps are tried in sequence. Note that if the process owns the file (step 2) then access is granted or denied based only on the user access permissions—the group permissions are never looked at. Similarly, if the process does not own the file, but the process belongs to an appropriate group, then access is granted or denied based only on the group access permissions—the other permissions are not looked at.

4.6 Ownership of New Files and Directories

When we described the creation of a new file in Chapter 3, using either `open` or `creat`, we never said what values were assigned to the user ID and group ID of the new file. We'll see how to create a new directory in Section 4.20 when we describe the `mkdir` function. The rules for the ownership of a new directory are identical to the rules in this section for the ownership of a new file.

The user ID of a new file is set to the effective user ID of the process. POSIX.1 allows an implementation to choose one of the following options to determine the group ID of a new file.

1. The group ID of a new file can be the effective group ID of the process.
2. The group ID of a new file can be the group ID of the directory in which the file is being created.

> With SVR4 the group ID of a new file depends on whether the set-group-ID bit is set for the directory in which the file is being created. If this bit is set for the directory, the group ID of the new file is set to the group ID of the directory; otherwise the group ID of the new file is set to the effective group ID of the process.
>
> 4.3+BSD always uses the group ID of the directory as the group ID of the new file.
>
> Other systems allow the choice between these two POSIX.1 options to be done on a filesystem basis, using a special flag to the `mount`(1) command.
>
> FIPS 151-1 requires that the group ID of a new file be the group ID of the directory in which the file is created.

Using the second POSIX.1 option (inheriting the group ID of the directory) assures us that all files and directories created in that directory will have the group ID belonging to the directory. This group ownership of files and directories will then propagate down the hierarchy from that point. This is used, for example, in the `/var/spool` directory.

> As we mentioned, this option for group ownership is the default for 4.3+BSD but an option for SVR4. Under SVR4 we have to enable the set-group-ID bit. Furthermore, the SVR4 `mkdir` function has to propagate a directory's set-group-ID bit automatically (as it does, described in Section 4.20) for this to work.

4.7 access Function

As we described earlier, when accessing a file with the open function, the kernel performs its access tests based on the effective user ID and the effective group ID. There are times when a process wants to test accessibility based on the real user ID and the real group ID. One instance where this is useful is when a process is running as someone else, using either the set-user-ID or the set-group-ID feature. Even though a process might be set-user-ID to root, it could still want to verify that the real user can access a given file. The access function bases its tests on the real user ID and the real group ID. (Go through the four steps at the end of Section 4.5 and replace effective with real.)

```
#include <unistd.h>

int access(const char *pathname, int mode);
```

Returns: 0 if OK, –1 on error

The *mode* is the bitwise OR of any of the constants shown in Figure 4.5.

mode	Description
R_OK	test for read permission
W_OK	test for write permission
X_OK	test for execute permission
F_OK	test for existence of file

Figure 4.5 The *mode* constants for access function, from <unistd.h>.

Example

Program 4.2 shows the use of the access function. Here is a sample session with this program.

```
$ ls -l a.out
-rwxrwxr-x  1 stevens    105216 Jan 18 08:48 a.out
$ a.out a.out
read access OK
open for reading OK
$ ls -l /etc/uucp/Systems
-rw-r-----  1 uucp         1441 Jul 18 15:05 /etc/uucp/Systems
$ a.out /etc/uucp/Systems
access error for /etc/uucp/Systems: Permission denied
open error for /etc/uucp/Systems: Permission denied
$ su                                    become superuser
Password:                               enter superuser password
# chown uucp a.out                      change file's user ID to uucp
# chmod u+s a.out                       and turn on set-user-ID bit
```

```
#include    <sys/types.h>
#include    <fcntl.h>
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <pathname>");

    if (access(argv[1], R_OK) < 0)
        err_ret("access error for %s", argv[1]);
    else
        printf("read access OK\n");

    if (open(argv[1], O_RDONLY) < 0)
        err_ret("open error for %s", argv[1]);
    else
        printf("open for reading OK\n");

    exit(0);
}
```

Program 4.2 Example of access function.

```
# ls -l a.out                          check owner and SUID bit
-rwsrwxr-x  1 uucp       105216 Jan 18 08:48 a.out
# exit                                 go back to normal user
$ a.out /etc/uucp/Systems
access error for /etc/uucp/Systems: Permission denied
open for reading OK
```

In this example, the set-user-ID program can determine that the real user cannot normally read the file, even though the open function will succeed. □

> In the preceding example and in Chapter 8, we'll sometimes switch to become the superuser, to demonstrate how something works. If you're on a multiuser system and do not have superuser permission, you won't be able to duplicate these examples completely.

4.8 umask Function

Now that we've described the nine permission bits associated with every file, we can describe the file mode creation mask that is associated with every process.

The umask function sets the file mode creation mask for the process and returns the previous value. (This is one of the few functions that doesn't have an error return.)

```
#include <sys/types.h>
#include <sys/stat.h>

mode_t umask(mode_t cmask);
```

Returns: previous file mode creation mask

The *cmask* argument is formed as the bitwise OR of any of the nine constants from Figure 4.4: `S_IRUSR`, `S_IWUSR`, and so on.

The file mode creation mask is used whenever the process creates a new file or a new directory. (Recall Sections 3.3 and 3.4 where we described the `open` and `creat` functions. Both accepted a *mode* argument that specified the new file's access permission bits.) We describe how to create a new directory in Section 4.20. Any bits that are *on* in the file mode creation mask are turned *off* in the file's *mode*.

Example

Program 4.3 creates two files, one with a `umask` of 0 and one with a `umask` that disables all the group and other permission bits. If we run this program

```
$ umask                          first print the current file mode creation mask
02
$ a.out
$ ls -l foo bar
-rw-------  1 stevens        0 Nov 16 16:23 bar
-rw-rw-rw-  1 stevens        0 Nov 16 16:23 foo
$ umask                          see if the file mode creation mask changed
02
```

we can see how the permission bits have been set. □

Most Unix users never deal with their `umask` value. It is usually set once, on log in, by the shell's start-up file, and never changed. Nevertheless, when writing programs that create new files, if we want to assure that specific access permission bits are enabled, we must modify the `umask` value while the process is running. For example, if we want to assure that anyone can read a file, we should set the `umask` to 0. Otherwise, the `umask` value that is in effect when our process is running can cause permission bits to be turned off.

In the preceding example we use the shell's `umask` command to print the file mode creation mask before we run the program and after. This shows us that changing the file mode creation mask of a process doesn't affect the mask of its parent (often a shell). All three of the shells have a built-in `umask` command that we can use to set or print the current file mode creation mask.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>
#include    "ourhdr.h"

int
main(void)
{
    umask(0);
    if (creat("foo", S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
                                         S_IROTH | S_IWOTH) < 0)
        err_sys("creat error for foo");

    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    if (creat("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
                                         S_IROTH | S_IWOTH) < 0)
        err_sys("creat error for bar");
    exit(0);
}
```

Program 4.3 Example of umask function.

4.9 chmod and fchmod Functions

These two functions allow us to change the file access permissions for an existing file.

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);

int fchmod(int filedes, mode_t mode);
```

Both return: 0 if OK, −1 on error

The chmod function operates on the specified file while the fchmod function operates on a file that has already been opened.

> The fchmod function is not part of POSIX.1. It is an extension provided by SVR4 and 4.3+BSD.

To change the permission bits of a file, the effective user ID of the process must equal the owner of the file, or the process must have superuser permissions.

The *mode* is specified as the bitwise OR of the constants shown in Figure 4.6.

mode	Description
S_ISUID	set-user-ID on execution
S_ISGID	set-group-ID on execution
S_ISVTX	saved-text (sticky bit)
S_IRWXU	read, write, and execute by user (owner)
S_IRUSR	read by user (owner)
S_IWUSR	write by user (owner)
S_IXUSR	execute by user (owner)
S_IRWXG	read, write, and execute by group
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXO	read, write, and execute by other (world)
S_IROTH	read by other (world)
S_IWOTH	write by other (world)
S_IXOTH	execute by other (world)

Figure 4.6 The *mode* constants for chmod functions, from <sys/stat.h>.

Note that nine of the entries in Figure 4.6 are the nine file access permission bits from Figure 4.4. We've added the two set-ID constants (S_IS[UG]ID), the saved-text constant (S_ISVTX), and the three combined constants (S_IRWX[UGO]). (Here we are using the standard Unix character class operator []. We mean any one of the characters contained within the square brackets. The final example, S_IRWX[UGO], refers to the three constants S_IRWXU, S_IRWXG, and S_IRWXO. This character class operator is a form of a regular expression that is provided by most Unix shells and many standard Unix applications.)

> The saved-text bit (S_ISVTX) is not part of POSIX.1. We describe its purpose in the next section.

Example

Recall the final state of the files foo and bar when we ran Program 4.3 to demonstrate the umask function:

```
$ ls -l foo bar
-rw-------  1 stevens          0 Nov 16 16:23 bar
-rw-rw-rw-  1 stevens          0 Nov 16 16:23 foo
```

Program 4.4 modifies the mode of these two files. After running Program 4.4 we see the final state of the two files is

```
$ ls -l foo bar
-rw-r--r--  1 stevens          0 Nov 16 16:23 bar
-rw-rwlrw-  1 stevens          0 Nov 16 16:23 foo
```

In this example we have set the permissions of foo relative to their current state. To do this we first call stat to obtain the current permissions and then modify them. We

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    "ourhdr.h"

int
main(void)
{
    struct stat     statbuf;

    /* turn on set-group-ID and turn off group-execute */

    if (stat("foo", &statbuf) < 0)
        err_sys("stat error for foo");
    if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("chmod error for foo");

    /* set absolute mode to "rw-r--r--" */

    if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
        err_sys("chmod error for bar");

    exit(0);
}
```

Program 4.4 Example of chmod function.

have explicitly turned on the set-group-ID bit and turned off the group-execute bit. Doing this for a regular file enables mandatory record locking, which we'll discuss in Section 12.3. Note that the `ls` command lists the group-execute permission as `l` to signify that mandatory record locking is enabled for this file. For the file `bar`, we set the permissions to an absolute value, regardless of the current permission bits.

Finally note that the time and date listed by the `ls` command did not change after we ran Program 4.4. We'll see in Section 4.18 that the chmod function updates only the time that the i-node was last changed. By default the `ls -l` lists the time the contents of the file were last modified. □

The chmod functions automatically clear two of the permission bits under the following conditions.

- If we try to set the sticky bit (`S_ISVTX`) of a regular file and we do not have superuser privileges, the sticky bit in the *mode* is automatically turned off. (We describe the sticky bit in the next section.) This means that only the superuser can set the sticky bit of a regular file. The reason is to prevent malicious users from setting the sticky bit and trying to fill up the swap area, if the system supports the saved-text feature.
- It is possible that the group ID of a newly created file is a group that the calling process does not belong to. Recall from Section 4.6 that it's possible for the

group ID of the new file to be the group ID of the parent directory. Specifically, if the group ID of the new file does not equal either the effective group ID of the process or one of the process's supplementary group IDs and if the process does not have superuser privileges, then the set-group-ID bit is automatically turned off. This prevents a user from creating a set-group-ID file owned by a group that the user doesn't belong to.

4.3+BSD and other Berkeley-derived systems add another security feature to try to prevent misuse of some of the protection bits. If a process that does not have superuser privileges writes to a file, the set-user-ID and set-group-ID bits are automatically turned off. If a malicious user finds a set-group-ID or set-user-ID file they can write to, even though they can modify the file, they lose the special privileges of the file.

4.10 Sticky Bit

The `S_ISVTX` bit has an interesting history. On earlier versions of Unix this bit was known as the *sticky bit*. If it was set for an executable program file, then the first time the program was executed a copy of the program's text was saved in the swap area when the process terminated. (The text portion of a program is the machine instructions.) This caused the program to load into memory faster the next time it was executed, because the swap area was handled as a contiguous file, compared to the possibly random location of data blocks in a normal Unix filesystem. The sticky bit was often set for common application programs such as the text editor and the passes of the C compiler. Naturally, there was a limit to the number of sticky files that could be contained in the swap area before running out of swap space, but it was a useful technique. The name sticky came about because the text portion of the file stuck around in the swap area until the system was rebooted. Later versions of Unix referred to this as the *saved-text* bit, hence the constant `S_ISVTX`. With today's newer Unix systems, most of which have a virtual memory system and a faster filesystem, the need for this technique has disappeared.

Both SVR4 and 4.3+BSD allow the sticky bit to be set for a directory. If the bit is set for a directory, a file in the directory can be removed or renamed only if the user has write permission for the directory, and either

- owns the file,
- owns the directory, or
- is the superuser.

The directories `/tmp` and `/var/spool/uucppublic` are candidates for the sticky bit—they are directories in which any user can typically create files. The permissions for these two directories are often read, write, and execute for everyone (user, group, and other). But users should not be able to delete or rename files owned by others.

The sticky bit is not defined by POSIX.1. It is an extension supported by both SVR4 and 4.3+BSD.

4.11 chown, fchown, and lchown Functions

The chown functions allow us to change the user ID of a file and the group ID of a file.

```
#include <sys/types.h>
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);

int fchown(int filedes, uid_t owner, gid_t group);

int lchown(const char *pathname, uid_t owner, gid_t group);
```

All three return: 0 if OK, –1 on error

These three functions operate similarly unless the referenced file is a symbolic link. In that case lchown changes the owners of the symbolic link itself, not the file pointed to by the symbolic link.

> The fchown function is not in the POSIX 1003.1–1990 standard, but will probably be added to 1003.1a. It is supported by SVR4 and 4.3+BSD.
>
> The lchown function is unique to SVR4. Under the non-SVR4 systems (POSIX.1 and 4.3+BSD), if the *pathname* for chown is a symbolic link then the ownership of the symbolic link is changed, not the ownership of the file referenced by the symbolic link. To change the ownership of the file referenced by the symbolic link we have to specify the *pathname* of the actual file itself, not the *pathname* of a symbolic link that points to the file.
>
> SVR4, 4.3+BSD, and XPG3 allow us to specify either of the arguments *owner* or *group* as –1 to leave the corresponding ID unchanged. This is not part of POSIX.1.

Historically, Berkeley-based systems have enforced the restriction that only the superuser can change the ownership of a file. This is to prevent users from giving away their files to others, thereby defeating any disk space quota restrictions. System V, however, has allowed any user to change the ownership of any files they own.

> POSIX.1 allows either form of operation, depending on the value of _POSIX_CHOWN_RESTRICTED. FIPS 151–1 requires _POSIX_CHOWN_RESTRICTED.
>
> With SVR4 this functionality is a configuration option, while 4.3+BSD always enforces the chown restriction.

Recall from Figure 2.5 that this constant can optionally be defined in the header <unistd.h> and can always be queried using either the pathconf or fpathconf functions. Also recall that this option can depend on the referenced file—it can be enabled or disabled on a per-filesystem basis. We'll use the phrase, if _POSIX_CHOWN_RESTRICTED is in effect, to mean if it applies to the particular file that we're talking about, regardless whether this actual constant is defined in the header.

If `_POSIX_CHOWN_RESTRICTED` is in effect for the specified file, then

1. only a superuser process can change the user ID of the file;
2. a nonsuperuser process can change the group ID of the file if
   a. the process owns the file (the effective user ID equals the user ID of the file), and
   b. *owner* equals the user ID of the file and *group* equals either the effective group ID of the process or one of the process's supplementary group IDs.

This means that when `_POSIX_CHOWN_RESTRICTED` is in effect you can't change the user ID of other users' files. You can change the group ID of files that you own, but only to groups that you belong to.

If these functions are called by a process other than a superuser process, on successful return both the set-user-ID and the set-group-ID bits are cleared.

4.12 File Size

The `st_size` member of the `stat` structure contains the size of the file in bytes. This field is meaningful only for regular files, directories, and symbolic links.

> SVR4 also defines the file size for a pipe as the number of bytes that are available for reading from the pipe. We'll discuss pipes in Section 14.2.

For a regular file, a file size of 0 is allowed—we'll get an end-of-file indication on the first read of the file.

For a directory, the file size is usually a multiple of a number such as 16 or 512. We talk about reading directories in Section 4.21.

For a symbolic link, the file size is the actual number of bytes in the filename. For example, in the case

```
lrwxrwxrwx  1 root            7 Sep 25 07:14 lib -> usr/lib
```

the file size of 7 is the length of the pathname `usr/lib`. (Note that symbolic links do not contain the normal C null byte at the end of the name, since the length is always specified by `st_size`.)

SVR4 and 4.3+BSD also provide the fields `st_blksize` and `st_blocks`. The first is the preferred block size for I/O for the file and the latter is the actual number of 512-byte blocks that are allocated. Recall from Section 3.9 that we encountered the minimum amount of time required to read a file when we used `st_blksize` for the `read` operations. The standard I/O library, which we describe in Chapter 5, also tries to read or write `st_blksize` bytes at a time, for efficiency.

> Be aware that different versions of Unix use units other than 512-byte blocks for `st_blocks`. Using this value is nonportable.

Holes in a File

In Section 3.6 we mentioned that a regular file can contain "holes." We showed an example of this in Program 3.2. Holes are created by seeking past the current end of file and writing some data. As an example, consider the following:

```
$ ls -l core
-rw-r--r--  1 stevens   8483248 Nov 18 12:18 core
$ du -s core
272     core
```

The size of the file `core` is just over 8 megabytes, yet the `du` command reports that the amount of disk space used by the file is 272 512-byte blocks (139,264 bytes). (The `du` command on many Berkeley-derived systems reports the number of 1024-byte blocks; SVR4 reports the number of 512-byte blocks.) Obviously this file has many holes.

As we mentioned in Section 3.6, the `read` function returns data bytes of 0 for any byte positions that have not been written. If we execute

```
$ wc -c core
 8483248 core
```

we can see that the normal I/O operations read up through the size of the file. (The `wc`(1) command with the `-c` option counts the number of characters (bytes) in the file.)

If we make a copy of this file, using a utility such as `cat`(1), all these holes are written out as actual data bytes of 0.

```
$ cat core > core.copy
$ ls -l core*
-rw-r--r--  1 stevens   8483248 Nov 18 12:18 core
-rw-rw-r--  1 stevens   8483248 Nov 18 12:27 core.copy
$ du -s core*
272     core
16592   core.copy
```

Here the actual number of bytes used by the new file is 8,495,104 ($512 \times 16{,}592$). The difference between this size and the size reported by `ls` is caused by the number of blocks used by the filesystem to hold pointers to the actual data blocks.

Interested readers should refer to Section 4.2 of Bach [1986] and Section 7.2 of Leffler et al. [1989] for additional details on the physical layout of files.

4.13 File Truncation

There are times when we would like to truncate a file by chopping off data at the end of the file. Emptying a file, which we can do with the `O_TRUNC` flag to `open`, is a special case of truncation.

```
#include <sys/types.h>
#include <unistd.h>

int truncate(const char *pathname, off_t length);

int ftruncate(int filedes, off_t length);
```

Both return: 0 if OK, –1 on error

These two functions truncate an existing file to *length* bytes. If the previous size of the file was greater than *length*, the data beyond *length* is no longer accessible. If the previous size was less than *length*, the effect is system dependent. If the implementation does extend a file, data between the old end-of-file and the new end-of-file will read as 0 (i.e., a hole is probably created in the file).

These two functions are provided by SVR4 and 4.3+BSD. They are not part of POSIX.1 or XPG3.

SVR4 truncates or extends a file. 4.3+BSD only truncates a file with these functions—they can't be used to extend a file.

There has never been a standard way of truncating a file with Unix. Truly portable applications must make a copy of the file, copying only the desired bytes of data.

SVR4 also includes an extension to `fcntl` (`F_FREESP`) that allows us to free any part of a file, not just a chunk at the end of the file.

We use `ftruncate` in Program 12.5 when we need to empty a file after obtaining a lock on the file.

4.14 Filesystems

To appreciate the concept of links to a file, we need a conceptual understanding of the structure of the Unix filesystem. Understanding the difference between an i-node and a directory entry that points to an i-node is also useful.

There are various implementations of the Unix filesystem in use today. SVR4, for example, supports two different types of disk filesystems: the traditional Unix System V filesystem (called S5), and the Unified File System (called UFS). We saw one difference between these two filesystem types in Figure 2.6. UFS is based on the Berkeley fast filesystem. SVR4 also supports additional nondisk filesystems, two distributed filesystems, and a bootstrap filesystem, none of which affects the following discussion. In this section we describe the traditional Unix System V filesystem. This type of filesystem dates back to Version 7.

We can think of a disk drive being divided into one or more partitions. Each partition can contain a filesystem, as shown in Figure 4.7.

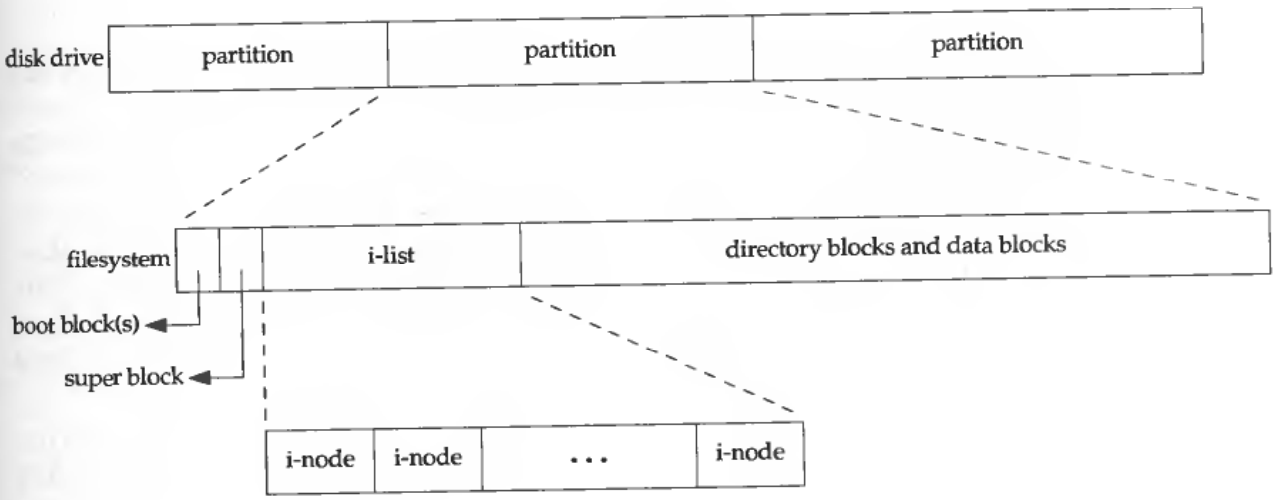


Figure 4.7 Disk drive, partitions, and a filesystem.

The i-nodes are fixed-length entries that contain most of the information about the file.

In Version 7 an i-node occupied 64 bytes; with 4.3+BSD an i-node occupies 128 bytes. Under SVR4 the size of an i-node on disk depends on the filesystem type: an S5 i-node occupies 64 bytes while a UFS i-node occupies 128 bytes.

If we examine the filesystem in more detail, ignoring the boot blocks and super block, we could have what is shown in Figure 4.8.

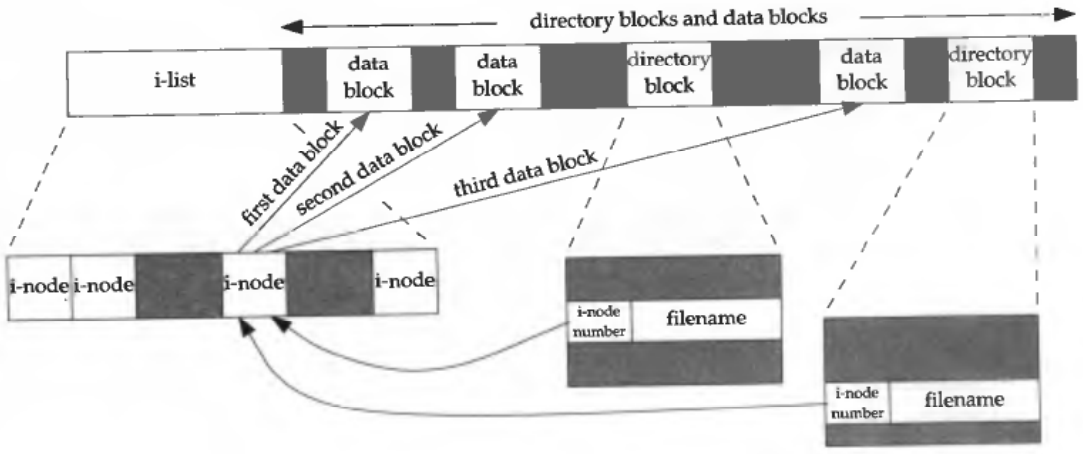


Figure 4.8 Filesystem in more detail.

Note the following points from Figure 4.8:

- We show two directory entries that point to the same i-node entry. Every i-node has a link count that contains the number of directory entries that point to the i-node. Only when the link count goes to 0 can the file be deleted (i.e., can the data blocks associated with the file be released). This is why the operation of "unlinking a file" does not always mean "deleting the blocks associated with the file." This is why the function that removes a directory entry is called `unlink` and not delete. In the `stat` structure the link count is contained in the `st_nlink` member. Its primitive system data type is `nlink_t`. These types of links are called hard links. Recall from Figure 2.7 that the POSIX.1 constant `LINK_MAX` specifies the maximum value for a file's link count.
- The other type of link is called a *symbolic link*. With a symbolic link, the actual contents of the file (the data blocks) contains the name of the file that the symbolic link points to. In the example

```
lrwxrwxrwx  1 root       7 Sep 25 07:14 lib -> usr/lib
```

  the filename in the directory entry is the three-character string `lib` and the 7 bytes of data in the file are `usr/lib`. The file type in the i-node would be `S_IFLNK` so that the system knows that this is a symbolic link.
- The i-node contains all the information about the file: the file type, the file's access permission bits, the size of the file, pointers to the data blocks for the file, and so on. Most of the information in the `stat` structure is obtained from the i-node. Only two items are stored in the directory entry: the filename and the i-node number. The data type for the i-node number is `ino_t`.
- Since the i-node number in the directory entry points to an i-node in the same filesystem, we cannot have a directory entry point to an i-node in a different filesystem. This is why the `ln`(1) command (make a new directory entry that points to an existing file) can't cross filesystems. We describe the `link` function in the next section.
- When renaming a file without changing filesystems, the actual contents of the file need not be moved—all that needs to be done is to have a new directory entry point to the existing i-node and have the old directory entry removed. For example, to rename the file `/usr/lib/foo` to `/usr/foo`, if the directories `/usr/lib` and `/usr` are on the same filesystem, the contents of the file `foo` need not be moved. This is how the `mv`(1) command usually operates.

We've talked about the concept of a link count for a regular file, but what about the link count field for a directory? Assume that we make a new directory in the working directory, as in

```
$ mkdir testdir
```

Figure 4.9 shows the result. Note in this figure we explicitly show the entries for dot and dot-dot.

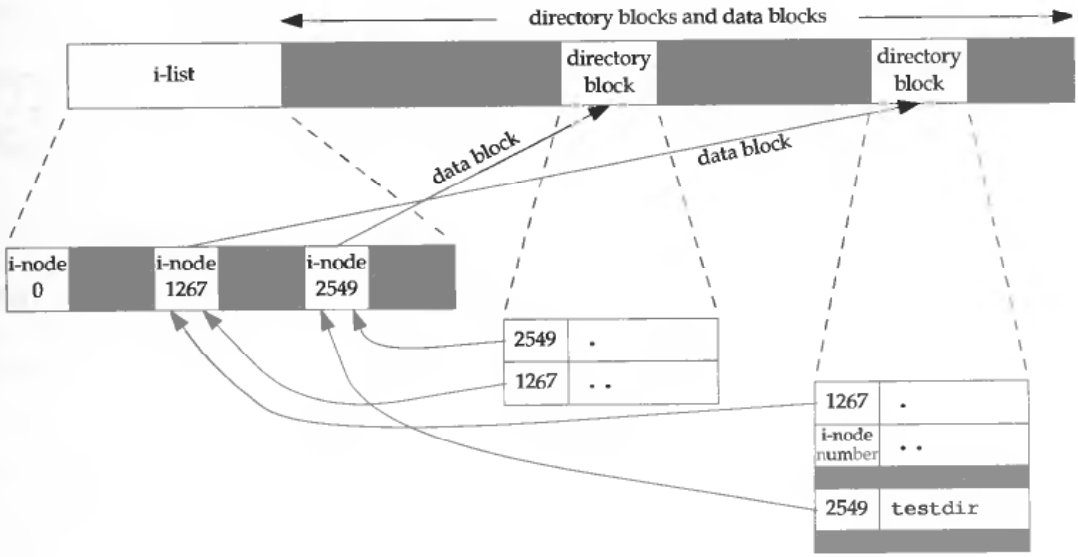


Figure 4.9 Sample filesystem after creating the directory `testdir`.

The i-node whose number is 2549 has a type field of "directory" and a link count equal to 2. Any leaf directory (a directory that does not contain any other directories) always has a link count of 2. The value of 2 is from the directory entry that names the directory (`testdir`) and from the entry for dot in that directory. The i-node whose number is 1267 has a type field of "directory" and a link count that is greater than or equal to 3. The reason we know the link count is greater than or equal to 3 is because minimally it is pointed to from the directory entry that names it (which we don't show in Figure 4.9), from dot, and from dot-dot in the `testdir` directory. Notice that every directory in the working directory causes the working directory's link count to be increased by 1.

As we said, this is the classic format of the Unix filesystem, which is described in detail in Chapter 4 of Bach [1986]. Refer to Chapter 7 of Leffler et al. [1989] for additional information on the changes made with the Berkeley fast filesystem.

4.15 `link`, `unlink`, `remove`, and `rename` Functions

As we saw in the previous section, any file can have multiple directory entries pointing to its i-node. The way we create a link to an existing file is with the `link` function.

```
#include <unistd.h>

int link(const char *existingpath, const char *newpath);
```

Returns: 0 if OK, –1 on error

This function creates a new directory entry, *newpath*, that references the existing file *existingpath*. If the *newpath* already exists an error is returned.

The creation of the new directory entry and the increment of the link count must be an atomic operation. (Recall the discussion of atomic operations in Section 3.11)

Most implementations, such as SVR4 and 4.3+BSD, require that both pathnames be on the same filesystem.

POSIX.1 allows an implementation to support linking across filesystems.

Only a superuser process can create a new link that points to a directory. The reason is that doing this can cause loops in the filesystem, which most utilities that process the filesystem aren't capable of handling. (We show an example of a loop introduced by a symbolic link in Section 4.16.)

To remove an existing directory entry we call the `unlink` function.

```
#include <unistd.h>

int unlink(const char *pathname);
```

Returns: 0 if OK, –1 on error

This function removes the directory entry and decrements the link count of the file referenced by *pathname*. If there are other links to the file, the data in the file is still accessible through the other links. The file is not changed if an error occurs.

We've mentioned before that to unlink a file we must have write permission and execute permission in the directory containing the directory entry, since it is the directory entry that we may be removing. Also, we mentioned in Section 4.10 that if the sticky bit is set in this directory we must have write permission for the directory and either

- own the file,
- own the directory, or
- have superuser privileges.

Only when the link count reaches 0 can the contents of the file be deleted. One other condition prevents the contents of a file from being deleted—as long as some process has the file open, its contents will not be deleted. When a file is closed the kernel first checks the count of the number of processes that have the file open. If this count has reached 0 then the kernel checks the link count, and if it is 0, the file's contents are deleted.

Example

Program 4.5 opens a file and then `unlinks` it. It then goes to sleep for 15 seconds before terminating.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>
#include    "ourhdr.h"

int
main(void)
{
    if (open("tempfile", O_RDWR) < 0)
        err_sys("open error");

    if (unlink("tempfile") < 0)
        err_sys("unlink error");

    printf("file unlinked\n");
    sleep(15);
    printf("done\n");

    exit(0);
}
```

Program 4.5 Open a file and then `unlink` it.

Running this program gives us

```
$ ls -l tempfile                    look at how big the file is
-rw-r--r--  1 stevens   9240990 Jul 31 13:42 tempfile
$ df /home                          check how much free space is available
Filesystem    kbytes    used   avail capacity  Mounted on
/dev/sd0h     282908  181979   72638    71%    /home
$ a.out &                           run Program 4.5 in the background
1364                                the shell prints its process ID
$ file unlinked                     the file is unlinked
ls -l tempfile                      see if the filename is still there
tempfile not found                  the directory entry is gone
$ df /home                          see if the space is available yet
Filesystem    kbytes    used   avail capacity  Mounted on
/dev/sd0h     282908  181979   72638    71%    /home
$ done                              the program is done, all open files are closed
df /home                            now the disk space should be available
Filesystem    kbytes    used   avail capacity  Mounted on
/dev/sd0h     282908  172939   81678    68%    /home
                                    now the 9.2 Mbytes of disk space are available
```

□

This property of `unlink` is often used by a program to assure that a temporary file it creates won't be left around in case the program crashes. The process creates a file using either `open` or `creat` and then immediately calls `unlink`. The file is not deleted, however, because it is still open. Only when the process either `close`s the file or terminates (which causes the kernel to close all its open files) is the file deleted.

If *pathname* is a symbolic link, `unlink` references the symbolic link, not the file referenced by the link.

The superuser can call `unlink` with *pathname* specifying a directory, but the function `rmdir` should be used instead to unlink a directory. We describe the `rmdir` function in Section 4.20.

We can also unlink a file or directory with the `remove` function. For a file, `remove` is identical to `unlink`. For a directory, `remove` is identical to `rmdir`.

```
#include <stdio.h>

int remove(const char *pathname);
```

Returns: 0 if OK, –1 on error

ANSI C specifies the `remove` function to delete a file. The name was changed from the historical Unix name of `unlink` since most non-Unix systems that implement the C standard don't support the concept of links to a file.

A file or directory is renamed with the `rename` function.

```
#include <stdio.h>

int rename(const char *oldname, const char *newname);
```

Returns: 0 if OK, –1 on error

This function is defined by ANSI C for files. (The C standard doesn't deal with directories.) POSIX.1 expanded the definition to include directories.

There are two conditions to describe, depending whether *oldname* refers to a file or a directory. We must also describe what happens if *newname* already exists.

1. If *oldname* specifies a file that is not a directory then we are renaming a file. In this case if *newname* exists, it cannot refer to a directory. If *newname* exists (and is not a directory), it is removed and *oldname* is renamed to *newname*. We must have write permission for the directory containing *oldname* and for the directory containing *newname*, since we are changing both directories.
2. If *oldname* specifies a directory then we are renaming a directory. If *newname* exists, it must refer to a directory and that directory must be empty. (When we say that a directory is empty, we mean that the only entries in the directory are dot and dot-dot.) If *newname* exists (and is an empty directory), it is removed and *oldname* is renamed to *newname*. Additionally, when we're renaming a directory, *newname* cannot contain a path prefix that names *oldname*. For example, we can't rename `/usr/foo` to `/usr/foo/testdir` since the old name (`/usr/foo`) is a path prefix of the new name and cannot be removed.

3. As a special case, if the *oldname* and *newname* refer to the same file, the function returns successfully without changing anything.

If *newname* already exists, we need permissions as if we were deleting it. Also, since we're removing the directory entry for *oldname* and possibly creating a directory entry for *newname*, we need write permission and execute permission in the directory containing *oldname* and in the directory containing *newname*.

4.16 Symbolic Links

A symbolic link is an indirect pointer to a file, unlike the hard links from the previous section, which pointed directly to the i-node of the file. Symbolic links were introduced to get around the limitations of hard links: (a) hard links normally require that the link and the file reside in the same filesystem, and (b) only the superuser can create a hard link to a directory. There are no filesystem limitations on a symbolic link and what it points to, and anyone can create a symbolic links to a directory. Symbolic links are typically used to move a file or an entire directory hierarchy to some other location on a system.

> Symbolic links were introduced with 4.2BSD and subsequently supported by SVR4. With SVR4 symbolic links are supported for both the traditional System V filesystem (S5) and the Unified File System (UFS).
>
> The original POSIX 1003.1–1990 standard does not include symbolic links. These will probably be added in 1003.1a.

When using functions that refer to a file by name we always need to know whether the function follows a symbolic link or not. If the function follows a symbolic link, a pathname argument to the function refers to the file pointed to by the symbolic link. Otherwise a pathname argument refers to the link itself, not the file pointed to by the link. Figure 4.10 summarizes whether the functions described in this chapter follow a symbolic link or not. The function `rmdir` is not in this figure since it is not defined for symbolic links (it returns an error). Also, the functions that take a file descriptor argument (`fstat`, `fchmod`, etc.) are not listed, since the handling of a symbolic link is done by the function that returns the file descriptor (usually `open`). Whether `chown` follows a symbolic link or not depends on the implementation—refer to Section 4.11 for details on the differences.

Example

It is possible to introduce loops into the filesystem using symbolic links. Most functions that look up a pathname return an `errno` of `ELOOP` when this occurs. Consider the following commands:

Function	Does not follow symbolic link	Follows symbolic link
access		•
chdir		•
chmod		•
chown	•	•
creat		•
exec		•
lchown	•	
link		•
lstat	•	
mkdir		•
mkfifo		•
mknod		•
open		•
opendir		•
pathconf		•
readlink	•	
remove	•	
rename	•	
stat		•
truncate		•
unlink	•	

Figure 4.10 Treatment of symbolic links by various functions.

```
$ mkdir foo                        make a new directory
$ touch foo/a                      create a 0-length file
$ ln -s ../foo foo/testdir         create a symbolic link
$ ls -l foo
total 1
-rw-rw-r--  1 stevens        0 Dec  6 06:06 a
lrwxrwxrwx  1 stevens        6 Dec  6 06:06 testdir -> ../foo
```

This creates a directory `foo` that contains the file a and a symbolic link that points to `foo`. We show this arrangement in Figure 4.11, drawing a directory as a circle and a file as a square. If we write a simple program that uses the standard function `ftw`(3) to descend through a file hierarchy, printing each pathname encountered, the output is

```
foo
foo/a
foo/testdir
foo/testdir/a
foo/testdir/testdir
foo/testdir/testdir/a
foo/testdir/testdir/testdir
foo/testdir/testdir/testdir/a
   (many more lines)
ftw returned -1: Too many levels of symbolic links
```

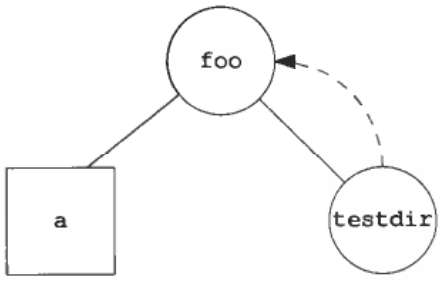


Figure 4.11 Symbolic link `testdir` that creates a loop.

We provide our own version of the `ftw` function in Section 4.21 that uses `lstat` instead of `stat`, to prevent it from following symbolic links.

A loop of this form is easy to remove—we are able to `unlink` the file `foo/testdir` since `unlink` does not follow a symbolic link. But if we create a hard link that forms a loop of this type, its removal is much harder.† This is why the `link` function will not form a hard link to a directory unless the process has superuser privileges.

When we `open` a file, if the pathname passed to `open` specifies a symbolic link, `open` follows the link to the specified file. If the file pointed to by the symbolic link doesn't exist, `open` returns an error saying that it can't open the file. This can confuse users who aren't familiar with symbolic links. For example,

```
$ ln -s /no/such/file myfile          create a symbolic link
$ ls myfile
myfile                                ls says it's there
$ cat myfile                          so we try to look at it
cat: myfile: No such file or directory
$ ls -l myfile                        try -l option
lrwxrwxrwx  1 stevens   13 Dec  6 07:27 myfile -> /no/such/file
```

The file `myfile` does exist, yet `cat` says there is no such file, because `myfile` is a symbolic link and the file pointed to by the symbolic link doesn't exist. The `-l` option to `ls` gives us two hints: the first character is an `l`, which means a symbolic link, and the sequence `->` also indicates a symbolic link. The `ls` command has another option (`-F`) that appends an at-sign to filenames that are symbolic links, which can help spot symbolic links in a directory listing without the `-l` option. □

† Indeed, the author did this on his own system as an experiment while writing this section. The filesystem got corrupted and the normal `fsck`(1) utility couldn't fix things. The deprecated tools `clri`(8) and `dcheck`(8) were needed to repair the filesystem.

4.17 symlink and readlink Functions

A symbolic link is created with the `symlink` function.

```
#include <unistd.h>

int symlink(const char *actualpath, const char *sympath);
```

Returns: 0 if OK, –1 on error

A new directory entry, *sympath*, is created that points to *actualpath*. It is not required that *actualpath* exist when the symbolic link is created. (We saw this in the example at the end of the previous section.) Also, *actualpath* and *sympath* need not reside in the same filesystem.

Since the `open` function follows a symbolic link, we need a way to open the link itself and read the name in the link. The `readlink` function does this.

```
#include <unistd.h>

int readlink(const char *pathname, char *buf, int bufsize);
```

Returns: number of bytes read if OK, –1 on error

This function combines the actions of `open`, `read`, and `close`.

If the function is successful it returns the number of bytes placed into *buf*. The contents of the symbolic link that are returned in *buf* are not null terminated.

4.18 File Times

Three time fields are maintained for each file. Their purpose is summarized in Figure 4.12.

Field	Description	Example	ls(1) option
st_atime	last-access time of file data	read	-u
st_mtime	last-modification time of file data	write	default
st_ctime	last-change time of i-node status	chmod, chown	-c

Figure 4.12 The three time values associated with each file.

Note the difference between the modification time (`st_mtime`) and the changed-status time (`st_ctime`). The modification time is when the contents of the file were last modified. The changed-status time is when the i-node of the file was last modified. We've described many operations in this chapter that affect the i-node without changing the actual contents of the file: changing the file access permissions, changing the user ID, changing the number of links, and so on. Since all the information in the i-node is stored separate from the actual contents of the file, we need the changed-status time, in addition to the modification time.

Note that the system does not maintain the last-access time for an i-node. This is why the functions `access` and `stat`, for example, don't change any of the three times.

The access time is often used by system administrators to delete files that have not been accessed for a certain amount of time. The classic example is the removal of files named `a.out` or `core` that haven't been accessed in the past week. The `find`(1) command is often used for this type of operation.

The modification time and the changed-status time can be used to archive only those files that have had their contents modified or their i-node modified.

The `ls` command displays or sorts only on one of the three time values. By default (when invoked with either the `-l` or `-t` option), it uses the modification time of a file. The `-u` option causes it to use the access time, and the `-c` option causes it to use the changed-status time.

Figure 4.13 summarizes the effects of the various functions that we've described on these three times. Recall from Section 4.14 that a directory is just a file containing directory entries (filenames and associated i-node numbers). Adding, deleting, or modifying these directory entries can affect the three times associated with that directory. This is why Figure 4.13 contains one column for the three times associated with the file (or directory), and another column for the three times associated with the parent directory of the referenced file (or directory). For example, creating a new file affects the directory that contains the new file, and it affects the i-node for the new file. Reading or writing a file, however, affects only the i-node of the file and has no effect on the directory. (The `mkdir` and `rmdir` functions are covered in Section 4.20. The `utime` function is covered in the next section. The six `exec` functions are described in Section 8.9. We describe the `mkfifo` and `pipe` functions in Chapter 14.)

4.19 `utime` Function

The access time and the modification time of a file can be changed with the `utime` function.

```
#include <sys/types.h>
#include <utime.h>

int utime(const char *pathname, const struct utimbuf *times);
```

Returns: 0 if OK, −1 on error

The structure used by this function is

```
struct utimbuf {
  time_t  actime;    /* access time */
  time_t  modtime;   /* modification time */
}
```

The two time values in the structure are calendar times, which count seconds since the Epoch, as described in Section 1.10.

Function	Referenced file (or directory)			Parent directory of referenced file (or directory)			Note
	a	m	c	a	m	c	
chmod, fchmod			•				
chown, fchown			•				
creat	•	•	•		•	•	O_CREAT new file
creat		•	•				O_TRUNC existing file
exec	•						
lchown			•				
link			•		•	•	
mkdir	•	•	•		•	•	
mkfifo	•	•	•		•	•	
open	•	•	•		•	•	O_CREAT new file
open		•	•				O_TRUNC existing file
pipe	•	•	•				
read	•						
remove			•		•	•	remove file = unlink
remove					•	•	remove directory = rmdir
rename			•		•	•	for both arguments
rmdir					•	•	
truncate, ftruncate		•	•				
unlink			•		•	•	
utime	•	•	•				
write		•	•				

Figure 4.13 Effect of various functions on the access, modification, and changed-status times.

The operation of this function, and the privileges required to execute it, depend on whether the *times* argument is NULL.

1. If *times* is a null pointer, the access time and modification time are both set to the current time. To do this, either (a) the effective user ID of the process must equal the owner ID of the file, or (b) the process must have write permission for the file.
2. If *times* is a nonnull pointer, the access time and the modification time are set to the values in the structure pointed to by *times*. For this case the effective user ID of the process must equal the owner ID of the file, or the process must be a superuser process. Merely having write permission for the file is not adequate.

Note that we are not able to specify a value for the changed-status time, `st_ctime` (the time the i-node was last changed), since this field is automatically updated when the `utime` function is called.

On some versions of Unix the `touch`(1) command uses this function. Also, the standard archive programs, `tar`(1) and `cpio`(1), optionally call `utime` to set the times for a file to their values when they were archived.

Example

Program 4.6 truncates files to zero-length with the O_TRUNC option with the open function, but does not change their access time or modification time. To do this it first obtains the times with the stat function, truncates the file, and then resets the times with the utime function.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>
#include    <utime.h>
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    int             i;
    struct stat     statbuf;
    struct utimbuf  timebuf;

    for (i = 1; i < argc; i++) {
        if (stat(argv[i], &statbuf) < 0) {  /* fetch current times */
            err_ret("%s: stat error", argv[i]);
            continue;
        }
        if (open(argv[i], O_RDWR | O_TRUNC) < 0) {  /* truncate */
            err_ret("%s: open error", argv[i]);
            continue;
        }
        timebuf.actime  = statbuf.st_atime;
        timebuf.modtime = statbuf.st_mtime;
        if (utime(argv[i], &timebuf) < 0) {     /* reset times */
            err_ret("%s: utime error", argv[i]);
            continue;
        }
    }
    exit(0);
}
```

Program 4.6 Example of utime function.

We can demonstrate Program 4.6 with the following script:

```
$ ls -l changemod times          look at sizes and last-modification times
-rwxrwxr-x  1 stevens   24576 Dec  4 16:13 changemod
-rwxrwxr-x  1 stevens   24576 Dec  6 09:24 times
$ ls -lu changemod times         look at last-access times
-rwxrwxr-x  1 stevens   24576 Feb  1 12:44 changemod
-rwxrwxr-x  1 stevens   24576 Feb  1 12:44 times
$ date                           print today's date
Sun Feb  3 18:22:33 MST 1991
```

```
$ a.out changemod times          run Program 4.6
$ ls -l changemod times          and check the results
-rwxrwxr-x  1 stevens      0 Dec  4 16:13 changemod
-rwxrwxr-x  1 stevens      0 Dec  6 09:24 times
$ ls -lu changemod times         check the last-access times also
-rwxrwxr-x  1 stevens      0 Feb  1 12:44 changemod
-rwxrwxr-x  1 stevens      0 Feb  1 12:44 times
$ ls -lc changemod times         and the changed-status times
-rwxrwxr-x  1 stevens      0 Feb  3 18:23 changemod
-rwxrwxr-x  1 stevens      0 Feb  3 18:23 times
```

As we expect, the last-modification times and the last-access times are not changed. The changed-status times, however, are changed to the time that the program was run. (The reason the last-access times are identical for the two files is because that was the time the directory was archived using `tar`.) □

4.20 `mkdir` and `rmdir` Functions

Directories are created with the `mkdir` function and deleted with the `rmdir` function.

```
#include <sys/types.h>
#include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);
```

Returns: 0 if OK, –1 on error

This function creates a new, empty directory. The entries for dot and dot-dot are automatically created. The specified file access permissions, *mode*, are modified by the file mode creation mask of the process.

A common mistake is to specify the same *mode* as for a file (read and write permissions only). But for a directory we normally want at least one of the execute bits enabled, to allow access to filenames within the directory. (See Exercise 4.18.)

The user ID and group ID of the new directory are established according to the rules we described in Section 4.6.

SVR4 also has the new directory inherit the set-group-ID bit from the parent directory. This is so that files created in the new directory will inherit the group ID of that directory.

4.3+BSD does not require this inheriting of the set-group-ID bit, since newly created files and directories always inherit the group ID of the parent directory, regardless of the set-group-ID bit.

Earlier versions of Unix did not have the `mkdir` function. It was introduced with 4.2BSD and SVR3. In the earlier versions a process had to call the `mknod` function to create a new directory. But use of the `mknod` function was restricted to superuser processes. To circumvent this, the normal command that created a directory, `mkdir`(1), had to be owned by root with the set-user-ID bit on. To create a directory from a process, the `mkdir`(1) command had to be invoked with the `system`(3) function.

An empty directory is deleted with the `rmdir` function.

```
#include <unistd.h>

int rmdir(const char *pathname);
```

Returns: 0 if OK, –1 on error

If the link count of the directory becomes 0 with this call, and no other process has the directory open, then the space occupied by the directory is freed. If one or more processes have the directory open when the link count reaches 0, the last link is removed and the dot and dot-dot entries are removed before this function returns. Additionally, no new files can be created in the directory. The directory is not freed, however, until the last process closes it. (Even though some other process has the directory open, they can't be doing much in the directory since the directory had to be empty for the `rmdir` function to succeed.)

4.21 Reading Directories

Directories can be read by anyone who has access permission to read the directory. But only the kernel can write to a directory (to preserve filesystem sanity). Recall from Section 4.5 that the write permission bits and execute permission bits for a directory determine if we can create new files in the directory and remove files from the directory—they don't specify if we can write to the directory itself.

The actual format of a directory depends on the Unix implementation. Earlier systems, such as Version 7, had a simple structure: each directory entry was 16 bytes, with 14 bytes for the filename and two bytes for the i-node number. When longer filenames were added to 4.2BSD each entry became variable length, which means any program that reads a directory is now system dependent. To simplify this, a set of directory routines were developed and are now part of POSIX.1.

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *pathname);
```

Returns: pointer if OK, NULL on error

```
struct dirent *readdir(DIR *dp);
```

Returns: pointer if OK, NULL on error

```
void rewinddir(DIR *dp);

int closedir(DIR *dp);
```

Returns: 0 if OK, –1 on error

Recall our use of these functions in Program 1.1, our bare bones implementation of the `ls` command.

The `dirent` structure defined in the file `<dirent.h>` is implementation dependent. SVR4 and 4.3+BSD define the structure to contain at least the following two members:

```
struct dirent {
  ino_t  d_ino;                    /* i-node number */
  char   d_name[NAME_MAX + 1];     /* null-terminated filename */
}
```

The `d_ino` entry is not defined by POSIX.1, since it's an implementation feature. POSIX.1 defines only the `d_name` entry in this structure.

Note that `NAME_MAX` is not a defined constant with SVR4—its value depends on the filesystem in which the directory resides, and its value is usually obtained from the `fpathconf` function. A common value for `NAME_MAX` on a BSD-type filesystem is 255. (Recall Figure 2.7.) Since the filename is null terminated, however, it doesn't matter how the array `d_name` is defined in the header.

The `DIR` structure is an internal structure used by these four functions to maintain information about the directory being read. It serves a purpose similar to the `FILE` structure that is maintained by the standard I/O library (which we describe in Chapter 5).

The pointer to a `DIR` structure that is returned by `opendir` is then used with the other three functions. `opendir` initializes things so that the first `readdir` reads the first entry in the directory. The ordering of entries within the directory is implementation dependent. It is usually not alphabetical.

Example

We'll use these directory routines to write a program that traverses a file hierarchy. The goal is to produce the count of the different types of files that we show in Figure 4.2. Program 4.7 takes a single argument, the starting pathname, and recursively descends the hierarchy from that point. System V provides a function, `ftw`(3), that performs the actual traversal of the hierarchy, calling a user-defined function for each file. The problem with this function is that it calls the `stat` function for each file, which causes the program to follow symbolic links. For example, if we start at the root and have a symbolic link named `/lib` that points to `/usr/lib`, all the files in the directory `/usr/lib` are counted twice. To correct this, SVR4 provides an additional function, `nftw`(3), with an option that stops it from following symbolic links. While we could use `nftw`, we'll write our own simple file walker to show the use of the directory routines.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <dirent.h>
#include    <limits.h>
#include    "ourhdr.h"
```

```
typedef int Myfunc(const char *, const struct stat *, int);
                /* function type that's called for each filename */

static Myfunc   myfunc;
static int      myftw(char *, Myfunc *);
static int      dopath(Myfunc *);

static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;

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

    if (argc != 2)
        err_quit("usage:  ftw  <starting-pathname>");

    ret = myftw(argv[1], myfunc);       /* does it all */

    if ( (ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock) == 0)
        ntot = 1;       /* avoid divide by 0; print 0 for all counts */
    printf("regular files  = %7ld, %5.2f %%\n", nreg,  nreg*100.0/ntot);
    printf("directories    = %7ld, %5.2f %%\n", ndir,  ndir*100.0/ntot);
    printf("block special  = %7ld, %5.2f %%\n", nblk,  nblk*100.0/ntot);
    printf("char special   = %7ld, %5.2f %%\n", nchr,  nchr*100.0/ntot);
    printf("FIFOs          = %7ld, %5.2f %%\n", nfifo, nfifo*100.0/ntot);
    printf("symbolic links = %7ld, %5.2f %%\n", nslink,nslink*100.0/ntot);
    printf("sockets        = %7ld, %5.2f %%\n", nsock, nsock*100.0/ntot);

    exit(ret);
}

/*
 * Descend through the hierarchy, starting at "pathname".
 * The caller's func() is called for every file.
 */

#define FTW_F   1       /* file other than directory */
#define FTW_D   2       /* directory */
#define FTW_DNR 3       /* directory that can't be read */
#define FTW_NS  4       /* file that we can't stat */

static char *fullpath;      /* contains full pathname for every file */

static int                  /* we return whatever func() returns */
myftw(char *pathname, Myfunc *func)
{
    fullpath = path_alloc(NULL);    /* malloc's for PATH_MAX+1 bytes */
                                    /* (Program 2.2) */
    strcpy(fullpath, pathname);     /* initialize fullpath */

    return(dopath(func));
}
```

```
/*
 * Descend through the hierarchy, starting at "fullpath".
 * If "fullpath" is anything other than a directory, we lstat() it,
 * call func(), and return.  For a directory, we call ourself
 * recursively for each name in the directory.
 */
static int                      /* we return whatever func() returns */
dopath(Myfunc* func)
{
    struct stat     statbuf;
    struct dirent   *dirp;
    DIR             *dp;
    int             ret;
    char            *ptr;

    if (lstat(fullpath, &statbuf) < 0)
        return(func(fullpath, &statbuf, FTW_NS));   /* stat error */

    if (S_ISDIR(statbuf.st_mode) == 0)
        return(func(fullpath, &statbuf, FTW_F));    /* not a directory */

    /*
     * It's a directory.  First call func() for the directory,
     * then process each filename in the directory.
     */

    if ( (ret = func(fullpath, &statbuf, FTW_D)) != 0)
        return(ret);

    ptr = fullpath + strlen(fullpath);  /* point to end of fullpath */
    *ptr++ = '/';
    *ptr = 0;

    if ( (dp = opendir(fullpath)) == NULL)
        return(func(fullpath, &statbuf, FTW_DNR));
                                        /* can't read directory */

    while ( (dirp = readdir(dp)) != NULL) {
        if (strcmp(dirp->d_name, ".") == 0  ||
            strcmp(dirp->d_name, "..") == 0)
                continue;           /* ignore dot and dot-dot */

        strcpy(ptr, dirp->d_name);  /* append name after slash */

        if ( (ret = dopath(func)) != 0)     /* recursive */
            break;  /* time to leave */
    }
    ptr[-1] = 0;    /* erase everything from slash onwards */

    if (closedir(dp) < 0)
        err_ret("can't close directory %s", fullpath);

    return(ret);
}
```

```
static int
myfunc(const char *pathname, const struct stat *statptr, int type)
{
    switch (type) {
    case FTW_F:
        switch (statptr->st_mode & S_IFMT) {
        case S_IFREG:   nreg++;     break;
        case S_IFBLK:   nblk++;     break;
        case S_IFCHR:   nchr++;     break;
        case S_IFIFO:   nfifo++;    break;
        case S_IFLNK:   nslink++;   break;
        case S_IFSOCK:  nsock++;    break;
        case S_IFDIR:
            err_dump("for S_IFDIR for %s", pathname);
                    /* directories should have type = FTW_D */
        }
        break;

    case FTW_D:
        ndir++;
        break;

    case FTW_DNR:
        err_ret("can't read directory %s", pathname);
        break;

    case FTW_NS:
        err_ret("stat error for %s", pathname);
        break;

    default:
        err_dump("unknown type %d for pathname %s", type, pathname);
    }

    return(0);
}
```

Program 4.7 Recursively descend a directory hierarchy, counting file types.

We have provided more generality in this program than needed. This was done to illustrate the actual `ftw` function. For example, the function `myfunc` always returns 0, even though the function that calls it is prepared to handle a nonzero return. □

For additional information on descending through a filesystem and the use of this technique in many standard Unix commands (`find`, `ls`, `tar`, etc.), refer to Fowler, Korn, and Vo [1989]. 4.3+BSD provides a new set of directory traversal functions—see the `fts`(3) manual page.

4.22 chdir, fchdir, and getcwd Functions

Every process has a current working directory. This directory is where the search for all relative pathnames starts (all pathnames that do not begin with a slash). When a user logs in to a Unix system, the current working directory normally starts at the directory specified by the sixth field in the `/etc/passwd` file—the user's home directory. The current working directory is an attribute of a process; the home directory is an attribute of a login name. We can change the current working directory of the calling process by calling the `chdir` or `fchdir` functions.

```
#include <unistd.h>

int chdir(const char *pathname);

int fchdir(int filedes);
```

Both return: 0 if OK, –1 on error

We can specify the new current working directory as either a *pathname* or through an open file descriptor.

The `fchdir` function is not part of POSIX.1. It is an extension supported by SVR4 and 4.3+BSD.

Example

Since the current working directory is an attribute of a process it cannot affect processes that invoke the process that executes the `chdir`. (We describe the relationship between processes in more detail in Chapter 8.) This means that Program 4.8 doesn't do what we expect. If we compile Program 4.8 and call the executable `mycd` we get the following:

```
$ pwd
/usr/lib
$ mycd
chdir to /tmp succeeded
$ pwd
/usr/lib
```

The current working directory for the shell that executed the `mycd` program didn't change. For this reason, the `chdir` function has to be called directly from the shell, so the `cd` command is built into the shells. □

Since the kernel must maintain knowledge of the current working directory, we should be able to fetch its current value. Unfortunately, all the kernel maintains for each process is the i-node number and device identification for the current working directory. The kernel does not maintain the full pathname of the directory.

```
#include    "ourhdr.h"

int
main(void)
{
    if (chdir("/tmp") < 0)
        err_sys("chdir failed");

    printf("chdir to /tmp succeeded\n");
    exit(0);
}
```

Program 4.8 Example of `chdir` function.

What we need is a function that starts at the current working directory (dot) and works its way up the directory hierarchy (using dot-dot to move up one level). At each directory it reads the directory entries until it finds the name that corresponds to the i-node of the directory that it just came from. Repeating this procedure until the root is encountered yields the entire absolute pathname of the current working directory. Fortunately a function is already provided for us that does this task.

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

Returns: *buf* if OK, NULL on error

We must pass this function the address of a buffer, *buf*, and its *size*. The buffer must be large enough to accommodate the absolute pathname plus a terminating null byte, or an error is returned. (Recall the discussion of allocating space for a maximum-sized pathname in Section 2.5.7.)

> Some implementations of `getcwd` allow the first argument *buf* to be NULL. In this case the function calls `malloc` to allocate *size* number of bytes dynamically. This is not part of POSIX.1 or XPG3 and should be avoided.

Example

Program 4.9 changes to a specific directory and then calls `getcwd` to print the working directory. If we run the program we get

```
$ a.out
cwd = /var/spool/uucppublic
$ ls -l /usr/spool
lrwxrwxrwx  1 root  12 Jan 31 07:57 /usr/spool -> ../var/spool
```

```
#include    "ourhdr.h"

int
main(void)
{
    char    *ptr;
    int     size;

    if (chdir("/usr/spool/uucppublic") < 0)
        err_sys("chdir failed");

    ptr = path_alloc(&size);    /* our own function */
    if (getcwd(ptr, size) == NULL)
        err_sys("getcwd failed");

    printf("cwd = %s\n", ptr);
    exit(0);
}
```

Program 4.9 Example of `getcwd` function.

Notice that `chdir` follows the symbolic link (as we expect it to, from Figure 4.10) but when `getcwd` goes up the directory tree, it has no idea when it hits the `/var/spool` directory that it is pointed to by the symbolic link `/usr/spool`. This is a characteristic of symbolic links. □

4.23 Special Device Files

The two fields `st_dev` and `st_rdev` are often confused. We'll need to use these fields in Section 11.9 when we write the `ttyname` function. The rules are simple.

- Every filesystem is known by its major and minor device number. This device number is encoded in the primitive system data type `dev_t`. Recall from Figure 4.7 that a disk drive often contains several filesystems.
- We can usually access the major and minor device numbers through two macros defined by most implementations: `major` and `minor`. This means we don't care how the two numbers are stored in a `dev_t` object.

> Early systems stored the device number in a 16-bit integer with 8 bits for the major number and 8 bits for the minor number. SVR4 uses 32 bits: 14 for the major and 18 for the minor. 4.3+BSD uses 16 bits: 8 for the major and 8 for the minor.
>
> POSIX.1 states that the `dev_t` type exists, but doesn't define what it contains or how to get at its contents. The macros `major` and `minor` are defined by most implementations. Which header they are defined in depends on the system.

- The st_dev value for every filename on a system is the device number of the filesystem containing that filename and its corresponding i-node.
- Only character special files and block special files have an st_rdev value. This value contains the device number for the actual device.

Example

Program 4.10 prints the device number for each command-line argument. Additionally, if the argument refers to a character special file or a block special file, the st_rdev value for the special file is also printed.

```
#include    <sys/types.h>   /* BSD: defines major() and minor() */
#include    <sys/stat.h>
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    int         i;
    struct stat buf;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            err_ret("lstat error");
            continue;
        }

        printf("dev = %d/%d", major(buf.st_dev),  minor(buf.st_dev));

        if (S_ISCHR(buf.st_mode) || S_ISBLK(buf.st_mode)) {
            printf(" (%s) rdev = %d/%d",
                    (S_ISCHR(buf.st_mode)) ? "character" : "block",
                    major(buf.st_rdev), minor(buf.st_rdev));
        }
        printf("\n");
    }
    exit(0);
}
```

Program 4.10 Print st_dev and st_rdev values.

Under SVR4 the header <sys/sysmacros.h> must be included to define the macros major and minor. Running this program gives us the following output:

```
$ a.out / /home/stevens /dev/tty[ab]
/: dev = 7/0
/home/stevens: dev = 7/7
/dev/ttya: dev = 7/0 (character) rdev = 12/0
/dev/ttyb: dev = 7/0 (character) rdev = 12/1
```

```
$ mount                                  which directories are mounted on which devices?
/dev/sd0a on /
/dev/sd0h on /home
$ ls -l /dev/sd0[ah] /dev/tty[ab]
brw-r-----  1 root        7,    0 Jan 31 08:23 /dev/sd0a
brw-r-----  1 root        7,    7 Jan 31 08:23 /dev/sd0h
crw-rw-rw-  1 root       12,    0 Jan 31 08:22 /dev/ttya
crw-rw-rw-  1 root       12,    1 Jul  9 10:11 /dev/ttyb
```

The first two arguments to the program are directories (root and `/home/stevens`), and the next two are the device names `/dev/tty[ab]`. We expect the devices to be character special files. The output from the program shows that the root directory has a different device number than the `/home/stevens` directory. This indicates that they are on different filesystems. Running the `mount`(1) command verifies this. We then use `ls` to look at the two disk devices reported by `mount` and the two terminal devices. The two disk devices are block special files, and the two terminal devices are character special files. (Normally the only types of devices that are block special files are those that can contain random-access filesystems: disk drives, floppy disk drives, and CD-ROMs, for example. Older versions of Unix supported magnetic tapes for filesystems, but this was never widely used.) Note that the filenames and i-nodes for the two terminal devices (`st_dev`) are on device 7/0 (the root filesystem, which contains the `/dev` filesystem) but their actual device numbers are 12/0 and 12/1. □

4.24 `sync` and `fsync` Functions

Traditional Unix implementations have a buffer cache in the kernel through which most disk I/O passes. When we `write` data to a file the data is normally copied by the kernel into one of its buffers and queued for I/O at some later time. This is called *delayed write*. (Chapter 3 of Bach [1986] discusses this buffer cache in detail.)

The kernel eventually writes all the delayed-write blocks to disk, normally when it needs to reuse the buffer for some other disk block. To ensure consistency of the actual filesystem on disk with the contents of the buffer cache, the `sync` and `fsync` functions are provided.

```
#include <unistd.h>

void sync(void);

int fsync(int filedes);
```

Returns: 0 if OK, –1 on error

`sync` just queues all the modified block buffers for writing and returns; it does not wait for the actual I/O to take place.

The function `sync` is normally called every 30 seconds from a system daemon (often called `update`). This guarantees regular flushing of the kernel's block buffers. The command `sync`(1) also calls the `sync` function.

The function `fsync` refers only to a single file (specified by the file descriptor *filedes*), and waits for the I/O to complete before returning. The intended use of `fsync` is for an application such as a database that needs to be sure that the modified blocks have been written to the disk. Compare `fsync`, which updates a file's contents when we say so, with the `O_SYNC` flag (described in Section 3.13), which updates a file's contents every time we `write` to the file.

> `sync` and `fsync` are supported by both SVR4 and 4.3+BSD. Neither is part of POSIX.1, but `fsync` is required by XPG3.

4.25 Summary of File Access Permission Bits

We've covered all the file access permission bits, some of which serve multiple purposes. Figure 4.14 summarizes all these permission bits and their interpretation when applied to a regular file versus their interpretation when applied to a directory.

Constant	Description	Effect on regular file	Effect on directory
`S_ISUID`	set-user-ID	set effective user ID on execution	(not used)
`S_ISGID`	set-group-ID	if group-execute set then set effective group ID on execution; otherwise enable mandatory record locking	set group ID of new files created in directory to group ID of directory
`S_ISVTX`	sticky bit	save program text in swap area (if supported)	restrict removal and renaming of files in directory
`S_IRUSR`	user-read	user permission to read file	user permission to read directory entries
`S_IWUSR`	user-write	user permission to write file	user permission to remove and create files in directory
`S_IXUSR`	user-execute	user permission to execute file	user permission to search for given pathname in directory
`S_IRGRP`	group-read	group permission to read file	group permission to read directory entries
`S_IWGRP`	group-write	group permission to write file	group permission to remove and create files in directory
`S_IXGRP`	group-execute	group permission to execute file	group permission to search for given pathname in directory
`S_IROTH`	other-read	other permission to read file	other permission to read directory entries
`S_IWOTH`	other-write	other permission to write file	other permission to remove and create files in directory
`S_IXOTH`	other-execute	other permission to execute file	other permission to search for given pathname in directory

Figure 4.14 Summary of file access permission bits.

The final nine constants can also be grouped into threes, since

```
S_IRWXU = S_IRUSR | S_IWUSR | S_IXUSR
S_IRWXG = S_IRGRP | S_IWGRP | S_IXGRP
S_IRWXO = S_IROTH | S_IWOTH | S_IXOTH
```

4.26 Summary

This chapter has been centered around the `stat` function. We've gone through each member in the `stat` structure in detail. This in turn has led us to examine all the attributes of Unix files. A thorough understanding of all the properties of a file and all the functions that operate on files is essential to all Unix programming.

Exercises

4.1 Modify Program 4.1 to use `stat` instead of `lstat`. What changes if one of the command-line arguments is a symbolic link?

4.2 We indicated in Figure 4.1 that SVR4 doesn't currently provide the `S_ISLNK` macro. But SVR4 does support symbolic links and defines `S_IFLNK` in `<sys/stat.h>`. (Perhaps someone forgot to define `S_ISLNK`?) Devise a way around this omission that can be placed in `ourhdr.h`, so any programs that need the `S_ISLNK` macro can use it.

4.3 What happens if the file mode creation mask is set to 777 (octal)? Verify the results using your shell's `umask` command.

4.4 Verify that turning off user-read permission for a file that you own denies your access to the file.

4.5 Run Program 4.3 *after* creating the files `foo` and `bar`. What happens?

4.6 In Section 4.12 we said that a file size of 0 is valid for a regular file. We also said that the `st_size` field is defined for directories and symbolic links. Should we ever see a file size of 0 for a directory or a symbolic link?

4.7 Write a utility like cp(1) that copies a file containing holes, without writing the bytes of 0 to the output file.

4.8 Note in output from the `ls` command in Section 4.12 that the files `core` and `core.copy` have different access permissions. If the `umask` value didn't change between the creation of the two files, explain how the difference could have occurred.

4.9 When running Program 4.5 we check the available disk space with the `df`(1) command. Why didn't we use the du(1) command?

4.10 In Figure 4.13 we show the `unlink` function as modifying the changed-status time of the file itself. How can this happen?

4.11 In Section 4.21 how does the system's limit on the number of open files affect the `myftw` function?

4.12 In Section 4.21 our version of `ftw` never changes its directory. Modify this routine so that each time it encounters a directory it does a `chdir` to that directory, allowing it to use the

filename and not the pathname for each call to `lstat`. When all the entries in a directory have been processed, execute `chdir("..")`. Compare the time used by this version and the version in the text.

4.13 Each process also has a root directory that is used for resolution of absolute pathnames. This root directory can be changed with the `chroot` function. Look up the description for this function in your manuals. When might this function be useful?

4.14 How can you set only one of the two time values with the `utime` function?

4.15 Some versions of the `finger`(1) command output "New mail received ..." and "unread since ..." where ... are the corresponding times and dates. How can the program determine these two times and dates?

4.16 Examine the archive formats by the `cpio`(1) and `tar`(1) commands. (These descriptions are usually found in Section 5 of the *Unix Programmer's Manual*.) How many of the three possible time values are saved for each file? When a file is restored, what value do you think the access time is set to, and why?

4.17 The command `file`(1) tries to determine the logical type of a file (C program, Fortran program, shell script, etc.) by reading the first part of the file, examining the contents, and applying some heuristics. Also, some Unix systems provide a command that allows us to execute another command and obtain a trace of all the system calls executed by the command. (Under SVR4 the command is `truss`(1). Under 4.3+BSD the commands are `ktrace`(1) and `kdump`(1). The following example uses the SunOS `trace`(1) command.) If we run a system call trace of the `file` command

```
trace file a.out
```

we find it calls the following functions

```
lstat ("a.out", 0xf7fff650) = 0
open ("a.out", 0, 0) = 3
read (3, ""..., 512) = 512
fstat (3, 0xf7fff160) = 0
write (1, "a.out: demand paged execu"..., 44) = 44
a.out: demand paged executable not stripped
utime ("a.out", 0xf7fff1b0) = 0
```

Why is the `file` command calling `utime`?

4.18 Does Unix have a fundamental limitation on the depth of a directory tree? To find out, write a program that creates a directory and then changes to that directory, in a loop. Make certain that the length of the absolute pathname of the leaf of this directory is greater than your system's `PATH_MAX` limit. Can you call `getcwd` to fetch the directory's pathname? How do the standard Unix tools deal with this long pathname? Can you archive the directory using either `tar` or `cpio`?

4.19 In Section 3.15 we described the `/dev/fd` feature. For any user to be able to access these files, their permissions must be `rw-rw-rw-`. Some programs that create an output file delete the file first, in case it already exists (ignoring the return code).

```
unlink(path);
if ( (fd = creat(path, FILE_MODE)) < 0)
    err_sys(...);
```

What happens if `path` is `/dev/fd/1`?

5

Standard I/O Library

5.1 Introduction

In this chapter we describe the standard I/O library. This library is specified by the ANSI C standard because it has been implemented on many operating systems other than Unix. This library handles details such as buffer allocation and performing I/O in optimal-sized chunks, obviating our need to worry about using the correct block size (as in Section 3.9). This makes the library easy to use, but at the same time introduces another set of problems if we're not cognizant of what's going on.

> The standard I/O library was written by Dennis Ritchie around 1975. It was a major revision of the Portable I/O library written by Mike Lesk. Surprisingly, little has changed in the standard I/O library after more than 15 years.

5.2 Streams and `FILE` Objects

In Chapter 3 all the I/O routines centered around file descriptors. When a file is opened a file descriptor is returned, and that descriptor is then used for all subsequent I/O operations. With the standard I/O library the discussion centers around *streams*. (Do not confuse the standard I/O term *stream* with the STREAMS I/O system that is part of System V.) When we open or create a file with the standard I/O library we say that we have associated a stream with the file.

When we open a stream, the standard I/O function `fopen` returns a pointer to a `FILE` object. This object is normally a structure that contains all the information required by the standard I/O library to manage the stream: the file descriptor used for actual I/O, a pointer to a buffer for the stream, the size of the buffer, a count of the number of characters currently in the buffer, an error flag, and the like.

Application software should never need to examine a FILE object. To reference the stream we pass its FILE pointer as an argument to each standard I/O function. Throughout this text we'll refer to a pointer to a FILE object, the type FILE * as a *file pointer*.

Throughout this chapter we describe the standard I/O library in the context of a Unix-based system. As we mentioned, this library has already been ported to a wide variety of operating systems other than Unix. But to provide some insight about how this library can be implemented, we need to talk about its typical Unix implementation.

5.3 Standard Input, Standard Output, and Standard Error

Three streams are predefined and automatically available to a process: standard input, standard output, and standard error. These refer to the same files as the file descriptors STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO, which we mentioned in Section 3.2.

These three standard I/O streams are referenced through the predefined file pointers stdin, stdout, and stderr. These three file pointers are defined in the <stdio.h> header.

5.4 Buffering

The goal of the buffering provided by the standard I/O library is to use the minimum number of read and write calls. (Recall Figure 3.1 where we showed the amount of CPU time required to perform I/O using different buffer sizes.) Also, it tries to do its buffering automatically for each I/O stream, obviating the need for the application to worry about it. Unfortunately, the single aspect of the standard I/O library that generates the most confusion is its buffering.

There are three types of buffering provided.

1. Fully buffered. For this case actual I/O takes place when the standard I/O buffer is filled. Files that reside on disk are normally fully buffered by the standard I/O library. The buffer that's used is normally obtained by one of the standard I/O functions calling malloc (Section 7.8) the first time I/O is performed on a stream.

   The term *flush* describes the writing of a standard I/O buffer. A buffer can be flushed automatically by the standard I/O routines (such as when a buffer fills), or we can call the function fflush to flush a stream. Unfortunately, in the Unix environment *flush* means two different things. In terms of the standard I/O library it means writing out the contents of a buffer (which may be partially filled). In terms of the terminal driver (such as the tcflush function in Chapter 11) it means to discard the data that's already stored in a buffer.

2. Line buffered. In this case the standard I/O library performs I/O when a newline character is encountered on input or output. This allows us to output a single character at a time (with the standard I/O `fputc` function), knowing that actual I/O will take place only when we finish writing each line. Line buffering is typically used on a stream when it refers to a terminal (e.g., standard input and standard output).

   There are two caveats with respect to line buffering. First, since the size of the buffer that the standard I/O library is using to collect each line is fixed, actual I/O might take place if we fill this buffer before writing a newline. Second, whenever input is requested through the standard I/O library from either (a) an unbuffered stream or (b) from a line-buffered stream (that requires data to be requested from the kernel), it is intended that this causes *all* line-buffered output streams to be flushed. The reason for the qualifier on (b) is that the requested data may already be in the buffer, which doesn't require data to be read from the kernel. Obviously, any input from an unbuffered stream, item (a), requires data to be obtained from the kernel.

3. Unbuffered. The standard I/O library does not buffer the characters. If we write 15 characters with the standard I/O `fputs` function, for example, we expect these 15 characters to be output as soon as possible (probably with the `write` function from Section 3.8).

   The standard error stream, for example, is normally unbuffered. This is so that any error messages are displayed as quickly as possible, regardless whether they contain a newline or not.

ANSI C requires the following buffering characteristics:

1. Standard input and standard output are fully buffered, if and only if they do not refer to an interactive device.
2. Standard error is never fully buffered.

This, however, doesn't tell us whether standard input and standard output can be unbuffered or line buffered if they refer to an interactive device and whether standard error should be unbuffered or line buffered. Both SVR4 and 4.3+BSD default to the following types of buffering:

- Standard error is always unbuffered.
- All other streams are line buffered if they refer to a terminal device; otherwise they are fully buffered.

If we don't like these defaults for any given stream, we can change the buffering by calling either of the following two functions.

```
#include <stdio.h>

void setbuf(FILE *fp, char *buf);

int setvbuf(FILE *fp, char *buf, int mode, size_t size);
```

Returns: 0 if OK, nonzero on error

These functions must be called *after* the stream has been opened (obviously, since each requires a valid file pointer as their first argument) but *before* any other operation is performed on the stream.

With `setbuf` we can turn buffering on or off. To enable buffering, *buf* must point to a buffer of length `BUFSIZ` (a constant defined in `<stdio.h>`). Normally the stream is then fully buffered, but some systems may set line buffering if the stream is associated with a terminal device. To disable buffering, we set *buf* to `NULL`.

With `setvbuf` we specify exactly which type of buffering we want. This is done with the *mode* argument:

`_IOFBF`	fully buffered
`_IOLBF`	line buffered
`_IONBF`	unbuffered

If we specify an unbuffered stream, the *buf* and *size* arguments are ignored. If we specify fully buffered or line buffered, *buf* and *size* can optionally specify a buffer and its size. If the stream is buffered and *buf* is `NULL`, then the standard I/O library will automatically allocate its own buffer of the appropriate size for the stream. By appropriate size we mean the value specified by the `st_blksize` member of the `stat` structure from Section 4.2. If the system can't determine this value for the stream (if the stream refers to a device or a pipe, for example), then a buffer of length `BUFSIZ` is allocated.

> Using the `st_blksize` value for the buffer size came from the Berkeley systems. Earlier versions of System V used the standard I/O constant `BUFSIZ` (typically 1024). Even 4.3+BSD still sets `BUFSIZ` to 1024, even though it uses `st_blksize` to determine the optimal standard I/O buffer size.

Figure 5.1 summarizes the actions of these two functions and their various options.

Function	*mode*	*buf*	Buffer & length	Type of buffering
`setbuf`		nonnull	user *buf* of length `BUFSIZ`	fully buffered or line buffered
		`NULL`	(no buffer)	unbuffered
`setvbuf`	`_IOFBF`	nonnull	user *buf* of length *size*	fully buffered
		`NULL`	system buffer of appropriate length	
	`_IOLBF`	nonnull	user *buf* of length *size*	line buffered
		`NULL`	system buffer of appropriate length	
	`_IONBF`	(ignored)	(no buffer)	unbuffered

Figure 5.1 Summary of the `setbuf` and `setvbuf` functions.

Be aware that if we allocate a standard I/O buffer as an automatic variable within a function, we have to close the stream before returning from the function. (We'll discuss this more in Section 7.8.) Also, SVR4 uses part of the buffer for its own bookkeeping, so the actual number of bytes of data that can be stored in the buffer is less than *size*. In general, we should let the system choose the buffer size and automatically allocate the buffer. When we do this the standard I/O library automatically releases the buffer when we close the stream.

At any time we can force a stream to be flushed.

```
#include <stdio.h>

int fflush(FILE *fp);
```

Returns: 0 if OK, EOF on error

This function causes any unwritten data for the stream to be passed to the kernel. As a special case, if *fp* is NULL this function causes all output streams to be flushed.

> The ability to pass a null pointer to force all output streams to be flushed is new with ANSI C. Non-ANSI C libraries (e.g., earlier releases of System V and 4.3BSD) do not support this feature.

5.5 Opening a Stream

The following three functions open a standard I/O stream.

```
#include <stdio.h>

FILE *fopen(const char *pathname, const char *type);

FILE *freopen(const char *pathname, const char *type, FILE *fp);

FILE *fdopen(int filedes, const char *type);
```

All three return: file pointer if OK, NULL on error

The differences in these three functions are as follows:

1. fopen opens a specified file.
2. freopen opens a specified file on a specified stream, closing the stream first, if it is already open. This function is typically used to open a specified file as one of the predefined streams: standard input, standard output, or standard error.
3. fdopen takes an existing file descriptor (which we could obtain from the open, dup, dup2, fcntl, or pipe functions) and associates a standard I/O stream with the descriptor. This function is often used with descriptors that are

returned by the functions that create pipes and network communication channels. Since these special types of files cannot be opened with the standard I/O `fopen` function, we have to call the device-specific function to obtain a file descriptor, and then associate this descriptor with a standard I/O stream using `fdopen`.

`fopen` and `freopen` are part of ANSI C. `fdopen` is part of POSIX.1, since ANSI C doesn't deal with file descriptors.

ANSI C specifies 15 different values for the *type* argument, shown in Figure 5.2.

type	Description
`r` or `rb`	open for reading
`w` or `wb`	truncate to 0 length or create for writing
`a` or `ab`	append; open for writing at end of file, or create for writing
`r+` or `r+b` or `rb+`	open for reading and writing
`w+` or `w+b` or `wb+`	truncate to 0 length or create for reading and writing
`a+` or `a+b` or `ab+`	open or create for reading and writing at end of file

Figure 5.2 The *type* argument for opening a standard I/O stream.

Using the character `b` as part of the *type* allows the standard I/O system to differentiate between a text file and a binary file. Since the Unix kernel doesn't differentiate between these types of files, specifying the character `b` as part of the *type* has no effect.

With `fdopen` the meanings of the *type* argument differ slightly. Since the descriptor has already been opened, opening for write does not truncate the file. (If the descriptor was created by the `open` function, for example, and the file already existed, the `O_TRUNC` flag would determine if the file were truncated or not. The `fdopen` function cannot just truncate any file it opens for writing.) Also, the standard I/O append mode cannot create the file (since the file has to exist if a descriptor refers to it).

When a file is opened with a type of append, each write will take place at the then current end of file. If multiple processes open the same file with the standard I/O append mode, the data from each process will be correctly written to the file.

Versions of `fopen` from Berkeley before 4.3+BSD, and the simple version shown on page 177 of Kernighan and Ritchie [1988] do not handle the append mode correctly. These versions do an `lseek` to the end of file when the stream is opened. To correctly support the append mode when multiple processes are involved, the file must be opened with the `O_APPEND` flag, which we discussed in Section 3.3. Doing an `lseek` before each write won't work either, as we discussed in Section 3.11.

When a file is opened for reading and writing (the plus sign in the *type*) the following restrictions apply:

- Output cannot be directly followed by input without an intervening `fflush`, `fseek`, `fsetpos`, or `rewind`.
- Input cannot be directly followed by output without an intervening `fseek`, `fsetpos`, or `rewind`, or an input operation that encounters an end of file.

We can summarize the six different ways to open a stream from Figure 5.2 in Figure 5.3.

Restriction	r	w	a	r+	w+	a+
file must already exist	•			•		
previous contents of file discarded		•			•	
stream can be read	•			•	•	•
stream can be written		•	•	•	•	•
stream can be written only at end			•			•

Figure 5.3 Six different ways to open a standard I/O stream

Note that if a new file is created by specifying a *type* of either `w` or `a`, we are not able to specify the file's access permission bits (as we were able to do with the `open` function and the `creat` function in Chapter 3). POSIX.1 requires that the file be created with the following permissions

```
S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH
```

By default, the stream that is opened is fully buffered, unless it refers to a terminal device, in which case it is line buffered. Once the stream is opened, but before we do any other operation on the stream, we can change the buffering if we want to, with the `setbuf` or `setvbuf` functions from the previous section.

An open stream is closed by calling `fclose`.

```
#include <stdio.h>

int fclose(FILE *fp);
```

Returns: 0 if OK, EOF on error

Any buffered output data is flushed before the file is closed. Any input data that may be buffered is discarded. If the standard I/O library had automatically allocated a buffer for the stream, that buffer is released.

When a process terminates normally, either by calling the `exit` function directly, or by returning from the `main` function, all standard I/O streams with unwritten buffered data are flushed, and all open standard I/O streams are closed.

5.6 Reading and Writing a Stream

Once we open a stream we can choose among three different types of unformatted I/O. (We describe the formatted I/O functions, such as `printf` and `scanf`, in Section 5.11.)

1. Character-at-a-time I/O. We can read or write one character at a time, with the standard I/O functions handling all the buffering (if the stream is buffered).

2. Line-at-a-time I/O. If we want to read or write a line at a time, we use `fgets` and `fputs`. Each line is terminated with a newline character, and we have to specify the maximum line length that we can handle when we call `fgets`. We describe these two functions in Section 5.7.
3. Direct I/O. This type of I/O is supported by the `fread` and `fwrite` functions. For each I/O operation we read or write some number of objects, where each object is of a specified size. These two functions are often used for binary files where we read or write a structure with each operation. We describe these two functions in Section 5.9.

The term *direct I/O* is from the ANSI C standard. It's known by many names: binary I/O, object-at-a-time I/O, record-oriented I/O, or structure-oriented I/O.

Input Functions

Three functions allow us to read one character at a time.

```
#include <stdio.h>

int getc(FILE *fp);

int fgetc(FILE *fp);

int getchar(void);
```

All three return: next character if OK, EOF on end of file or error

The function `getchar` is defined to be equivalent to `getc(stdin)`. The difference between the first two functions is that `getc` can be implemented as a macro while `fgetc` cannot be implemented as a macro. This means three things:

1. The argument to `getc` should not be an expression with side effects.
2. Since `fgetc` is guaranteed to be a function, we can take its address. This allows us to pass the address of `fgetc` as an argument to another function.
3. Calls to `fgetc` probably take longer than calls to `getc`, since it usually takes more time to call a function. Indeed, examining most implementations of the `<stdio.h>` header shows that `getc` is a macro that has been coded for efficiency.

These three functions return the next character as an `unsigned char` converted to an `int`. The reason for specifying unsigned is so that the high-order bit, if set, doesn't cause the return value to be negative. The reason for requiring an integer return value is so that all possible character values can be returned, along with an indication that either an error occurred or the end of file has been encountered. The constant `EOF` in `<stdio.h>` is required to be a negative value. Its value is often –1. This representation

also means that we cannot store the return value from these three functions in a character variable, and compare this value later against the constant EOF.

Notice that these functions return the same value whether an error occurs or the end of file is reached. To distinguish between the two we must call either ferror or feof.

```
#include <stdio.h>

int ferror(FILE *fp);

int feof(FILE *fp);
```

Both return: nonzero (true) if condition is true, 0 (false) otherwise

```
void clearerr(FILE *fp);
```

In most implementations, two flags are maintained for each stream in the FILE object:

- an error flag,
- an end-of-file flag.

Both flags are cleared by calling clearerr.

After reading from a stream we can push back characters by calling ungetc.

```
#include <stdio.h>

int ungetc(int c, FILE *fp);
```

Returns: c if OK, EOF on error

The characters that are pushed back are returned by subsequent reads on the stream in reverse order of their pushing. Be aware, however, that although ANSI C allows an implementation to support any amount of pushback, an implementation is required to provide only a single character of pushback. We should not count on more than a single character.

The character that we push back does not have to be the same character that was read. We are not able to push back EOF. But when we've reached the end of file, we can push back a character. The next read will return that character, and the read after that will return EOF. This works because a successful call to ungetc clears the end-of-file indication for the stream.

Pushback is often used when we're reading an input stream and breaking into words or tokens of some form. Sometimes we need to peek at the next character to determine how to handle the current character. It's then easy to push back the character that we peeked at, for the next call to getc to return. If the standard I/O library didn't provide this pushback capability, we would have to store the character in a variable of our own, along with a flag telling us to use this character instead of calling getc the next time we need a character.

Output Functions

We'll find an output function that corresponds to each of the input functions that we've already described.

```
#include <stdio.h>

int putc(int c, FILE *fp);

int fputc(int c, FILE *fp);

int putchar(int c);
```

All three return: *c* if OK, EOF on error

Like the input functions, `putchar(c)` is equivalent to `putc(c,stdout)`, and `putc` can be implemented as a macro while `fputc` cannot be implemented as a macro.

5.7 Line-at-a-Time I/O

Line-at-a-time input is provided by the following two functions.

```
#include <stdio.h>

char *fgets(char *buf, int n, FILE *fp);

char *gets(char *buf);
```

Both return: `buf` if OK, NULL on end of file or error

Both specify the address of the buffer to read the line into. `gets` reads from standard input while `fgets` reads from the specified stream.

With `fgets` we have to specify the size of the buffer, *n*. This function reads up through and including the next newline, but no more than *n–1* characters, into the buffer. The buffer is terminated with a null byte. If the line, including the terminating newline, is longer than *n–1*, then only a partial line is returned, but the buffer is always null terminated. Another call to `fgets` will read what follows on the line.

`gets` is a deprecated function. The problem is that it doesn't allow the caller to specify the buffer size. This allows the buffer to overflow, if the line is longer than the buffer, writing over whatever happens to follow the buffer in memory. For a description of how this flaw was used as part of the Internet worm of 1988, see the June 1989 issue (vol. 32, no. 6) of *Communications of the ACM*. An additional difference with `gets` is that it doesn't store the newline in the buffer, as does `fgets`.

> This difference in newline handling between the two functions goes way back in the evolution of Unix. Even the Version 7 manual (1979) states "`gets` deletes a newline, `fgets` keeps it, all in the name of backward compatibility."

Even though ANSI C requires an implementation to provide gets, it should never be used.

Line-at-a-time output is provided by fputs and puts.

```
#include <stdio.h>

int fputs(const char *str, FILE *fp);

int puts(const char *str);
```

Both return: nonnegative value if OK, EOF on error

The function fputs writes the null terminated string to the specified stream. The null byte at the end is not written. Note that this need not be line-at-a-time output, since the string need not contain a newline as the last nonnull character. Usually this is the case (the last nonnull character is a newline), but it's not required.

puts writes the null terminated string to the standard output (without writing the null byte). But puts then writes a newline character to the standard output.

puts is not unsafe like its counterpart gets. Nevertheless, we'll avoid using it to prevent having to remember whether it appends a newline or not. If we always use fgets and fputs we know that we always have to deal with the newline character at the end of each line.

5.8 Standard I/O Efficiency

Using the functions from the previous section we can get an idea of the efficiency of the standard I/O system. Program 5.1 is like Program 3.3: it just copies standard input to standard output, using getc and putc. These two routines can be implemented as macros.

```
#include    "ourhdr.h"

int
main(void)
{
    int     c;

    while ( (c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

Program 5.1 Copy standard input to standard output using getc and putc.

We can make another version of this program that uses `fgetc` and `fputc`, which should be functions and not macros. (We don't show this trivial change to the source code.)

Finally we have a version that reads and writes lines, Program 5.2.

```
#include    "ourhdr.h"

int
main(void)
{
    char    buf[MAXLINE];

    while (fgets(buf, MAXLINE, stdin) != NULL)
        if (fputs(buf, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

Program 5.2 Copy standard input to standard output using `fgets` and `fputs`.

Note that we do not close the standard I/O streams explicitly in Program 5.1 or Program 5.2. Instead, we know that the `exit` function will flush any unwritten data and then close all open streams. (We'll discuss this in Section 8.5.) It is interesting to compare the timing of these three programs with the timing data from Figure 3.1. We show this data when operating on the same file (1.5 Mbytes with 30,000 lines) in Figure 5.4.

Function	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Bytes of program text
best time from Figure 3.1	0.0	0.3	0.3	
`fgets, fputs`	2.2	0.3	2.6	184
`getc, putc`	4.3	0.3	4.8	384
`fgetc, fputc`	4.6	0.3	5.0	152
single byte time from Figure 3.1	23.8	397.9	423.4	

Figure 5.4 Timing results using standard I/O routines.

For each of the three standard I/O versions, the user CPU time is larger than the best `read` version from Figure 3.1, because the character-at-a-time standard I/O versions have a loop that is executed 1.5 million times, and the loop in the line-at-a-time version is executed 30,000 times. In the `read` version, its loop is executed only 180 times (for a buffer size of 8192). This difference in user CPU times accounts for the difference in clock times, since the system CPU times are all the same.

The system CPU time is the same as before, because the same number of kernel requests are being made. Note that an advantage of using the standard I/O routines is that we don't have to worry about buffering or choosing the optimal I/O size. We do

have to determine the maximum line size for the version that uses `fgets`, but that's easier than trying to choose the optimal I/O size.

The final column in Figure 5.4 is the number of bytes of text space (the machine instructions generated by the C compiler) for each of the `main` functions. We can see that the version using `getc` expands the `getc` and `putc` macros inline, which takes more instructions than calling the `fgetc` and `fputc` functions. Looking at the difference in user CPU times between the `getc` version and the `fgetc` version, we see that expanding the macros inline versus calling two functions doesn't make a big difference on the system used for these tests.

The version using line-at-a-time I/O is almost twice as fast as the character-at-a-time version (both the user CPU time and the clock time). If the `fgets` and `fputs` functions are implemented using `getc` and `putc` (see Section 7.7 of Kernighan and Ritchie [1988], for example) then we would expect the timing to be similar to the `getc` version. Actually, we might expect the line-at-a-time version to take longer, since we would be adding 3 million macro invocations to the existing 60,000 function calls. What is happening with this example is that the line-at-a-time functions are implemented using `memccpy`(3). Often the `memccpy` function is implemented in assembler, instead of C, for efficiency.

The last point of interest with these timing numbers is that the `fgetc` version is so much faster than the `BUFFSIZE=1` version from Figure 3.1. Both involve the same number of function calls (about 3 million), yet the `fgetc` version is over 5 times faster in user CPU time, and almost 100 times faster in clock time. The difference is that the version using `read` executes 3 million function calls, which in turn execute 3 million system calls. With the `fgetc` version we still execute 3 million function calls but this ends up being only 360 system calls. System calls are usually much more expensive than ordinary function calls.

As a disclaimer you should be aware that these timing results are valid only on the single system they were run on. The results depend on many implementation features that aren't the same on every Unix system. Nevertheless, having a set of numbers such as these, and explaining why the various versions differ, helps us understand the system better. The basic fact that we've learned from this section and Section 3.9 is that the standard I/O library is not much slower than calling the `read` and `write` functions directly. The approximate cost that we've seen is about 3.0 seconds of CPU time to copy a megabyte of data using `getc` and `putc`. For most nontrivial applications, the largest amount of the user CPU time is taken by the application and not by the standard I/O routines.

5.9 Binary I/O

The functions from Section 5.6 operated with one character at a time or one line at a time. If we're doing binary I/O we often would like to read or write an entire structure at a time. To do this using `getc` or `putc` we have to loop through the entire structure, one byte at a time, reading or writing each byte. We can't use the line-at-a-time functions, since `fputs` stops writing when it hits a null byte, and there might be null bytes

within the structure. Similarly, `fgets` won't work right on input if any of the data bytes are nulls or newlines. Therefore, the following two functions are provided for binary I/O.

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nobj, FILE *fp);

size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);
```

Both return: number of objects read or written

There are two common uses for these functions.

1. Read or write a binary array. For example, to write elements 2 through 5 of a floating point array, we could write

```
float   data[10];

if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
    err_sys("fwrite error");
```

   Here we specify *size* as the size of each element of the array, and *nobj* as the number of elements.

2. Read or write a structure. For example, we could write

```
struct {
  short  count;
  long   total;
  char   name[NAMESIZE];
} item;

if (fwrite(&item, sizeof(item), 1, fp) != 1)
    err_sys("fwrite error");
```

   Here we specify *size* as the size of structure, and *nobj* as one (the number of objects to write).

The obvious generalization of these two cases is to read or write an array of structures. To do this *size* would be the `sizeof` the structure, and *nobj* would be the number of elements in the array.

Both `fread` and `fwrite` return the number of objects read or written. For the read case, this number can be less than *nobj* if an error occurs or if the end of file is encountered. In this case `ferror` or `feof` must be called. For the write case, if the return value is less than the requested *nobj*, an error has occurred.

A fundamental problem with binary I/O is that it can be used to read only data that has been written on the same system. While this was OK many years ago (when all the Unix systems were PDP-11s), today it is the norm to have heterogeneous systems connected together with networks. It is common to want to write data on one system and process it on another. These two functions won't work because

1. The offset of a member within a structure can differ between compilers and systems (due to different alignment requirements). Indeed, some compilers have an option allowing structures to be packed tightly (to save space with a possible run-time performance penalty) or aligned accurately to optimize run-time access of each member. This means even on a single system the binary layout of a structure can differ, depending on compiler options.
2. The binary formats used to store multibyte integers and floating-point values differs between different machine architectures.

The real solution for exchanging binary data between different systems is to use a higher level protocol. Refer to Section 18.2 of Stevens [1990] for a description of some techniques used by various network protocols to exchange binary data.

We'll return to the `fread` function in Section 8.13 when we'll use it to read a binary structure, the Unix process accounting records.

5.10 Positioning a Stream

There are two ways to position a standard I/O stream.

1. `ftell` and `fseek`. These two functions have been around since Version 7, but they assume that a file's position can be stored in a long integer.
2. `fgetpos` and `fsetpos`. These two functions are new with ANSI C. They introduce a new abstract data type, `fpos_t`, that records a file's position. Under non-Unix systems this data type can be made as big as necessary to record a file's position.

Portable applications that need to move to non-Unix systems should use `fgetpos` and `fsetpos`.

```
#include <stdio.h>

long ftell(FILE *fp);
```

Returns: current file position indicator if OK, –1L on error

```
int fseek(FILE *fp, long offset, int whence);
```

Returns: 0 if OK, nonzero on error

```
void rewind(FILE *fp);
```

For a binary file, a file's position indicator is measured in bytes from the beginning of the file. The value returned by `ftell` for a binary file is this byte position. To position a binary file using `fseek` we must specify a byte *offset* and how that offset is interpreted. The values for *whence* are the same as for the `lseek` function from Section 3.6:

SEEK_SET means from the beginning of the file, SEEK_CUR means from the current file position, and SEEK_END means from the end of file. ANSI C doesn't require an implementation to support the SEEK_END specification for a binary file, since some systems require a binary file to be padded at the end with zeroes to make the file size a multiple of some magic number. Under Unix, however, SEEK_END is supported for binary files.

For text files, the file's current position may not be measurable as a simple byte offset. Again, this is mainly under non-Unix systems that might store text files in a different format. To position a text file, *whence* has to be SEEK_SET and only two values for *offset* are allowed: 0 (meaning rewind the file to its beginning) or a value that was returned by ftell for that file. A stream can also be set to the beginning of the file with the rewind function.

As we mentioned, the following two functions are new with the C Standard.

```
#include <stdio.h>

int fgetpos(FILE *fp, fpos_t *pos);

int fsetpos(FILE *fp, const fpos_t *pos);
```

Both return: 0 if OK, nonzero on error

fgetpos stores the current value of the file's position indicator in the object pointed to by *pos*. This value can be used in a later call to fsetpos to reposition the stream to that location.

5.11 Formatted I/O

Formatted Output

Formatted output is handled by the three printf functions.

```
#include <stdio.h>

int printf(const char *format, ...);

int fprintf(FILE *fp, const char *format, ...);
```

Both return: number of characters output if OK, negative value if output error

```
int sprintf(char *buf, const char *format, ...);
```

Returns: number of characters stored in array

printf writes to the standard output, fprintf writes to the specified stream, and sprintf places the formatted characters in the array *buf*. sprintf automatically appends a null byte at the end of the array, but this null byte is not included in the return value.

4.3BSD defines `sprintf` as returning a `char *`, its first argument (the buffer pointer), instead of an integer. ANSI C requires that `sprintf` return an integer.

Note that it's possible for `sprintf` to overflow the buffer pointed to by *buf*. It's the caller's responsibility to assure the buffer is large enough.

We will not go through all the gory details of the different format conversions possible with these three functions. Refer to your local Unix manual or Appendix B of Kernighan and Ritchie [1988].

The following three variants of the `printf` family are similar to the previous three, but the variable argument list (`...`) is replaced with *arg*.

```
#include <stdarg.h>
#include <stdio.h>

int vprintf(const char *format, va_list arg);

int vfprintf(FILE *fp, const char *format, va_list arg);
```

Both return: number of characters output if OK, negative value if output error

```
int vsprintf(char *buf, const char *format, va_list arg);
```

Returns: number of characters stored in array

We use the `vsprintf` function in the error routines in Appendix B.

Refer to Section 7.3 of Kernighan and Ritchie [1988] for additional details on handling variable length argument lists with ANSI Standard C. Be aware that the variable length argument list routines provided with ANSI C (the `<stdarg.h>` header and its associated routines) differ from the `<varargs.h>` routines that were provided with SVR3 (and earlier) and 4.3BSD.

Formatted Input

Formatted input is handled by the three `scanf` functions.

```
#include <stdio.h>

int scanf(const char *format, ...);

int fscanf(FILE *fp, const char *format, ...);

int sscanf(const char *buf, const char *format, ...);
```

All three return: number of input items assigned, `EOF` if input error or end of file before any conversion

As with the `printf` family, refer to your Unix manual for all the details on the various format options.

5.12 Implementation Details

As we've mentioned, under Unix the standard I/O library ends up calling the I/O routines that we described in Chapter 3. Each standard I/O stream has an associated file descriptor, and we can obtain the descriptor for a stream by calling `fileno`.

```
#include <stdio.h>

int fileno(FILE *fp);
```

Returns: the file descriptor associated with the stream

We need this function if we want to call the dup or `fcntl` functions, for example.

To look at the implementation of the standard I/O library on your system, the place to start is with the header `<stdio.h>`. This will show how the `FILE` object is defined, the definitions of the per-stream flags, and any standard I/O routines that are defined as macros (such as `getc`). Section 8.5 of Kernighan and Ritchie [1988] has a sample implementation that shows the flavor of many Unix implementations. Chapter 12 of Plauger [1992] provides the complete source code for an implementation of the standard I/O library. The implementation of the standard I/O library in 4.3+BSD (written by Chris Torek) is also publicly available.

Example

Program 5.3 prints the buffering for the three standard streams and for a stream that is associated with a regular file. Note that we perform I/O on each stream before printing its buffering status, since the first I/O operation usually causes the buffers to be allocated for a stream. The structure members `_flag` and `_bufsiz` and the constants `_IONBF` and `_IOLBF` are defined by the system used by the author. Be aware that other Unix systems may have different implementations of the standard I/O library.

If we run Program 5.3 twice, once with the three standard streams connected to the terminal and once with the three standard streams redirected to files, the result is

```
$ a.out                                  stdin, stdout, and stderr connected to terminal
enter any character
                                         we type a newline
one line to standard error
stream = stdin, line buffered, buffer size = 128
stream = stdout, line buffered, buffer size = 128
stream = stderr, unbuffered, buffer size = 8
stream = /etc/motd, fully buffered, buffer size = 8192
$ a.out < /etc/termcap > std.out 2> std.err
                                         run it again with all three streams redirected
$ cat std.err
one line to standard error
$ cat std.out
enter any character
stream = stdin, fully buffered, buffer size = 8192
```

```
#include    "ourhdr.h"

void    pr_stdio(const char *, FILE *);

int
main(void)
{
    FILE    *fp;

    fputs("enter any character\n", stdout);
    if (getchar() == EOF)
        err_sys("getchar error");
    fputs("one line to standard error\n", stderr);

    pr_stdio("stdin",  stdin);
    pr_stdio("stdout", stdout);
    pr_stdio("stderr", stderr);

    if ( (fp = fopen("/etc/motd", "r")) == NULL)
        err_sys("fopen error");
    if (getc(fp) == EOF)
        err_sys("getc error");
    pr_stdio("/etc/motd", fp);
    exit(0);
}

void
pr_stdio(const char *name, FILE *fp)
{
    printf("stream = %s, ", name);
                /* following is nonportable */
    if      (fp->_flag & _IONBF)    printf("unbuffered");
    else if (fp->_flag & _IOLBF)    printf("line buffered");
    else /* if neither of above */  printf("fully buffered");
    printf(", buffer size = %d\n", fp->_bufsiz);
}
```

Program 5.3 Print buffering for various standard I/O streams.

```
stream = stdout, fully buffered, buffer size = 8192
stream = stderr, unbuffered, buffer size = 8
stream = /etc/motd, fully buffered, buffer size = 8192
```

We can see that the default for this system is to have standard input and standard output line buffered when they're connected to a terminal. The line buffer is 128 bytes. Note that this doesn't restrict us to 128-byte input and output lines, that's just the size of the buffer. Writing a 512-byte line to standard output will require four `write` system calls. When we redirect these two streams to regular files they become fully buffered, with buffer sizes equal to the preferred I/O size (the `st_blksize` value from the `stat` structure) for the filesystem. We also see that the standard error is always unbuffered (as it should be) and that a regular file defaults to fully buffered. □

5.13 Temporary Files

The standard I/O library provides two functions to assist in creating temporary files.

```
#include <stdio.h>

char *tmpnam(char *ptr);

                                        Returns: pointer to unique pathname

FILE *tmpfile(void);

                                    Returns: file pointer if OK, NULL on error
```

`tmpnam` generates a string that is a valid pathname and that is not the same name as an existing file. It generates a different pathname each time it is called, up to `TMP_MAX` times. `TMP_MAX` is defined in `<stdio.h>`.

> Although `TMP_MAX` is defined by ANSI C, the C standard requires only that its value be at least 25. XPG3, however, requires that its value be at least 10,000. While this minimum value allows an implementation to use four digits (0000–9999), most Unix implementations use lowercase or uppercase characters.

If *ptr* is `NULL`, the generated pathname is stored in a static area, and a pointer to this static area is returned as the value of the function. Subsequent calls to `tmpnam` can overwrite this static area. (This means if we call this function more than once and we want to save the pathname, we have to save a copy of the pathname, not a copy of the pointer.) If *ptr* is not `NULL`, it is assumed that it points to an array of at least `L_tmpnam` characters. (The constant `L_tmpnam` is defined in `<stdio.h>`.) The generated pathname is stored in this array, and *ptr* is also returned as the value of the function.

`tmpfile` creates a temporary binary file (type `wb+`) that is automatically removed when it is closed or on program termination. The fact that this file is a binary file makes no difference under Unix.

Example

Program 5.4 demonstrates these two functions. If we execute Program 5.4 we get

```
$ a.out
/usr/tmp/aaaa00470
/usr/tmp/baaa00470
one line of output
```

The five-digit suffix added to each of the temporary names is the process ID. This is how the generated pathnames are known to be unique to each process that may call `tmpnam`. □

The standard Unix technique often used by the `tmpfile` function is to create a unique pathname by calling `tmpnam`, then create the file, and immediately `unlink` it.

```
#include    "ourhdr.h"

int
main(void)
{
    char    name[L_tmpnam], line[MAXLINE];
    FILE    *fp;

    printf("%s\n", tmpnam(NULL));       /* first temp name */

    tmpnam(name);                       /* second temp name */
    printf("%s\n", name);

    if ( (fp = tmpfile()) == NULL)      /* create temp file */
        err_sys("tmpfile error");
    fputs("one line of output\n", fp);  /* write to temp file */
    rewind(fp);                         /* then read it back */
    if (fgets(line, sizeof(line), fp) == NULL)
        err_sys("fgets error");
    fputs(line, stdout);                /* print the line we wrote */

    exit(0);
}
```

Program 5.4 Demonstrate `tmpnam` and `tmpfile` functions.

Recall from Section 4.15 that unlinking a file does not delete its contents until the file is closed. This way, when the file is closed, either explicitly or on program termination, the contents of the file are deleted.

`tempnam` is a variation of `tmpnam` that allows the caller to specify both the directory and a prefix for the generated pathname.

```
#include <stdio.h>

char *tempnam(const char *directory, const char *prefix);
```

Returns: pointer to unique pathname

There are four different choices for the directory, and the first one that is true is used:

1. If the environment variable `TMPDIR` is defined, it is used as the directory. (We describe environment variables in Section 7.9.)
2. If *directory* is not `NULL`, it is used as the directory.
3. The string `P_tmpdir` in `<stdio.h>` is used as the directory.
4. A local directory, usually `/tmp` is used as the directory.

If the *prefix* argument is not NULL, it should be a string of up to five characters to be used as the first characters of the filename.

This function calls the malloc function to allocate dynamic storage for the constructed pathname. We can free this storage when we're done with the pathname. (We describe the malloc and free functions in Section 7.8.)

Although tempnam is not part of POSIX.1 or ANSI C, it is part of XPG3.

The implementation that we've described corresponds to SVR4 and 4.3+BSD. The XPG3 version is identical, except the XPG3 version does not support the TMPDIR environment variable.

Example

Program 5.5 shows use of tempnam.

```
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    if (argc != 3)
        err_quit("usage: a.out <directory> <prefix>");

    printf("%s\n", tempnam( argv[1][0] != ' ' ? argv[1] : NULL,
                            argv[2][0] != ' ' ? argv[2] : NULL) );
    exit(0);
}
```

Program 5.5 Demonstrate tempnam function.

Note that if either command-line argument (the directory or the prefix) begins with a blank, we pass a null pointer to the function. We can now show the various ways to use it:

```
$ a.out /home/stevens TEMP    specify both directory and prefix
/home/stevens/TEMPAAAa00571
$ a.out " " PFX               use default directory: P_tmpdir
/usr/tmp/PFXAAAa00572
$ TMPDIR=/tmp a.out /usr/tmp " "    use environment variable; no prefix
/tmp/AAAa00573                environment variable overrides directory
$ TMPDIR=/no/such/dir a.out /tmp QQQQ
/tmp/QQQQAAAa00574            invalid environment directory is ignored
$ TMPDIR=/no/such/file a.out /etc/uucp MMMMM
/usr/tmp/MMMMMAAAa00575       invalid environment; invalid directory; both ignored
```

As the four steps that we listed earlier for specifying the directory name are tried in order, this function also checks if the corresponding directory name makes sense. If the directory doesn't exist (the /no/such/dir example) or if we don't have write permission in the directory (the /etc/uucp example), that case is skipped and the next choice

for the directory name is tried. From this example we can see how the process ID is used as part of the pathname, and we can also see that for this implementation the `P_tmpdir` directory is `/usr/tmp`. The technique that we used to set the environment variable, specifying `TMPDIR=` before the program name, is used by both the Bourne shell and the KornShell. □

5.14 Alternatives to Standard I/O

The standard I/O library is not perfect. Korn and Vo [1991] list numerous defects—some in the basic design, but most in the various, different implementations.

One inefficiency inherent in the standard I/O library is the amount of data copying that takes place. When we use the line-at-a-time functions, `fgets` and `fputs`, the data is usually copied twice: once between the kernel and the standard I/O buffer (when the corresponding `read` or `write` is issued) and again between the standard I/O buffer and our line buffer. The Fast I/O library [`fio`(3) in AT&T 1990a] gets around this by having the function that reads a line return a pointer to the line instead of copying the line into another buffer. Hume [1988] reports a threefold increase in the speed of a version of the `grep`(1) utility, just by making this change.

Korn and Vo [1991] describe another replacement for the standard I/O library: *sfio*. This package is similar in speed to the *fio* library and normally faster than the standard I/O library. *sfio* also provides some new features that aren't in the others: I/O streams are generalized to represent both files and regions of memory, processing modules can be written and stacked on an I/O stream to change the operation of a stream, and better exception handling.

Krieger, Stumm, and Unrau [1992] describe another alternative that uses mapped files—the `mmap` function that we describe in Section 12.9. This new package is called ASI, the Alloc Stream Interface. The programming interface resembles the Unix memory allocation functions (`malloc`, `realloc`, and `free`, described in Section 7.8). As with the *sfio* package, ASI attempts to minimize the amount of data copying by using pointers.

5.15 Summary

The standard I/O library is used by most Unix applications. We have looked at all the functions provided by this library, and some implementation details and efficiency considerations. Be aware of the buffering that takes place with this library, as this is the area that generates the most problems and confusion.

Exercises

5.1 Implement `setbuf` using `setvbuf`.

5.2 Type in the program that copies a file using line-at-a-time I/O (`fgets` and `fputs`) from Section 5.8, but use a `MAXLINE` of 4. What happens if you copy lines that exceed this length? Explain what is happening.

5.3 What does a return value of 0 from `printf` mean?

5.4 The following code works correctly on some machines, but not on others. What could be the problem?

```
#include    <stdio.h>

int
main(void)
{
    char    c;

    while ( (c = getchar()) != EOF )
        putchar(c);
}
```

5.5 Why does `tempnam` restrict the *prefix* to five characters?

5.6 How would you use the `fsync` function (Section 4.24) with a standard I/O stream?

5.7 In Programs 1.5 and 1.8 the prompt that is printed does not contain a newline and we don't call `fflush`. What causes the prompt to be output?

6

System Data Files and Information

6.1 Introduction

There are numerous data files required for normal operation: the password file `/etc/passwd` and the group file `/etc/group` are two files that are frequently used by various programs. For example, the password file is used every time a user logs in to a Unix system and every time someone executes an `ls -l` command.

Historically, these data files have been ASCII text files and were read with the standard I/O library. But for larger systems a sequential scan through the password file becomes time consuming. We want to be able to store these data files in a format other than ASCII text, but still provide an interface for an application program that works with any file format. The portable interfaces to these data files are the subject of this chapter. We also cover the system identification functions and the time and date functions.

6.2 Password File

The Unix password file, called the user database by POSIX.1, contains the fields shown in Figure 6.1. These fields are contained in a `passwd` structure that is defined in `<pwd.h>`.

> Note that POSIX.1 specifies only five of the seven fields in the `passwd` structure. The other two elements are supported by both SVR4 and 4.3+BSD.

Description	struct passwd member		POSIX.1
user name	char	*pw_name	•
encrypted password	char	*pw_passwd	
numerical user ID	uid_t	pw_uid	•
numerical group ID	gid_t	pw_gid	•
comment field	char	*pw_gecos	
initial working directory	char	*pw_dir	•
initial shell (user program)	char	*pw_shell	•

Figure 6.1 Fields in `/etc/passwd` file.

Historically the password file has been stored in `/etc/passwd` and has been an ASCII file. Each line contains the seven fields described in Figure 6.1, separated by colons. For example, three lines from the file could be

```
root:jheVopR58x9Fx:0:1:The superuser:/:/bin/sh
nobody:*:65534:65534::/:
stevens:3hKVD8R58r9Fx:224:20:Richard Stevens:/home/stevens:/bin/ksh
```

Note the following points about these entries.

- There is usually an entry with the user name `root`. This entry has a user ID of 0 (the superuser).
- The encrypted password field contains a copy of the user's password that has been put through a one-way encryption algorithm. Since this algorithm is one-way, we can't guess the original password from the encrypted version. The algorithm that is currently used (see Morris and Thompson [1979]) always generates 13 printable characters from the 64-character set `[a-zA-z0-9./]`. Since the entry for the user name `nobody` contains a single character, an encrypted password will never match this value. This user name can be used by network servers that allow us to log in to a system, but with a user ID and group ID (65534) that provide no privileges. The only files that we can access with this user ID and group ID are those that are readable or writable by the world. (This assumes that there are no files specifically owned by used ID 65534 or group ID 65534, which should be the case.) We'll discuss a recent change to the password file (shadow passwords) later in this section.
- Some fields in a password file entry can be empty. If the encrypted password field is empty, it usually means the user does not have a password. (This is not recommended.) The entry for `nobody` has two blank fields: the comment field and the initial shell field. An empty comment field has no effect. The default value for an empty shell field is usually `/bin/sh`.
- Some Unix systems that support the `finger`(1) command support additional information in the comment field. Each of these fields are separated by a

comma: the user's name, office location, office phone number, and home phone number. Additionally, if the user's name in the comment field is an ampersand, it is replaced with the login name (capitalized). For example, we could have

```
stevens:3hKVD8R58r9Fx:224:20:Richard &, B232, 555-1111, 555-2222:/home/stevens:/bin/ksh
```

Even if your system doesn't support the `finger` command, these fields can still go into the comment field, since that field is just a comment and not interpreted by system utilities.

POSIX.1 defines only two functions to fetch entries from the password file. These two functions allow us to look up an entry given a user's login name or numerical user ID.

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);

struct passwd *getpwnam(const char *name);
```

Both return: pointer if OK, NULL on error

`getpwuid` is used by the `ls`(1) program, to map the numerical user ID contained in an i-node into a user's login name. `getpwnam` is used by the `login`(1) program, when we enter our login name.

Both functions return a pointer to a `passwd` structure that the functions fill in. This structure is usually a `static` variable within the function, so its contents are overwritten each time we call either of these functions.

These two POSIX.1 functions are fine if we want to look up either a login name or a user ID, but there are programs that need to go through the entire password file. The following three functions can be used for this.

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwent(void);
```

Returns: pointer if OK, NULL on error or end of file

```
void setpwent(void);

void endpwent(void);
```

While not part of POSIX.1, these three functions are supported by SVR4 and 4.3+BSD.

We call `getpwent` to return the next entry in the password file. As with the two POSIX.1 functions, `getpwent` returns a pointer to a structure that it has filled in. This

structure is normally overwritten each time we call this function. If this is the first call to this function, it opens whatever files it uses. There is no order implied when we use this function—the entries can be in any order. (This is because some systems use a hashed version of the file `/etc/passwd`.)

The function `setpwent` rewinds whatever files it uses, and `endpwent` closes these files. When using `getpwent` we must always be sure to close these files by calling `endpwent` when we're through. `getpwent` is smart enough to know when it has to open its files (the first time we call it) but it never knows when we're through.

Example

Program 6.1 shows an implementation of the function `getpwnam`.

```
#include <sys/types.h>
#include <pwd.h>
#include <stddef.h>
#include <string.h>

struct passwd *
getpwnam(const char *name)
{
    struct passwd  *ptr;

    setpwent();
    while ( (ptr = getpwent()) != NULL) {
        if (strcmp(name, ptr->pw_name) == 0)
            break;       /* found a match */
    }
    endpwent();
    return(ptr);     /* ptr is NULL if no match found */
}
```

Program 6.1 The `getpwnam` function.

The call to `setpwent` at the beginning is self defense—we assure that the files are rewound, in case the caller has already opened them by calling `getpwent`. The call to `endpwent` when we're done is because neither `getpwnam` nor `getpwuid` should leave any of the files open. □

6.3 Shadow Passwords

We mentioned in the previous section that the encryption algorithm normally used for Unix passwords is a one-way algorithm. Given an encrypted password, we can't apply an algorithm that inverts it and returns the plaintext password. (The plaintext password is what we enter at the `Password:` prompt.) But we could guess a password, run it through the one-way algorithm, and compare the result with the encrypted password.

If user passwords were randomly chosen, this brute force approach wouldn't be too useful. Users, however, tend to choose nonrandom passwords (spouse's name, street names, pet names, etc.). A frequently repeated experiment is for someone to obtain a copy of the password file and try guessing the passwords. (Chapter 2 of Garfinkel and Spafford [1991] contains additional details and history on Unix passwords and the password encryption scheme.)

To make it harder to obtain the raw materials (the encrypted passwords), some systems store the encrypted password in another file, often called the *shadow password file*. Minimally this file has to contain the user name and the encrypted password. Other information relating to the password is also stored here. For example, systems with shadow passwords often require the user to choose a new password at certain intervals. This is called password aging, and the time between having to choose new passwords is often stored in the shadow password file also.

In SVR4 the shadow password file is `/etc/shadow`. In 4.3+BSD the encrypted passwords are stored in `/etc/master.passwd`.

The shadow password file should not be readable by the world. Only a few programs need to access encrypted passwords, `login`(1) and `passwd`(1), for example, and these programs are often set-user-ID root. With shadow passwords the regular password file, `/etc/passwd`, can be left readable by the world.

6.4 Group File

The Unix group file, called the group database by POSIX.1, contains the fields shown in Figure 6.2. These fields are contained in a `group` structure that is defined in `<grp.h>`.

Description	`struct group` member		POSIX.1
group name	`char`	`*gr_name`	•
encrypted password	`char`	`*gr_passwd`	
numerical group ID	`int`	`gr_gid`	•
array of pointers to individual user names	`char`	`**gr_mem`	•

Figure 6.2 Fields in `/etc/group` file.

POSIX.1 defines only three of the four fields. The other field, `gr_passwd`, is supported by both SVR4 and 4.3+BSD.

The field `gr_mem` is an array of pointers to the user names that belong to this group. This array is terminated by a null pointer.

We can look up either a group name or a numerical group ID with the following two functions, which are defined by POSIX.1.

```
#include <sys/types.h>
#include <grp.h>

struct group *getgrgid(gid_t gid);

struct group *getgrnam(const char *name);
```

Both return: pointer if OK, NULL on error

As with the password file functions, both of these functions normally return pointers to a `static` variable, which is overwritten on each call.

If we want to search the entire group file we need some additional functions. The following three functions are like their counterparts for the password file.

```
#include <sys/types.h>
#include <grp.h>

struct group *getgrent(void);
```

Returns: pointer if OK, NULL on error or end of file

```
void setgrent(void);

void endgrent(void);
```

These three functions are provided by SVR4 and 4.3+BSD. They are not part of POSIX.1.

`setgrent` opens the group file (if it's not already open) and rewinds it. `getgrent` reads the next entry from the group file, opening the file first, if it's not already open. `endgrent` closes the group file.

6.5 Supplementary Group IDs

The use of groups in Unix has changed over time. With Version 7 each user belonged to a single group at any point in time. When we logged in we were assigned the real group ID corresponding to the numerical group ID in our password file entry. We could change this at any point by executing `newgrp`(1). If the `newgrp` command succeeded (refer to the manual page for the permission rules), our real group ID was changed to the new group's ID, and this was used for all subsequent file access permission checks. We could always go back to our original group by executing `newgrp` without any arguments.

This form of group membership persisted until it was changed in 4.2BSD (circa 1983). With 4.2BSD the concept of supplementary group IDs was introduced. Not only did we belong to the group corresponding to the group ID in our password file entry, we could also belong to up to 16 additional groups. The file access permission checks

were modified so that not only was the effective group ID compared to the file's group ID, but all the supplementary group IDs were also compared to the file's group ID.

Supplementary group IDs are an optional feature of POSIX.1. The constant NGROUPS_MAX (Figure 2.7) specifies the number of supplementary group IDs. A common value is 16. This constant is 0 if supplementary group IDs aren't supported.

Both SVR4 and 4.3+BSD support supplementary group IDs.

FIPS 151-1 requires the support of supplementary group IDs and requires that NGROUPS_MAX be at least 8.

The advantage in using supplementary group IDs is that we no longer have to change groups explicitly. It is not uncommon to belong to multiple groups (i.e., participate in multiple projects) at the same time.

Three functions are provided to fetch and set the supplementary group IDs.

```
#include <sys/types.h>
#include <unistd.h>

int getgroups(int gidsetsize, gid_t grouplist[]);
```

Returns: number of supplementary group IDs if OK, –1 on error

```
int setgroups(int ngroups, const gid_t grouplist[]);

int initgroups(const char *username, gid_t basegid);
```

Both return: 0 if OK, –1 on error

Of these three functions, only `getgroups` is specified by POSIX.1. Since `setgroups` and `initgroups` are privileged operations, they are not part of POSIX.1. SVR4 and 4.3+BSD, however, support all three functions.

`getgroups` fills in the array *grouplist* with the supplementary group IDs. Up to *gidsetsize* elements are stored in the array. The number of supplementary group IDs stored in the array is returned by the function. If the system constant NGROUPS_MAX is 0, the function returns 0, not an error.

As a special case, if *gidsetsize* is 0, the function returns only the number of supplementary group IDs. The array *grouplist* is not modified. (This allows the caller to determine the size of the *grouplist* array to allocate.)

`setgroups` can be called by the superuser to set the supplementary group ID list for the calling process. *grouplist* contains the array of group IDs, and *ngroups* specifies the number of elements in the array.

The only use of `setgroups` is usually from the `initgroups` function, which reads the entire group file (with the functions `getgrent`, `setgrent`, and `endgrent`, which we described earlier) and determines the group membership for *username*. It then calls `setgroups` to initialize the supplementary group ID list for the user. One must be superuser to call `initgroups` since it calls `setgroups`. In addition to finding all the groups that *username* is a member of in the group file, `initgroups` also includes

basegid in the supplementary group ID list. *basegid* is the group ID from the password file for *username*.

`initgroups` is called by only a few programs—the `login`(1) program, for example, calls it when we log in.

6.6 Other Data Files

We've discussed only two of the system's data files so far—the password file and the group file. Numerous other files are used by Unix systems in normal day-to-day operation. For example, the BSD networking software has one data file for the services provided by the various network servers (`/etc/services`), one for the protocols (`/etc/protocols`), and one for the networks (`/etc/networks`). Fortunately the interfaces to these various files are like the ones we've already described for the password and group files.

The general principle is that every data file has at least three functions:

1. A `get` function that reads the next record, opening the file if necessary. These functions normally return a pointer to a structure. A null pointer is returned when the end of the data file is reached. Most of the `get` functions return a pointer to a `static` structure, so we always have to copy it if we want to save it.
2. A `set` function that opens the file (if not already open) and rewinds the file. This function is used when we know we want to start again at the beginning of the file.
3. An `end` entry that closes the data file. As we mentioned earlier, we always have to call this when we're done, to close all the files.

Additionally, if the data file supports some form of keyed lookup, routines are provided to search for a record with a specific key. For example, two keyed lookup routines are provided for the password file: `getpwnam` looks for a record with a specific user name, and `getpwuid` looks for a record with a specific user ID.

Figure 6.3 shows some of these routines, which are common to both SVR4 and 4.3+BSD. In this figure we show the functions for the password file and group file, which we discussed earlier in this chapter, and some of the networking functions. There are `get`, `set`, and `end` functions for all the data files in this figure.

Under SVR4 the last four data files in Figure 6.3 are symbolic links to files of the same name in the directory `/etc/inet`.

Both SVR4 and 4.3+BSD have additional functions that are like these, but the additional functions tend to deal with system administration files and are specific to each implementation.

Description	Data file	Header	Structure	Additional keyed lookup functions
passwords	/etc/passwd	<pwd.h>	passwd	getpwnam, getpwuid
groups	/etc/group	<grp.h>	group	getgrnam, getgrgid
hosts	/etc/hosts	<netdb.h>	hostent	gethostbyname, gethostbyaddr
networks	/etc/networks	<netdb.h>	netent	getnetbyname, getnetbyaddr
protocols	/etc/protocols	<netdb.h>	protoent	getprotobyname, getprotobynumber
services	/etc/services	<netdb.h>	servent	getservbyname, getservbyport

Figure 6.3 Similar routines for accessing system data files.

6.7 Login Accounting

Two data files that have been provided with most Unix systems are the utmp file, which keeps track of all the users currently logged in, and the wtmp file, which keeps track of all logins and logouts.

With Version 7, one type of record was written to both files, a binary record consisting of the following structure:

```
struct utmp {
  char  ut_line[8]; /* tty line: "ttyh0", "ttyd0", "ttyp0", ... */
  char  ut_name[8]; /* login name */
  long  ut_time;    /* seconds since Epoch */
};
```

On login, one of these structures was filled in and written to the utmp file by the login program, and the same structure was appended to the wtmp file. On logout, the entry in the utmp file was erased (filled with 0 bytes) by the init process, and a new entry was appended to the wtmp file. This logout entry in the wtmp file had the ut_name field zeroed out. Special entries were appended to the wtmp file to indicate when the system was rebooted and right before and after the system's time and date was changed. The who(1) program read the utmp file and printed its contents in a readable form. Later versions of Unix provided the last(1) command, which read through the wtmp file and printed selected entries.

Most versions of Unix still provide the utmp and wtmp files, but as expected, the amount of the information in these files has grown. The 20-byte structure that was written by Version 7 grew to 36 bytes with SVR2, and the extended utmp structure with SVR4 takes over 350 bytes!

> The detailed format of these records in SVR4 is given in the utmp(4) and utmpx(4) manual pages. With SVR4 both files are in the /var/adm directory. SVR4 provides numerous functions described in getut(3) and getutx(3) to read and write these two files.
>
> The 4.3+BSD utmp(5) manual page gives the format of its version of these login records. The pathnames of these two files are /var/run/utmp and /var/log/wtmp.

6.8 System Identification

POSIX.1 defines the uname function to return information on the current host and operating system.

```
#include <sys/utsname.h>

int uname(struct utsname *name);
```

Returns: nonnegative value if OK, –1 on error

We pass the address of a utsname structure, and the function fills it in. POSIX.1 defines only the minimum fields in the structure (which are all character arrays), and it's up to each implementation to set the size of each array. Some implementations provide additional fields in the structure. Historically, System V has allocated 8 bytes for each element, with room for a null byte at the end.

```
struct utsname {
  char  sysname[9];    /* name of the operating system */
  char  nodename[9];   /* name of this node */
  char  release[9];    /* current release of operating system */
  char  version[9];    /* current version of this release */
  char  machine[9];    /* name of hardware type */
};
```

The information in the utsname structure can usually be printed with the uname(1) command.

> POSIX.1 warns that the nodename element may not be adequate to reference the host on a communications network. This function is from System V, and in older days the nodename element was adequate for referencing the host on a UUCP network.
>
> Realize also that the information in this structure does not give any information on the POSIX.1 level. This should be obtained using _POSIX_VERSION as described in Section 2.5.2.
>
> Finally, this function gives us a way only to fetch the information in the structure—there is nothing specified by POSIX.1 about initializing this information. Most versions of System V have this information compiled into the kernel when the kernel is built.

Berkeley-derived systems provide the gethostname function to return just the name of the host. This name is usually the name of the host on a TCP/IP network.

```
#include <unistd.h>

int gethostname(char *name, int namelen);
```

Returns: 0 if OK, –1 on error

The string returned through *name* is null terminated, unless insufficient room is provided. The constant MAXHOSTNAMELEN in <sys/param.h> specifies the maximum

length of this name (normally 64 bytes). If the host is connected to a TCP/IP network, the host name is normally the fully qualified domain name of the host.

There is also a `hostname`(1) command that can fetch or set the host name. (The host name is set by the superuser using a similar function, `sethostname`.) The host name is normally set at bootstrap time from one of the start-up files invoked by `/etc/rc`.

> Although this function is Berkeley-specific, SVR4 provides the `gethostname` and `sethostname` functions, and the `hostname` command as part of the BSD compatibility package. SVR4 also extends `MAXHOSTNAMELEN` to 256 bytes.

6.9 Time and Date Routines

The basic time service provided by the Unix kernel is to count the number of seconds that have passed since the Epoch: 00:00:00 January 1, 1970, UTC. In Section 1.10 we said that these seconds are represented in a `time_t` data type, and we call them *calendar times*. These calendar times represent both the time and date. Unix has always differed from other operating systems in (a) keeping time in UTC instead of the local time, (b) automatically handling conversions such as daylight saving time, and (c) keeping the time and date as a single quantity.

The `time` function returns the current time and date.

```
#include <time.h>

time_t time(time_t *calptr);
```

Returns: value of time if OK, –1 on error

The time value is always returned as the value of the function. If the argument is nonnull, the time value is also stored at the location pointed to by *calptr*.

> In many Berkeley-derived systems `time`(3) is just a function that invokes the `gettimeofday`(2) system call.
>
> We haven't said how the kernel's notion of the current time is initialized. Under SVR4 the `stime`(2) function is called, while Berkeley-derived systems use `settimeofday`(2).
>
> The BSD `gettimeofday` and `settimeofday` functions provide greater resolution (up to a microsecond) than the `time` and `stime` functions. This is important for some applications.

Once we have this large integer value that counts the number of seconds since the Epoch, we normally call one of the other time functions to convert it to a human-readable time and date. Figure 6.4 shows the relationships between the various time functions. (The four functions in this figure that are shown with dashed lines, `localtime`, `mktime`, `ctime`, and `strftime`, are all affected by the `TZ` environment variable, which we describe later in this section.)

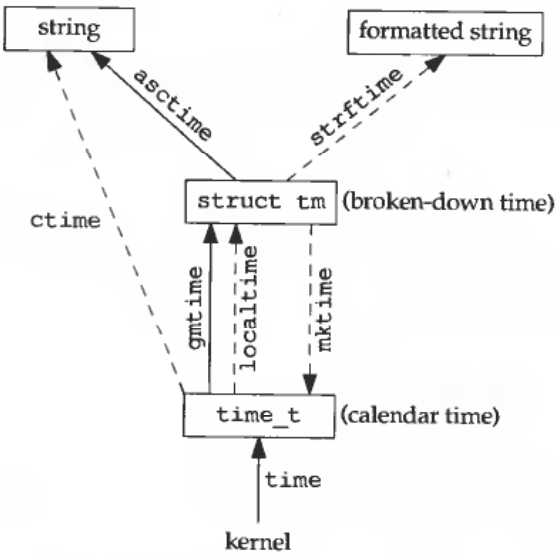


Figure 6.4 Relationship of the various time functions.

The two functions `localtime` and `gmtime` convert a calendar time into what's called a broken-down time, a `tm` structure.

```
struct tm {  /* a broken-down time */
  int  tm_sec;   /* seconds after the minute: [0, 61] */
  int  tm_min;   /* minutes after the hour: [0, 59] */
  int  tm_hour;  /* hours after midnight: [0, 23] */
  int  tm_mday;  /* day of the month: [1, 31] */
  int  tm_mon;   /* month of the year: [0, 11] */
  int  tm_year;  /* years since 1900 */
  int  tm_wday;  /* days since Sunday: [0, 6] */
  int  tm_yday;  /* days since January 1: [0, 365] */
  int  tm_isdst; /* daylight saving time flag: <0, 0, >0 */
};
```

The reason that the seconds can be greater than 59 is to allow for leap seconds. Notice that all the fields except the day of the month are 0-based. The daylight saving time flag is positive if daylight saving time is in effect, 0 if it's not in effect, and negative if the information isn't available.

```
#include <time.h>

struct tm *gmtime(const time_t *calptr);

struct tm *localtime(const time_t *calptr);
```

Both return: pointer to broken-down time

The difference between `localtime` and `gmtime` is that the first converts the calendar time to the local time (taking into account the local time zone and daylight saving time flag), while the latter converts the calendar time into a broken-down time expressed as UTC.

The function `mktime` takes a broken-down time (expressed as a local time) and converts it into a `time_t` value.

```
#include <time.h>

time_t mktime(struct tm *tmptr);
```

Returns: calendar time if OK, –1 on error

The `asctime` and `ctime` functions produce the familiar 26-byte string that is similar to the default output of the `date`(1) command:

```
Tue Jan 14 17:49:03 1992\n\0
```

```
#include <time.h>

char *asctime(const struct tm *tmptr);

char *ctime(const time_t *calptr);
```

Both return: pointer to null terminated string

The argument to `asctime` is a pointer to a broken-down string, while the argument to `ctime` is a pointer to a calendar time.

The final time function is the most complicated. `strftime` is a `printf`-like function for time values.

```
#include <time.h>

size_t strftime(char *buf, size_t maxsize, const char *format,
                const struct tm *tmptr);
```

Returns: number of characters stored in array if room, else 0

The final argument is the time value to format, specified by a pointer to a broken-down time value. The formatted result is stored in the array *buf* whose size is *maxsize* characters. If the size of the result, including the terminating null, fits in the buffer, the function returns the number of characters stored in *buf* (excluding the terminating null). Otherwise the function returns 0.

The *format* argument controls the formatting of the time value. Like the `printf` functions, conversion specifiers are given as a percent followed by a special character. All other characters in the *format* string are copied to the output. Two percents in a row generate a single percent in the output. Unlike the `printf` functions, each conversion

specified generates a fixed size output string—there are no field widths in the *format* string. Figure 6.5 describes the 21 different ANSI C conversion specifiers.

Format	Description	Example
%a	abbreviated weekday name	Tue
%A	full weekday name	Tuesday
%b	abbreviated month name	Jan
%B	full month name	January
%c	date and time	Tue Jan 14 19:40:30 1992
%d	day of the month: [01, 31]	14
%H	hour of the 24-hour day: [00, 23]	19
%I	hour of the 24-hour day: [01, 12]	07
%j	day of the year: [001, 366]	014
%m	month: [01, 12]	01
%M	minute: [00, 59]	40
%p	AM/PM	PM
%S	second: [00, 61]	30
%U	Sunday week number: [00, 53]	02
%w	weekday: [0=Sunday, 6]	2
%W	Monday week number: [00, 53]	02
%x	date	01/14/92
%X	time	19:40:30
%y	year without century: [00, 99]	92
%Y	year with century	1992
%Z	time zone name	MST

Figure 6.5 Conversion specifiers for `strftime`.

The third column of this figure is from the output of `strftime` under SVR4 corresponding to the time and date

```
Tue Jan 14 19:40:30 MST 1992
```

The only two specifiers that are not self-evident are `%U` and `%W`. The first is the week number of the year where the week containing the first Sunday is week 1. `%W` is the week number of the year where the week containing the first Monday is week 1.

> Both SVR4 and 4.3+BSD support additional, nonstandard extensions to the *format* string for `strftime`.

We mentioned that the four functions in Figure 6.4 with dashed lines were affected by the `TZ` environment variable: `localtime`, `mktime`, `ctime`, and `strftime`. If defined, the value of this environment variable is used by these functions instead of the default time zone. If the variable is defined to be a null string (e.g., `TZ=`) then UTC is normally used. The value of this environment variable is often something like `TZ=EST5EDT`, but POSIX.1 allows a much more detailed specification. Refer to Section 8.1.1 of the POSIX.1 standard [IEEE 1990], the SVR4 `environ`(5) manual page [AT&T 1990e], or the 4.3+BSD `ctime`(3) manual page for all the details on the `TZ` variable.

All the time and date functions described in this section are defined by the ANSI C standard. POSIX.1, however, added the `TZ` environment variable.

Five of the seven functions in Figure 6.4 date back to Version 7 (or earlier): `time`, `localtime`, `gmtime`, `asctime`, and `ctime`. Many of the recent additions to the Unix time keeping have dealt with non-U.S. time zones and the changing rules for daylight saving time.

6.10 Summary

The password file and group file are used on all Unix systems. We've looked at the various functions that read these files. We've also talked about shadow passwords, which can help system security. Supplementary group IDs are becoming common and provide a way to participate in multiple groups at the same time. We also looked at how similar functions are provided by most systems to access other system-related data files. We finished the chapter with a look at the time and date functions provided by ANSI C and POSIX.1.

Exercises

6.1 If the system uses a shadow file and we need to obtain the encrypted password, how do we do it?

6.2 If you have superuser access and your system uses shadow passwords, implement the previous exercise.

6.3 Write a program that calls `uname` and prints all the fields in the `utsname` structure. Compare the output to the output from the `uname(1)` command.

6.4 Write a program to obtain the current time and print it using `strftime` so that it looks like the default output from `date(1)`. Set the `TZ` environment variable to different values and see what happens.

7

The Environment of a Unix Process

7.1 Introduction

Before looking at the process control primitives in the next chapter, we need to examine the environment of a single process. We'll see how the `main` function is called when the program is executed, how command-line arguments are passed to the new program, what the typical memory layout looks like, how to allocate additional memory, how the process can use environment variables, and different ways for the process to terminate. Additionally we'll look at the `longjmp` and `setjmp` functions and their interaction with the stack. We finish the chapter by examining the resource limits of a process.

7.2 `main` Function

A C program starts execution with a function called `main`. The prototype for the `main` function is

```
int main(int argc, char *argv[]);
```

argc is the number of command-line arguments and *argv* is an array of pointers to the arguments. We describe these in Section 7.4.

When a C program is started by the kernel (by one of the `exec` functions, which we describe in Section 8.9), a special start-up routine is called before the `main` function is called. The executable program file specifies this start-up routine as the starting address for the program—this is set up by the link editor when it is invoked by the C compiler, usually `cc`. This start-up routine takes values from the kernel (the command-line arguments and the environment) and sets things up so that the `main` function is called as shown earlier.

7.3 Process Termination

There are five ways for a process to terminate.

1. Normal termination:
   (a) return from `main`
   (b) calling `exit`
   (c) calling `_exit`
2. Abnormal termination:
   (a) calling `abort` (Chapter 10)
   (b) terminated by a signal (Chapter 10)

The start-up routine that we mentioned in the previous section is also written so that if the `main` function returns, the `exit` function is called. If the start-up routine were coded in C (it is often coded in assembler) the call to `main` could look like

```
exit( main(argc, argv) );
```

`exit` and `_exit` Functions

Two functions terminate a program normally: `_exit`, which returns to the kernel immediately, and `exit`, which performs certain cleanup processing and then returns to the kernel.

```
#include <stdlib.h>

void exit(int status);

#include <unistd.h>

void _exit(int status);
```

We'll discuss the effect of these two functions on other processes, such as the children and the parent of the terminating process, in Section 8.5.

> The reason for the different headers is that `exit` is specified by ANSI C, while `_exit` is specified by POSIX.1.

Historically the `exit` function has always performed a clean shutdown of the standard I/O library: the `fclose` function is called for all open streams. Recall from Section 5.5 that this causes all buffered output data to be flushed (written to the file).

Both the `exit` and `_exit` functions expect a single integer argument, which we call the *exit status*. Most Unix shells provide a way to examine the exit status of a process. If (a) either of these functions is called without an exit status, (b) `main` does a `return`

without a return value, or (c) `main` "falls off the end" (an implicit return), the exit status of the process is undefined. This means that the classic example

```
#include    <stdio.h>

main()
{
    printf("hello, world\n");
}
```

is incomplete, since the `main` function falls off the end, returning to the C start-up routine, but without returning a value (the exit status). Adding either

```
return(0);
```

or

```
exit(0);
```

provides an exit status of 0 to the process that executed this program (often a shell). Also, the declaration of `main` should really be

```
int main(void)
```

In the next chapter we'll see how any process can cause a program to be executed, wait for the process to complete, then fetch its exit status.

> The declaration of `main` as returning an integer and the use of `exit` (instead of `return`) produces needless warnings from some compilers and the Unix `lint`(1) program. The problem is that these compilers don't know that an `exit` from `main` is the same as a `return`. The warning message is something like "control reaches end of nonvoid function." One way around these warnings (which become annoying after a while) is to use `return` instead of `exit` from `main`. But doing this prevents us from using the Unix `grep` utility to locate all calls to `exit` from a program. Another solution is to declare `main` as returning `void`, instead of `int`, and continue calling `exit`. This gets rid of the compiler warnings but doesn't look right (especially in a programming text). In this text we show `main` as returning an integer, since that is the definition specified by both ANSI C and POSIX.1. We'll just put up with the extraneous compiler warnings.

`atexit` Function

With ANSI C a process can register up to 32 functions that are automatically called by `exit`. These are called *exit handlers* and are registered by calling the `atexit` function.

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

Returns: 0 if OK, nonzero on error

This declaration says that we pass the address of a function as the argument to `atexit`. When this function is called it is not passed any arguments and it is not expected to return a value. The `exit` function calls these functions in reverse order of their registration. Each function is called as many times as it was registered.

These exit handlers are new with ANSI C. They are provided by both SVR4 and 4.3+BSD. Earlier releases of System V and 4.3BSD did not provide these exit handlers.

With ANSI C and POSIX.1, `exit` first calls the exit handlers and then `fcloses` all open streams. Figure 7.1 summarizes how a C program is started and the various ways it can terminate.

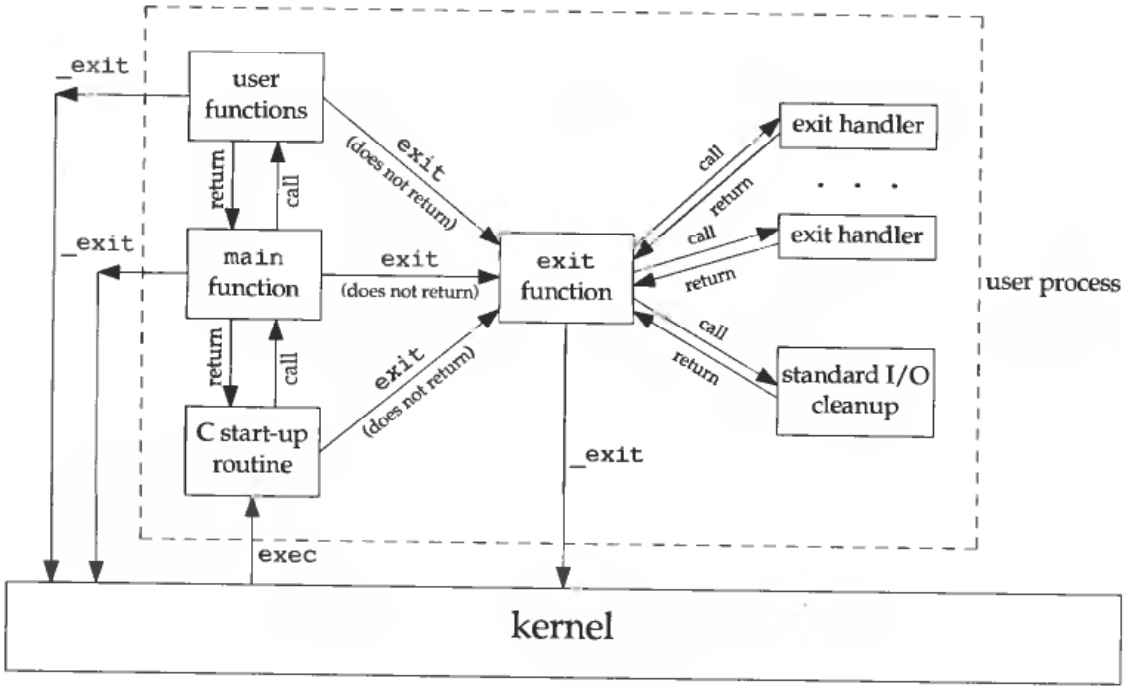


Figure 7.1 How a C program is started and how it terminates.

Note that the only way a program is executed by the kernel is when one of the `exec` functions is called. The only way a process voluntarily terminates is when `_exit` is called, either explicitly or implicitly (by calling `exit`). A process can also be involuntarily terminated by a signal (not shown in Figure 7.1).

Example

Program 7.1 demonstrates the use of the `atexit` function. Executing Program 7.1 yields

```
$ a.out
main is done
first exit handler
first exit handler
second exit handler
```

Note that we don't call `exit`, instead we return from `main`. □

```
#include    "ourhdr.h"

static void my_exit1(void), my_exit2(void);

int
main(void)
{
    if (atexit(my_exit2) != 0)
        err_sys("can't register my_exit2");

    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");
    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");

    printf("main is done\n");
    return(0);
}

static void
my_exit1(void)
{
    printf("first exit handler\n");
}

static void
my_exit2(void)
{
    printf("second exit handler\n");
}
```

Program 7.1 Example of exit handlers.

7.4 Command-Line Arguments

When a program is executed, the process that does the exec can pass command-line arguments to the new program. This is part of the normal operation of the Unix shells. We have already seen this in many of the examples from earlier chapters.

Example

Program 7.2 echoes all its command-line arguments to standard output. (The normal Unix echo(1) program doesn't echo the zeroth argument.) If we compile this program and name the executable echoarg, we have

```
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

```
#include    "ourhdr.h"

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

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}
```

Program 7.2 Echo all command-line arguments to standard output.

We are guaranteed by both ANSI C and POSIX.1 that `argv[argc]` is a null pointer. This lets us alternatively code the argument processing loop as

```
for (i = 0; argv[i] != NULL; i++)
```

□

7.5 Environment List

Each program is also passed an environment list. Like the argument list, the environment list is an array of character pointers, with each pointer containing the address of a null-terminated C string. The address of the array of pointers is contained in the global variable `environ`.

```
extern char **environ;
```

For example, if the environment consisted of five strings it could look like

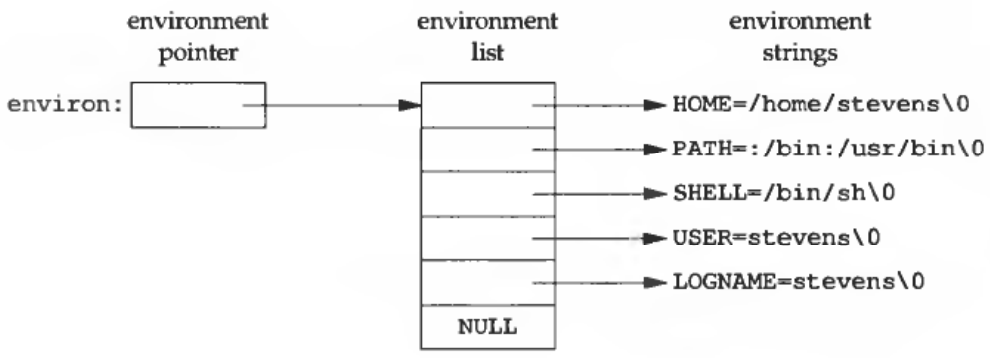


Figure 7.2 Environment consisting of five C character strings.

Here we explicitly show the null bytes at the end of each string. We'll call `environ` the environment pointer, the array of pointers the environment list, and the strings they point to the environment strings.

By convention the environment consists of

name=value

strings, as shown in Figure 7.2. Most predefined names are entirely uppercase, but this is only a convention.

Historically, most Unix systems have provided a third argument to the `main` function that is the address of the environment list:

```
int main(int argc, char *argv[], char *envp[]);
```

Since ANSI C specifies that the `main` function be written with two arguments, and since this third argument provides no benefit over the global variable `environ`, POSIX.1 specifies that `environ` should be used instead of the (possible) third argument. Access to specific environment variables is normally through the `getenv` and `putenv` functions (described in Section 7.9), instead of through the `environ` variable. But to go through the entire environment, the `environ` pointer must be used.

7.6 Memory Layout of a C Program

Historically a C program has been composed of the following pieces:

- Text segment. This is the machine instructions that are executed by the CPU. Usually the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs (text editors, the C compiler, the shells, etc.). Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.
- Initialized data segment. This is usually just called the data segment and it contains variables that are specifically initialized in the program. For example, the C declaration

  ```
  int   maxcount = 99;
  ```

  appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.
- Uninitialized data segment. This segment is often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to 0 before the program starts executing. The C declaration

  ```
  long   sum[1000];
  ```

  appearing outside any function causes this variable to be stored in the uninitialized data segment.
- Stack. This is where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to, and certain information about the caller's environment (such as some of the machine registers) is saved on the stack. The newly

called function then allocates room on the stack for its automatic and temporary variables. By utilizing a stack in this fashion, C functions can be recursive.

- Heap. Dynamic memory allocation usually takes place on the heap. Historically the heap has been located between the top of the uninitialized data and the bottom of the stack.

Figure 7.3 shows the typical arrangement of these segments. This is a logical picture of how a program looks—there is no requirement that a given implementation arrange its memory in this fashion. Nevertheless, this gives us a typical arrangement to describe.

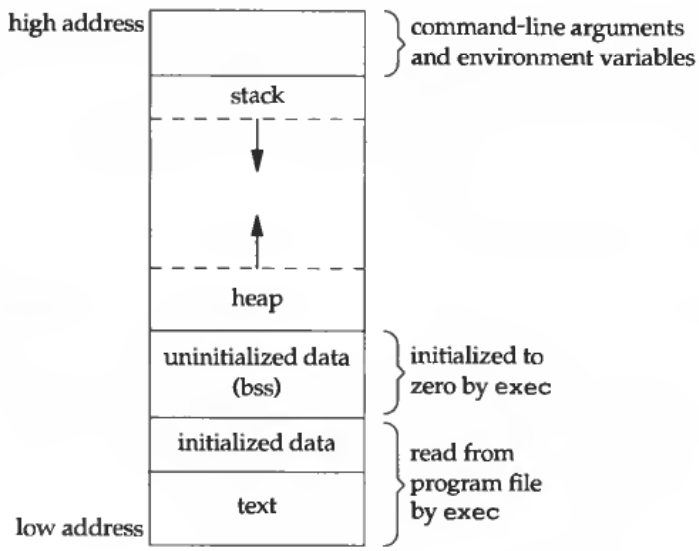


Figure 7.3 Typical memory arrangement.

With 4.3+BSD on a VAX, the text segment starts at location 0 and the top of the stack starts just below `0x7fffffff`. On the VAX the unused virtual address space between the top of the heap and the bottom of the stack is large.

Note from Figure 7.3 that the contents of the uninitialized data segment are not stored in the program file on disk. This is because the kernel sets it to 0 before the program starts running. The only portions of the program that need to be saved in the program file are the text segment and the initialized data.

The `size`(1) command reports the sizes in bytes of the text, data, and bss segments. For example

```
$ size /bin/cc /bin/sh
text    data    bss     dec     hex
81920   16384   664     98968   18298   /bin/cc
90112   16384   0       106496  1a000   /bin/sh
```

The fourth and fifth columns are the total of the sizes in decimal and hexadecimal.

7.7 Shared Libraries

Many Unix systems today support shared libraries. Arnold [1986] describes an early implementation under System V and Gingell et al. [1987] describe a different implementation under SunOS. Shared libraries remove the common library routines from the executable file, instead maintaining a single copy of the library routine somewhere in memory that all processes reference. This reduces the size of each executable file but may add some run-time overhead, either when the program is first `exec`ed, or the first time each shared library function is called. Another advantage of shared libraries is that library functions can be replaced with new versions without having to re-link edit every program that uses the library. (This assumes that the number and type of arguments haven't changed.)

Different systems provide different ways for a program to say that it wants to use or not use the shared libraries. Options for the `cc`(1) and `ld`(1) commands are typical. As an example of the size differences, the following executable file (the classic `hello.c` program) was first created without shared libraries.

```
$ ls -l a.out
-rwxrwxr-x  1 stevens    104859 Aug  2 14:25 a.out
$ size a.out
text    data    bss     dec     hex
49152   49152   0       98304   18000
```

If we compile this program to use shared libraries, the text and data sizes of the executable file are greatly decreased.

```
$ ls -l a.out
-rwxrwxr-x  1 stevens     24576 Aug  2 14:26 a.out
$ size a.out
text    data    bss     dec     hex
8192    8192    0       16384   4000
```

7.8 Memory Allocation

There are three functions specified by ANSI C for memory allocation.

1. `malloc`. Allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
2. `calloc`. Allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.
3. `realloc`. Changes the size of a previously allocated area (increases or decreases). When the size increases, it may involve moving the previously allocated area somewhere else, to provide the additional room at the end. Also, when the size increases, the initial value of the space between the old contents and the end of the new area is indeterminate.

```
#include <stdlib.h>

void *malloc(size_t size);

void *calloc(size_t nobj, size_t size);

void *realloc(void *ptr, size_t newsize);

                    All three return: nonnull pointer if OK, NULL on error

void free(void *ptr);
```

The pointer returned by the three allocation functions is guaranteed to be suitably aligned so that it can be used for any data object. For example, if the most restrictive alignment requirement on a particular system requires that `doubles` must start at memory locations that are multiples of 8, then all pointers returned by these three functions would be so aligned.

Recall our discussion of the generic `void *` pointer and function prototypes in Section 1.6. Since the three `alloc` functions return generic pointers, if we `#include <stdlib.h>` (to obtain the function prototypes), we do not explicitly have to cast the pointer returned by these functions when we assign it to a pointer of a different type.

The function `free` causes the space pointed to by *ptr* to be deallocated. This freed space is usually put into a pool of available memory and can be allocated in a later call to one of the three `alloc` functions.

`realloc` lets us increase or decrease the size of a previously allocated area. (The most common usage is to increase an area.) For example, if we allocate room for 512 elements in an array that we fill in at run time, and find we need room for more than 512 elements, we can call `realloc`. If there is room beyond the end of the existing region for the requested space, then `realloc` doesn't have to move anything, it just allocates the additional area at the end and returns the same pointer that we passed it. But if there isn't room at the end of the existing region, `realloc` allocates another area that is large enough, copies the existing 512-element array to the new area, frees the old area, and returns the pointer to the new area. Since the area may move, we shouldn't have any pointers into this area. Exercise 4.18 shows the use of `realloc` with `getcwd` to handle any length pathname. Program 15.27 shows an example that uses `realloc` to avoid arrays with fixed, compile-time sizes.

Notice that the final argument to `realloc` is the *newsize* of the region, not the difference between the old and new sizes. As a special case, if *ptr* is a null pointer, `realloc` behaves like `malloc` and allocates a region of the specified *newsize*.

> This feature is new with ANSI C. Older versions of `realloc` can fail miserably if passed a null pointer.
>
> Older versions of these routines allowed us to `realloc` a block that we had `freed` since the last call to `malloc`, `realloc`, or `calloc`. This trick dates back to Version 7 and exploited the search strategy of `malloc` to perform storage compaction. 4.3+BSD still supports this feature, but SVR4 doesn't. This feature is deprecated and should not be used.

The allocation routines are usually implemented with the `sbrk`(2) system call. This system call expands (or contracts) the heap of the process. (Refer to Figure 7.3.) A sample implementation of `malloc` and `free` is given in Section 8.7 of Kernighan and Ritchie [1988].

Although `sbrk` can expand or contract the memory of a process, most versions of `malloc` and `free` never decrease their memory size. The space that we free is available for a later allocation, but the freed space is not returned to the kernel—it is kept in the `malloc` pool.

It is important to realize that most implementations allocate a little more space that is requested and use the additional space for record keeping—the size of the allocated block, a pointer to the next allocated block, and the like. This means that writing past the end of an allocated area could overwrite this record keeping information in a later block. These types of errors are often catastrophic, but hard to find, because the error may not show up until much later. Also, it is possible to overwrite this record keeping in the current block by moving the pointer to the block backward.

Other possible errors that can be fatal are freeing a block that was already freed and calling `free` with a pointer that was not obtained from one of the three `alloc` functions. If a process calls `malloc` and thinks it's calling `free`, but its memory usage continually increases, this is called leakage. It is usually caused by not calling `free` to return unused space.

Since memory allocation errors are hard to track down, some systems provide versions of these functions that do additional error checking every time one of the three `alloc` functions or `free` is called. These versions of the functions are often specified by including a special library for the link editor. There are also publicly available sources (such as the one provided with 4.3+BSD) that you can compile with special flags to enable additional run-time checking.

Since the operation of the memory allocator is often crucial to the run-time performance of certain applications, some systems provide additional capabilities. For example, SVR4 provides a function named `mallopt` that allows a process to set certain variables that control the operation of the storage allocator. A function called `mallinfo` is also available to provide statistics on the memory allocator. Check the `malloc`(3) manual page for your system to see if any of these features are available.

`alloca` Function

One additional function is also worth mentioning. The function `alloca` has the same calling sequence as `malloc`, however instead of allocating memory from the heap, the memory is allocated from the stack frame of the current function. The advantage to this is that we don't have to free the space—it goes away automatically when the function returns. `alloca` increases the size of the stack frame. The disadvantage is that some systems can't support `alloca`, if it's impossible to increase the size of the stack frame after the function has been called. Nevertheless, many software packages use it, and implementations exist for a wide variety of systems.

7.9 Environment Variables

As we mentioned earlier, the environment strings are usually of the form

name=value

The Unix kernel never looks at these strings—their interpretation is up to the various applications. The shells, for example, use numerous environment variables. Some are set automatically at login (HOME, USER, etc.) and others are for us to set. We normally set environment variables in a shell start-up file to control the shell's actions. If we set the environment variable MAILPATH, for example, it tells the Bourne shell and Korn-Shell where to look for mail.

ANSI C defines a function that we can use to fetch values from the environment, but this standard says that the contents of the environment are implementation defined.

```
#include <stdlib.h>

char *getenv(const char *name);
```

Returns: pointer to *value* associated with *name*, NULL if not found

Note that this function returns a pointer to the *value* of a *name=value* string. We should always use getenv to fetch a specific value from the environment, instead of accessing environ directly.

Some environment variables are defined by POSIX.1 and XPG3. Figure 7.4 lists the ones defined by these standards and which are supported by SVR4 and 4.3+BSD. There are many additional implementation-dependent environment variables used in SVR4 and 4.3+BSD. Note that ANSI C doesn't define any environment variables.

FIPS 151-1 requires that a login shell must define the environment variables HOME and LOGNAME.

Variable	Standards		Implementations		Description
	POSIX.1	XPG3	SVR4	4.3+BSD	
HOME	•	•	•	•	home directory
LANG	•	•	•		name of locale
LC_ALL	•	•	•		name of locale
LC_COLLATE	•	•	•		name of locale for collation
LC_CTYPE	•	•	•		name of locale for character classification
LC_MONETARY	•	•	•		name of locale for monetary editing
LC_NUMERIC	•	•	•		name of locale for numeric editing
LC_TIME	•	•	•		name of locale for date/time formatting
LOGNAME	•	•	•	•	login name
NLSPATH		•	•		sequence of templates for message catalogs
PATH	•	•	•	•	list of path prefixes to search for executable file
TERM	•	•	•	•	terminal type
TZ	•	•	•	•	time zone information

Figure 7.4 Environment variables.

In addition to fetching the value of an environment variable, sometimes we want to set an environment variable. We may want to change the value of an existing variable, or add a new variable to the environment. (In the next chapter we'll see that we can affect the environment of only the current process and any child processes that we invoke. We cannot affect the environment of the parent process, which is often a shell. Nevertheless, it is still useful to be able to modify the environment list.) Unfortunately, not all systems support this capability. Figure 7.5 shows the various functions that are supported by the different standards and implementations.

Function	Standards			Implementations	
	ANSI C	POSIX.1	XPG3	SVR4	4.3+BSD
getenv	•	•	•	•	•
putenv		(maybe)	•	•	•
setenv					•
unsetenv					•
clearenv		(maybe)			

Figure 7.5 Support for various environment list functions.

The Rationale in the POSIX.1 standard states that `putenv` and `clearenv` are being considered for an amendment to POSIX.1.

The prototypes for the middle three functions listed in Figure 7.5 are

```
#include <stdlib.h>

int putenv(const char *str);

int setenv(const char *name, const char *value, int rewrite);
                                        Both return: 0 if OK, nonzero on error

void unsetenv(const char *name);
```

The operation of these three functions is

- `putenv` takes a string of the form *name=value* and places it in the environment list. If the *name* already exists, its old definition is first removed.
- `setenv` sets *name* to *value*. If *name* already exists in the environment then (a) if *rewrite* is nonzero, the existing definition for *name* is first removed; (b) if *rewrite* is 0, an existing definition for *name* is not removed (and *name* is not set to the new *value*, and no error occurs).
- `unsetenv` removes any definition of *name*. It is not an error if such a definition does not exist.

It is interesting to examine how these functions must operate when modifying the environment list. Recall Figure 7.3 where the environment list (the array of pointers to

the actual *name=value* strings) and the environment strings are typically stored at the top of a process's memory space (above the stack). Deleting a string is simple—we just find the pointer in the environment list and move all subsequent pointers down one—but adding a string, or modifying an existing string, is harder. The space at the top of the stack cannot be expanded because it is often at the top of the address space of the process. Since it's at the top it can't expand upward and it can't be expanded downward because all the stack frames below it can't be moved.

1. If we're modifying an existing *name*:
   (a) If the size of the new *value* is less than or equal to the size of the existing *value*, we can just copy the new string over the old string.
   (b) If the new *value* is larger than the old one, however, we must `malloc` to obtain room for the new string, copy the new string to this area, then replace the old pointer in the environment list for *name* with the pointer to this `malloc`ed area.
2. If we're adding a new *name* it's more complicated. First we have to call `malloc` to allocate room for the *name=value* string and copy the string to this area.
   (a) Then, if it's the first time we've added a new *name*, we have to call `malloc` to obtain room for a new list of pointers. We copy the old environment list to this new area and store a pointer to the *name=value* string at the end of this list of pointers. We also store a null pointer at the end of this list, of course. Finally we set `environ` to point to this new list of pointers. Note from Figure 7.3 that if the original environment list was contained above the top of the stack (as is common), then we have moved this list of pointers to the heap. But most of the pointers in this list still point to *name=value* strings above the top of the stack.
   (b) If this isn't the first time we've added new strings to the environment list, then we know we've already `malloc`ed room for the list on the heap, so we just call `realloc` to allocate room for one more pointer. The pointer to the new *name=value* string is stored at the end of the list (on top of the previous null pointer), followed by a null pointer.

7.10 `setjmp` and `longjmp` Functions

In C we can't `goto` a label that's in another function. Instead we must use the `setjmp` and `longjmp` functions to perform this type of branching. As we'll see, these two functions are useful for handling error conditions that occur in a deeply nested function call.

Consider the skeleton in Program 7.3. It consists of a main loop that reads lines from standard input and calls the function `do_line` to process each line. This function then calls `get_token` to fetch the next token from the input line. The first token of a line is assumed to be a command of some form and a `switch` statement selects each command. For the single command shown, the function `cmd_add` is called.

```
#include    "ourhdr.h"

#define TOK_ADD    5

void    do_line(char *);
void    cmd_add(void);
int     get_token(void);

int
main(void)
{
    char    line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

char    *tok_ptr;       /* global pointer for get_token() */

void
do_line(char *ptr)      /* process one line of input */
{
    int     cmd;

    tok_ptr = ptr;
    while ( (cmd = get_token()) > 0) {
        switch (cmd) {  /* one case for each command */
        case TOK_ADD:
                cmd_add();
                break;
        }
    }
}

void
cmd_add(void)
{
    int     token;

    token = get_token();
    /* rest of processing for this command */
}

int
get_token(void)
{
    /* fetch next token from line pointed to by tok_ptr */
}
```

Program 7.3 Typical program skeleton for command processing.

Program 7.3 is typical for programs that read commands, determine the command type, and then call functions to process each command. Figure 7.6 shows what the stack could look like after `cmd_add` has been called.

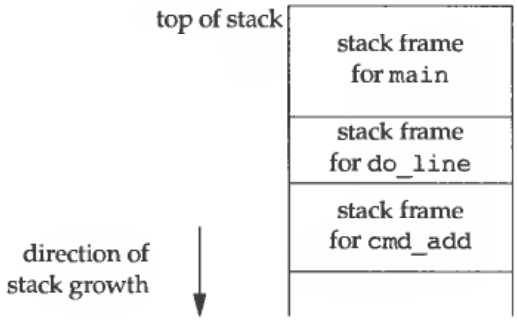


Figure 7.6 Stack frames after `cmd_add` has been called.

Storage for the automatic variables is within the stack frame for each function. The array `line` is in the stack frame for `main`, the integer `cmd` is in the stack frame for `do_line`, and the integer `token` is in the stack frame for `cmd_add`.

As we've said, this type of arrangement of the stack is typical, but not required. Stacks do not have to grow toward lower memory addresses. On systems that don't have built-in hardware support for stacks, a C implementation might use a linked list for its stack frames.

The coding problem that's often encountered with programs like Program 7.3 is how to handle nonfatal errors. For example, if the `cmd_add` function encounters an error, say an invalid number, it might want to print an error, ignore the rest of the input line, and return to the `main` function to read the next input line. But when we're deeply nested numerous levels down from the `main` function, this is hard to do in C. (In this example, in the `cmd_add` function, we're only two levels down from `main`, but it's not uncommon to be 5 or more levels down from where we want to return to.) It becomes messy if we have to code each function with a special return value that tells it to return one level.

The solution to this problem is to use a nonlocal goto—the `setjmp` and `longjmp` functions. The adjective nonlocal is because we're not doing a normal C `goto` statement within a function; instead we're branching back through the call frames to a function that is in the call path of the current function.

```
#include <setjmp.h>

int setjmp(jmp_buf env);

                Returns: 0 if called directly, nonzero if returning from a call to longjmp

void longjmp(jmp_buf env, int val);
```

We call setjmp from the location that we want to return to, which in this example is in the main function. setjmp returns 0 in this case, because we called it directly. In the call to setjmp the argument *env* is of the special type jmp_buf. This data type is some form of array that is capable of holding all the information required to restore the status of the stack to the state when we call longjmp. Normally the *env* variable is a global variable, since we'll need to reference it from another function.

When we encounter an error, say in the cmd_add function, we call longjmp with two arguments. The first is the same *env* that we used in a call to setjmp, and the second, *val*, is a nonzero value that becomes the return value from setjmp. The reason for the second argument is to allow us to have more than one longjmp for each setjmp. For example, we could longjmp from cmd_add with a *val* of 1 and also call longjmp from get_token with a *val* of 2. In the main function, the return value from setjmp is either 1 or 2, and we can test this value (if we want) and determine if the longjmp was from cmd_add or get_token.

Let's return to the example. Program 7.4 shows both the main and cmd_add functions. (The other two functions, do_line and get_token haven't changed.)

```
#include    <setjmp.h>
#include    "ourhdr.h"

#define TOK_ADD    5

jmp_buf jmpbuffer;

int
main(void)
{
    char    line[MAXLINE];

    if (setjmp(jmpbuffer) != 0)
        printf("error");
    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

 . . .

void
cmd_add(void)
{
    int     token;

    token = get_token();
    if (token < 0)      /* an error has occurred */
        longjmp(jmpbuffer, 1);
    /* rest of processing for this command */
}
```

Program 7.4 Example of setjmp and longjmp.

When `main` is executed, we call `setjmp` and it records whatever information it needs to in the variable `jmpbuffer` and returns 0. We then call `do_line`, which calls `cmd_add`, and assume an error of some form is detected. Before the call to `longjmp` in `cmd_add`, the stack looks like that in Figure 7.6. But `longjmp` causes the stack to be "unwound" back to the `main` function, throwing away the stack frames for `cmd_add` and `do_line`. Calling `longjmp` causes the `setjmp` in `main` to return, but this time it returns with a value of 1 (the second argument for `longjmp`).

Automatic, Register, and Volatile Variables

The next question is "what are the states of the automatic variables and register variables in the `main` function?" When `main` is returned to by the `longjmp`, do these variables have values corresponding to when the `setjmp` was previously called (i.e., are their values rolled back), or are their values left alone so that their values are whatever they were when `do_line` was called (which caused `cmd_add` to be called, which caused `longjmp` to be called)? Unfortunately, the answer is "it depends." Most implementations do not try to roll back these automatic variables and register variables, but all that the standards say is that their values are indeterminate. If you have an automatic variable that you don't want rolled back, define it with the `volatile` attribute. Variables that are declared global or static are left alone when `longjmp` is executed.

Example

Program 7.5 demonstrates the different behavior that can be seen with automatic, register, and volatile variables, after calling `longjmp`. If we compile and test Program 7.5, with and without compiler optimizations, the results are different:

```
$ cc testjmp.c                    compile without any optimization
$ a.out
in f1(): count = 97, val = 98, sum = 99
after longjmp: count = 97, val = 98, sum = 99
$ cc -O testjmp.c                 compile with full optimization
$ a.out
in f1(): count = 97, val = 98, sum = 99
after longjmp: count = 2, val = 3, sum = 99
```

Note that the `volatile` variable (`sum`) isn't affected by the optimizations—its value after the `longjmp` is the last value that it assumed. The `setjmp`(3) manual page on one system states that variables stored in memory will have values as of the time of the `longjmp`, while variables in the CPU and floating point registers are restored to their values when `setjmp` was called. This is indeed what we see when we run Program 7.5. Without optimization all three variables are stored in memory (i.e., the `register` hint is ignored for `val`). When we enable optimization, both `count` and `val` go into registers (even though the former wasn't declared `register`) and the `volatile` variable stays in memory. The thing to realize with this example is that you must use the `volatile` attribute if you're writing portable code that uses nonlocal jumps. Anything else can change from one system to the next. □

```
#include    <setjmp.h>
#include    "ourhdr.h"

static void f1(int, int, int);
static void f2(void);

static jmp_buf  jmpbuffer;

int
main(void)
{
    int             count;
    register int    val;
    volatile int    sum;

    count = 2; val = 3; sum = 4;
    if (setjmp(jmpbuffer) != 0) {
        printf("after longjmp: count = %d, val = %d, sum = %d\n",
                count, val, sum);
        exit(0);
    }
    count = 97; val = 98; sum = 99;
                /* changed after setjmp, before longjmp */
    f1(count, val, sum);        /* never returns */
}

static void
f1(int i, int j, int k)
{
    printf("in f1(): count = %d, val = %d, sum = %d\n", i, j, k);
    f2();
}

static void
f2(void)
{
    longjmp(jmpbuffer, 1);
}
```

Program 7.5 Effect of `longjmp` on automatic, register, and volatile variables.

We'll return to these two functions, `setjmp` and `longjmp` in Chapter 10 when we discuss signal handlers and their signal versions `sigsetjmp` and `siglongjmp`.

Potential Problem with Automatic Variables

Having looked at the way stack frames are usually handled, it is worth looking at a potential error in dealing with automatic variables. The basic rule is that an automatic

variable can never be referenced after the function that declared the automatic variable returns. There are numerous warnings about this throughout the Unix manuals.

Program 7.6 is a function called `open_data` that opens a standard I/O stream and sets the buffering for the stream.

```
#include    <stdio.h>

#define DATAFILE    "datafile"

FILE *
open_data(void)
{
    FILE    *fp;
    char    databuf[BUFSIZ];  /* setvbuf makes this the stdio buffer */

    if ( (fp = fopen(DATAFILE, "r")) == NULL)
        return(NULL);

    if (setvbuf(fp, databuf, BUFSIZ, _IOLBF) != 0)
        return(NULL);

    return(fp);     /* error */
}
```

Program 7.6 Incorrect usage of an automatic variable.

The problem is that when `open_data` returns, the space it used on the stack will be used by the stack frame for the next function that is called. But the standard I/O library will still be using that portion of memory for its buffer for the stream. Chaos is sure to result. To correct this problem the array `databuf` needs to be allocated from global memory, either statically (`static` or `extern`) or dynamically (one of the `alloc` functions).

7.11 `getrlimit` and `setrlimit` Functions

Every process has a set of resource limits, some of which can be queried and changed by the `getrlimit` and `setrlimit` functions.

```
#include <sys/time.h>
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlptr);

int setrlimit(int resource, const struct rlimit *rlptr);
```

Both return: 0 if OK, nonzero on error

Each call to these two functions specifies a single *resource* and a pointer to the following structure.

```
struct rlimit {
  rlim_t  rlim_cur;  /* soft limit: current limit */
  rlim_t  rlim_max;  /* hard limit: maximum value for rlim_cur */
};
```

These two functions are not part of POSIX.1, but SVR4 and 4.3+BSD provide them.

SVR4 uses the primitive system data type `rlim_t` in the preceding structure. Other systems define these two members as integers or long integers.

The resource limits for a process are normally established by process 0 when the system is initialized and then inherited by each successive process. In SVR4 the defaults can be examined in the file `/etc/conf/cf.d/mtune`. In 4.3+BSD the defaults are scattered among various headers.

Three rules govern the changing of the resource limits:

1. A soft limit can be changed by any process to a value less than or equal to its hard limit.
2. Any process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.
3. Only a superuser process can raise a hard limit.

An infinite limit is specified by the constant `RLIM_INFINITY`.

The *resource* argument takes on one of the following values. Note that not all resources are supported by both SVR4 and 4.3+BSD.

`RLIMIT_CORE`	(SVR4 and 4.3+BSD) The maximum size in bytes of a core file. A limit of 0 prevents the creation of a core file.
`RLIMIT_CPU`	(SVR4 and 4.3+BSD) The maximum amount of CPU time in seconds. When the soft limit is exceeded, the `SIGXCPU` signal is sent to the process.
`RLIMIT_DATA`	(SVR4 and 4.3+BSD) The maximum size in bytes of the data segment. This is the sum of the initialized data, uninitialized data, and heap from Figure 7.3.
`RLIMIT_FSIZE`	(SVR4 and 4.3+BSD) The maximum size in bytes of a file that may be created. When the soft limit is exceeded, the `SIGXFSZ` signal is sent to the process.
`RLIMIT_MEMLOCK`	(4.3+BSD only) Locked-in-memory address space (not implemented yet).
`RLIMIT_NOFILE`	(SVR4 only) The maximum number of open files per process. Changing this limit affects the value returned by the `sysconf` function for its `_SC_OPEN_MAX` argument (Section 2.5.4). See Program 2.3 also.

RLIMIT_NPROC	(4.3+BSD only) The maximum number of child processes per real user ID. Changing this limit affects the value returned for _SC_CHILD_MAX by the sysconf function (Section 2.5.4).
RLIMIT_OFILE	(4.3+BSD) Same as the SVR4 RLIMIT_NOFILE.
RLIMIT_RSS	(4.3+BSD only) Maximum resident set size (RSS) in bytes. If physical memory is tight, the kernel takes memory from processes that exceed their RSS.
RLIMIT_STACK	(SVR4 and 4.3+BSD) The maximum size in bytes of the stack. See Figure 7.3.
RLIMIT_VMEM	(SVR4 only) The maximum size in bytes of the mapped address space. This affects the mmap function (Section 12.9).

The resource limits affect the calling process and are inherited by any of its children. This means that the setting of resource limits really needs to be built into the shells to affect all our future processes. Indeed, the Bourne shell and KornShell have the built-in ulimit command, and the C shell has the built-in limit command. (The umask and chdir functions also have to be handled as shell built-ins.)

> Older Bourne shells, such as the one distributed by Berkeley, don't support the ulimit command.
>
> Newer versions of the KornShell have undocumented -H and -S options for the ulimit command, to examine or modify the hard or soft limits, respectively.

Example

Program 7.7 prints out the current soft limit and the hard limit for all the resource limits supported on the system. To run this program under both SVR4 and 4.3+BSD we have conditionally compiled the resource names that differ.

```
#include    <sys/types.h>
#include    <sys/time.h>
#include    <sys/resource.h>
#include    "ourhdr.h"

#define doit(name)  pr_limits(#name, name)

static void pr_limits(char *, int);

int
main(void)
{
    doit(RLIMIT_CORE);
    doit(RLIMIT_CPU);
    doit(RLIMIT_DATA);
    doit(RLIMIT_FSIZE);
```

```
#ifdef  RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif
#ifdef  RLIMIT_NOFILE    /* SVR4 name */
    doit(RLIMIT_NOFILE);
#endif
#ifdef  RLIMIT_OFILE     /* 4.3+BSD name */
    doit(RLIMIT_OFILE);
#endif
#ifdef  RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif
#ifdef  RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif
    doit(RLIMIT_STACK);
#ifdef  RLIMIT_VMEM
    doit(RLIMIT_VMEM);
#endif
    exit(0);
}

static void
pr_limits(char *name, int resource)
{
    struct rlimit   limit;

    if (getrlimit(resource, &limit) < 0)
        err_sys("getrlimit error for %s", name);
    printf("%-14s  ", name);
    if (limit.rlim_cur == RLIM_INFINITY)
        printf("(infinite)  ");
    else
        printf("%10ld  ", limit.rlim_cur);
    if (limit.rlim_max == RLIM_INFINITY)
        printf("(infinite)\n");
    else
        printf("%10ld\n", limit.rlim_max);
}
```

Program 7.7 Print the current resource limits.

Note that we've used the new ANSI C string-creation operator (#) in the `doit` macro, to generate the string value for each resource name. When we say

```
doit(RLIMIT_CORE);
```

this is expanded by the C preprocessor into

```
pr_limits("RLIMIT_CORE", RLIMIT_CORE);
```

Running this program under SVR4 gives us the following:

```
$ a.out
RLIMIT_CORE          1048576     1048576
RLIMIT_CPU         (infinite)  (infinite)
RLIMIT_DATA         16777216    16777216
RLIMIT_FSIZE         2097152     2097152
RLIMIT_NOFILE             64        1024
RLIMIT_STACK        16777216    16777216
RLIMIT_VMEM         16777216    16777216
```

4.3+BSD gives us the following results:

```
$ a.out
RLIMIT_CORE        (infinite)  (infinite)
RLIMIT_CPU         (infinite)  (infinite)
RLIMIT_DATA          8388608    16777216
RLIMIT_FSIZE       (infinite)  (infinite)
RLIMIT_MEMLOCK     (infinite)  (infinite)
RLIMIT_OFILE              64   (infinite)
RLIMIT_NPROC              40   (infinite)
RLIMIT_RSS          27070464    27070464
RLIMIT_STACK          524288    16777216
```

□

Exercise 10.11 continues the discussion of resource limits, after we've covered signals.

7.12 Summary

Understanding the environment of a C program in a Unix environment is a requisite to understanding the process control features of Unix. In this chapter we've looked at how a process is started, how it can terminate, and how it's passed an argument list and an environment. Although both are uninterpreted by the kernel, it is the kernel that passes both from the caller of `exec` to the new process.

We've also examined the typical memory layout of a C program and how a process can dynamically allocate and free memory. It is worthwhile to look in detail at the functions available for manipulating the environment, since they involve memory allocation. The functions `setjmp` and `longjmp` were presented, providing a way to perform nonlocal branching within a process. We finished the chapter describing the resource limits that are provided by SVR4 and 4.3+BSD.

Exercises

7.1 On an 80386 system under both SVR4 and 4.3+BSD, if we execute the program that prints "hello, world", without calling `exit` or `return`, the termination status of the program (which we can examine with the shell) is 13. Why?

7.2 When is the output from the `printfs` in Program 7.1 actually output?

7.3 Is there any way for a function that is called by `main` to examine the command-line arguments, without (a) passing `argc` and `argv` as arguments from `main` to the function, or (b) having `main` copy `argc` and `argv` into global variables?

7.4 Some Unix implementations purposely arrange that, when a program is executed, location 0 in the data segment is not accessible. Why?

7.5 Use the `typdef` facility of C to define a new data type `Exitfunc` for an exit handler. Redo the prototype for `atexit` using this data type.

7.6 If we allocate an array of `long`s using `calloc` is the array initialized to 0? If we allocate an array of pointers using `calloc` is the array initialized to null pointers?

7.7 In the output from the `size` command at the end of Section 7.6, why aren't any sizes given for the heap and the stack?

7.8 In Section 7.7 the two file sizes (104859 and 24576) don't equal the sums of their respective text and data sizes. Why?

7.9 In Section 7.7 why is there such a difference in the size of the executable file when using shared libraries for such a trivial program?

7.10 At the end of Section 7.10 we showed how a function can't return a pointer to an automatic variable. Is the following code correct?

```
int
f1(int val)
{
    int     *ptr;

    if (val == 0) {
        int     val;

        val = 5;
        ptr = &val;
    }
    return(*ptr + 1);
}
```

8

Process Control

8.1 Introduction

We now turn to the process control provided by Unix. This includes the creation of new processes, executing programs, and process termination. We also look at the various IDs that are the property of the process—real, effective, and saved; user and group IDs—and how they're affected by the process control primitives. Interpreter files and the `system` function are also covered. We conclude the chapter by looking at the process accounting that is provided by most Unix systems. This lets us look at the process control functions from a different perspective.

8.2 Process Identifiers

Every process has a unique process ID, a nonnegative integer. Since the process ID is the only well-known identifier of a process that is always unique, it is often used as a piece of other identifiers, to guarantee uniqueness. The `tmpnam` function in Section 5.13 created unique pathnames by incorporating the process ID in the name.

There are some special processes. Process ID 0 is usually the scheduler process and is often known as the *swapper*. No program on disk corresponds to this process—it is part of the kernel and is known as a system process. Process ID 1 is usually the `init` process and is invoked by the kernel at the end of the bootstrap procedure. The program file for this process was `/etc/init` in older versions of Unix and is `/sbin/init` in newer versions. This process is responsible for bringing up a Unix system after the kernel has been bootstrapped. `init` usually reads the system-dependent initialization files (the `/etc/rc*` files) and brings the system to a certain state (such as multiuser). The `init` process never dies. It is a normal user process (not a system process within

the kernel like the swapper), although it does run with superuser privileges. Later in this chapter we'll see how `init` becomes the parent process of any orphaned child process.

On some virtual memory implementations of Unix, process ID 2 is the *pagedaemon*. This process is responsible for supporting the paging of the virtual memory system. Like the swapper, the pagedaemon is a kernel process.

In addition to the process ID, there are other identifiers for every process. The following functions return these identifiers.

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);                    Returns: process ID of calling process

pid_t getppid(void);                   Returns: parent process ID of calling process

uid_t getuid(void);                    Returns: real user ID of calling process

uid_t geteuid(void);                   Returns: effective user ID of calling process

gid_t getgid(void);                    Returns: real group ID of calling process

gid_t getegid(void);                   Returns: effective group ID of calling process
```

Note that none of these functions has an error return. We'll return to the parent process ID in the next section when we discuss the `fork` function. The real and effective user and group IDs were discussed in Section 4.4.

8.3 `fork` Function

The *only* way a new process is created by the Unix kernel is when an existing process calls the `fork` function. (This doesn't apply to the special processes that we mentioned in the previous section—the swapper, `init`, and the pagedaemon. These processes are created specially by the kernel as part of the bootstrapping.)

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);

                    Returns: 0 in child, process ID of child in parent, -1 on error
```

The new process created by `fork` is called the *child process*. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0 while the return value in the parent is the process ID of the new child. The reason the child's process ID is returned to the parent is because a process can have more than one child, so there is no function that allows a process to obtain the process IDs of its

children. The reason `fork` returns 0 to the child is because a process can have only a single parent, so the child can always call `getppid` to obtain the process ID of its parent. (Process ID 0 is always in use by the swapper, so it's not possible for 0 to be the process ID of a child.)

Both the child and parent continue executing with the instruction that follows the call to `fork`. The child is a copy of the parent. For example, the child gets a copy of the parent's data space, heap, and stack. Note that this is a copy for the child—the parent and child do not share these portions of memory. Often the parent and child share the text segment (Section 7.6), if it is read-only.

Many current implementations don't perform a complete copy of the parent's data, stack, and heap, since a `fork` is often followed by an `exec`. Instead, a technique called *copy-on-write* (COW) is used. These regions are shared by the parent and child and have their protection changed by the kernel to read-only. If either process tries to modify these regions, the kernel then makes a copy of that piece of memory only, typically a "page" in a virtual memory system. Section 9.2 of Bach [1986] and Section 5.7 of Leffler et al. [1989] provide more detail on this feature.

Example

Program 8.1 demonstrates the `fork` function. If we execute this program we get

```
$ a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89      child's variables were changed
pid = 429, glob = 6, var = 88      parent's copy were not changed
$ a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

In general, we never know if the child starts executing before the parent or vice versa. This depends on the scheduling algorithm used by the kernel. If it's required that the child and parent synchronize with each other, some form of interprocess communication is required. In Program 8.1 we just have the parent put itself to sleep for 2 seconds, to let the child execute. There is no guarantee that this is adequate, and we talk about this and other types of synchronization in Section 8.8 when we talk about race conditions. In Section 10.16 we show how to synchronize a parent and child after a `fork` using signals.

Note the interaction of `fork` with the I/O functions in Program 8.1. Recall from Chapter 3 that the `write` function is not buffered. Since `write` is called before the `fork`, its data is written once to standard output. The standard I/O library, however, is buffered. Recall from Section 5.12 that standard output is line buffered if it's connected to a terminal device, otherwise it's fully buffered. When we run the program interactively we get only a single copy of the `printf` line, because the standard output buffer

```
#include    <sys/types.h>
#include    "ourhdr.h"

int     glob = 6;       /* external variable in initialized data */
char    buf[] = "a write to stdout\n";

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");     /* we don't flush stdout */

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {         /* child */
        glob++;                  /* modify variables */
        var++;
    } else
        sleep(2);                /* parent */

    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

Program 8.1 Example of `fork` function.

is flushed by the newline. But when we redirect standard output to a file we get two copies of the `printf` line. What has happened in this second case is that the `printf` before the `fork` is called once, but the line remains in the buffer when `fork` is called. This buffer is then copied into the child, when the parent's data space is copied to the child. Both the parent and child now have a standard I/O buffer with this line in it. The second `printf`, right before the `exit`, just appends its data to the existing buffer. When each process terminates, its copy of the buffer is finally flushed. □

File Sharing

Another point to note from Program 8.1 is, when we redirect the standard output of the parent, the child's standard output is also redirected. Indeed, one characteristic of `fork` is that all descriptors that are open in the parent are duplicated in the child. We say "duplicated" because it's as if the dup function had been called for each descriptor. The parent and child share a file table entry for every open descriptor (recall Figure 3.4).

Consider a process that has three different files opened for standard input, standard output, and standard error. On return from `fork` we have the arrangement shown in Figure 8.1.

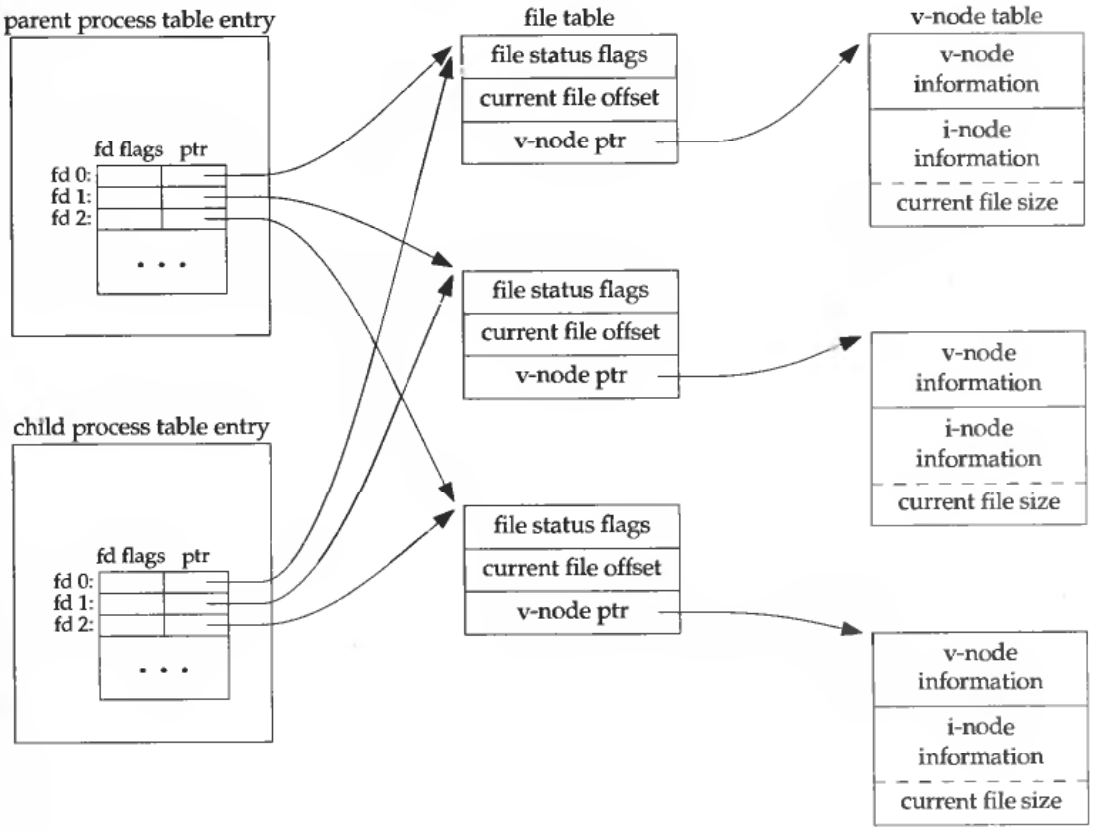


Figure 8.1 Sharing of open files between parent and child after `fork`.

It is important that the parent and child share the same file offset. Consider a process that `forks` a child, then `waits` for the child to complete. Assume that both processes write to standard output as part of their normal processing. If the parent has its standard output redirected (by a shell, perhaps) it is essential that the parent's file offset be updated by the child, if the child writes to standard output. In this case the child can write to standard output while the parent is `waiting` for it, and on completion of the child the parent can continue writing to standard output, knowing that its output will be appended to whatever the child wrote. If the parent and child did not share the same file offset, this type of interaction would be harder to accomplish and would require explicit actions by the parent.

If both parent and child write to the same descriptor, without any form of synchronization (such as having the parent `wait` for the child), their output will be intermixed (assuming it's a descriptor that was open before the `fork`). While this is possible (we saw it in Program 8.1), it's not the normal mode of operation.

There are two normal cases for handling the descriptors after a `fork`.

1. The parent waits for the child to complete. In this case, the parent does not need to do anything with its descriptors. When the child terminates, any of the

shared descriptors that the child read from or wrote to will have their file offsets updated accordingly.

2. The parent and child each go their own way. Here, after the `fork`, the parent closes the descriptors that it doesn't need and the child does the same thing. This way neither interferes with the other's open descriptors. This scenario is often the case with network servers.

Besides the open files, there are numerous other properties of the parent that are inherited by the child:

- real user ID, real group ID, effective user ID, effective group ID
- supplementary group IDs
- process group ID
- session ID
- controlling terminal
- set-user-ID flag and set-group-ID flag
- current working directory
- root directory
- file mode creation mask
- signal mask and dispositions
- the close-on-exec flag for any open file descriptors
- environment
- attached shared memory segments
- resource limits

The differences between the parent and child are

- the return value from `fork`
- the process IDs are different
- the two processes have different parent process IDs—the parent process ID of the child is the parent; the parent process ID of the parent doesn't change
- the child's values for `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_ustime` are set to 0
- file locks set by the parent are not inherited by the child
- pending alarms are cleared for the child
- the set of pending signals for the child is set to the empty set

Many of these features haven't been discussed yet—we'll cover them in later chapters.

The two main reasons for fork to fail are (a) if there are already too many processes in the system (which usually means something else is wrong), or (b) if the total number of processes for this real user ID exceeds the system's limit. Recall from Figure 2.7 that CHILD_MAX specifies the maximum number of simultaneous processes per real user ID.

There are two uses for fork.

1. When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time. This is common for network servers—the parent waits for a service request from a client. When the request arrives, the parent calls fork and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.
2. When a process wants to execute a different program. This is common for shells. In this case the child does an exec (which we describe in Section 8.9) right after it returns from the fork.

Some operating systems combine the operations from step 2 (a fork followed by an exec) into a single operation called a spawn. Unix separates the two as there are numerous uses for fork without doing an exec. Also, separating the two allows the child to change the per-process attributes between the fork and exec, such as I/O redirection, user ID, signal disposition, and so on. We'll see numerous examples of this in Chapter 14.

8.4 vfork Function

The function vfork has the same calling sequence and same return values as fork. But the semantics of the two functions differ.

> vfork originated with the early virtual-memory releases of 4BSD. In Section 5.7 of Leffler et al. [1989] they state, "Although it is extremely efficient, vfork has peculiar semantics and is generally considered to be an architectural blemish."
>
> Nevertheless, both SVR4 and 4.3+BSD support vfork.
>
> Some systems have a header <vfork.h> that should be included when calling vfork.

vfork is intended to create a new process when the purpose of the new process is to exec a new program (step 2 at the end of the previous section). The bare bones shell in Program 1.5 is also an example of this type of program. vfork creates the new process, just like fork, without fully copying the address space of the parent into the child, since the child won't reference that address space—the child just calls exec (or exit) right after the vfork. Instead, while the child is running, until it calls either exec or exit, the child runs in the address space of the parent. This optimization provides an efficiency gain on some paged virtual-memory implementations of Unix. (As we mentioned in the previous section, some implementations use copy-on-write to improve the efficiency of a fork followed by an exec.)

Another difference between the two functions is that `vfork` guarantees that the child runs first, until the child calls `exec` or `exit`. When the child calls either of these functions, the parent resumes. (This can lead to deadlock if the child depends on further actions of the parent before calling either of these two functions.)

Example

Let's look at Program 8.1, replacing the call to `fork` with `vfork`. We've removed the `write` to standard output. Also, we don't need to have the parent call `sleep`, since we're guaranteed that it is put to sleep by the kernel until the child calls either `exec` or `exit`.

```
#include    <sys/types.h>
#include    "ourhdr.h"

int     glob = 6;       /* external variable in initialized data */

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    printf("before vfork\n");   /* we don't flush stdio */

    if ( (pid = vfork()) < 0)
        err_sys("vfork error");
    else if (pid == 0) {        /* child */
        glob++;                 /* modify parent's variables */
        var++;
        _exit(0);               /* child terminates */
    }

    /* parent */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

Program 8.2 Example of `vfork` function.

Running this program gives us

```
$ a.out
before vfork
pid = 607, glob = 7, var = 89
```

Here the incrementing of the variables done by the child changes the values in the parent. Since the child runs in the address space of the parent, this doesn't surprise us. This behavior, however, differs from `fork`.

Notice in Program 8.2 that we call _exit instead of exit. As we described in Section 8.5, _exit does not perform any flushing of standard I/O buffers. If we call exit instead, the output is different.

```
$ a.out
before vfork
```

Here the output from the parent's printf has disappeared! What's happening here is that the child calls exit, which flushes and closes all the standard I/O streams. This includes standard output. Even though this is done by the child, it's done in the parent's address space, so all the standard I/O FILE objects that are modified are modified in the parent. When the parent calls printf later, standard output has been closed, and printf returns –1. □

Section 5.7 of Leffler et al. [1989] contains additional information on the implementation issues of fork and vfork. Exercises 8.1 and 8.2 continue the discussion of vfork.

8.5 exit Functions

There are three ways for a process to terminate normally, as we described in Section 7.3, and two forms of abnormal termination.

1. Normal termination:
   (a) Executing a return from the main function. As we saw in Section 7.3, this is equivalent to calling exit.
   (b) Calling the exit function. This function is defined by ANSI C and includes the calling of all exit handlers that have been registered by calling atexit and closing all standard I/O streams. Since ANSI C does not deal with file descriptors, multiple processes (parents and children), and job control, the definition of this function is incomplete for a Unix system.
   (c) Calling the _exit function. This function is called by exit and handles the Unix-specific details. _exit is specified by POSIX.1.

   In most Unix implementations exit(3) is a function in the standard C library while _exit(2) is a system call.

2. Abnormal termination:
   (a) Calling abort. This is a special case of the next item, since it generates the SIGABRT signal.
   (b) When the process receives certain signals. (We describe signals in more detail in Chapter 10). The signal can be generated by the process itself (e.g., calling the abort function), by some other process, or by the kernel. Examples of signals generated by the kernel could be because the process references a memory location not within its address space or dividing by 0.

Regardless how a process terminates, the same code in the kernel is eventually executed. This kernel code closes all the open descriptors for the process, releases the memory that it was using, and the like.

For any of the preceding cases we want the terminating process to be able to notify its parent how it terminated. For the `exit` and `_exit` functions this is done by passing an exit status as the argument to these two functions. In the case of an abnormal termination, however, the kernel (not the process) generates a termination status to indicate the reason for the abnormal termination. In any case, the parent of the process can obtain the termination status from either the `wait` or `waitpid` function (described in the next section).

Note that we're differentiating between the "exit status" (which is the argument to either `exit` or `_exit`, or the return value from `main`) and the "termination status." The exit status is converted into a termination status by the kernel when `_exit` is finally called (recall Figure 7.1). Figure 8.2 describes the different ways the parent can examine the termination status of a child. If the child terminated normally, then the parent can obtain the exit status of the child.

When we described the `fork` function it was obvious that the child has a parent process after the call to `fork`. Now we're talking about returning a termination status to the parent, but what happens if the parent terminates before the child? The answer is that the `init` process becomes the parent process of any process whose parent terminates. We say that the process has been inherited by `init`. What normally happens is that whenever a process terminates the kernel goes through all active processes to see if the terminating process is the parent of any process that still exists. If so, the parent process ID of the still existing process is changed to be 1 (the process ID of `init`). This way we're guaranteed that every process has a parent.

Another condition we have to worry about is when the child terminates before the parent. If the child completely disappeared, the parent wouldn't be able to fetch its termination status, when (and if) the parent were finally ready to check if the child had terminated. The answer is that the kernel has to keep a certain amount of information for every terminating process, so that the information is available when the parent of the terminating process calls `wait` or `waitpid`. Minimally, this information consists of the process ID, the termination status of the process, and the amount of CPU time taken by the process. The kernel can discard all the memory used by the process and close its open files. In Unix terminology the process that has terminated, but whose parent has not yet waited for it, is called a *zombie*. The `ps`(1) command prints the state of a zombie process as Z. If we write a long running program that `fork`s many child processes, unless we wait for these processes to fetch their termination status, they become zombies.

System V provides a nonstandard way to avoid zombies, as we describe in Section 10.7.

The final condition to consider is this: what happens when a process that has been inherited by `init` terminates? Does it become a zombie? The answer is "no," because `init` is written so that whenever one of its children terminates, `init` calls one of the `wait` functions to fetch the termination status. By doing this `init` prevents the system from being clogged by zombies. When we say "one of `init`'s children" we mean either

a process that init generates directly (such as getty, which we describe in Section 9.2) or a process whose parent has terminated and has been inherited by init.

8.6 wait and waitpid Functions

When a process terminates, either normally or abnormally, the parent is notified by the kernel sending the parent the SIGCHLD signal. Since the termination of a child is an asynchronous event (it can happen at any time while the parent is running) this signal is the asynchronous notification from the kernel to the parent. The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs (a signal handler). The default action for this signal is to be ignored. We describe these options in Chapter 10. For now we need to be aware that a process that calls wait or waitpid can

- block (if all of its children are still running), or
- return immediately with the termination status of a child (if a child has terminated and is waiting for its termination status to be fetched), or
- return immediately with an error (if it doesn't have any child processes).

If the process is calling wait because it received the SIGCHLD signal, we expect wait to return immediately. But if we call it at any random point in time, it can block.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both return: process ID if OK, 0 (see later), or –1 on error

The differences between these two functions are

- wait can block the caller until a child process terminates, while waitpid has an option that prevents it from blocking.
- waitpid doesn't wait for the first child to terminate—it has a number of options that control which process it waits for.

If a child has already terminated and is a zombie, wait returns immediately with that child's status. Otherwise it blocks the caller until a child terminates. If the caller blocks and has multiple children, wait returns when one terminates. We can always tell which child terminated because the process ID is returned by the function.

For both functions the argument *statloc* is a pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the

location pointed to by the argument. If we don't care about the termination status, we just pass a null pointer as this argument.

Traditionally the integer status that is returned by these two functions has been defined by the implementation with certain bits indicating the exit status (for a normal return), other bits indicating the signal number (for an abnormal return), one bit to indicate if a core file was generated, and so on. POSIX.1 specifies that the termination status is to be looked at using various macros that are defined in `<sys/wait.h>`. There are three mutually exclusive macros that tell us how the process terminated, and they all begin with `WIF`. Based on which of these three macros is true, other macros are used to obtain the exit status, signal number, and the like. These are shown in Figure 8.2. We'll discuss how a process can be stopped in Section 9.8 when we discuss job control.

Macro	Description
`WIFEXITED` (*status*)	True if status was returned for a child that terminated normally. In this case we can execute `WEXITSTATUS` (*status*) to fetch the low-order 8 bits of the argument that the child passed to `exit` or `_exit`.
`WIFSIGNALED` (*status*)	True if status was returned for a child that terminated abnormally (by receipt of a signal that it didn't catch). In this case we can execute `WTERMSIG` (*status*) to fetch the signal number that caused the termination. Additionally, SVR4 and 4.3+BSD (but not POSIX.1) define the macro `WCOREDUMP` (*status*) that returns true if a core file of the terminated process was generated.
`WIFSTOPPED` (*status*)	True if status was returned for a child that is currently stopped. In this case we can execute `WSTOPSIG` (*status*) to fetch the signal number that caused the child to stop.

Figure 8.2 Macros to examine the termination status returned by `wait` and `waitpid`.

Example

The function `pr_exit` in Program 8.3 uses the macros from Figure 8.2 to print a description of the termination status. We'll call this function from numerous programs in the text. Note that this function handles the `WCOREDUMP` macro, if it is defined.

Program 8.4 calls the `pr_exit` function, demonstrating the different values for the termination status. If we run Program 8.4 we get

```
$ a.out
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
abnormal termination, signal number = 8 (core file generated)
```

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

void
pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
                WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
                WTERMSIG(status),
#ifdef  WCOREDUMP
                WCOREDUMP(status) ? " (core file generated)" : "");
#else
                "");
#endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
                WSTOPSIG(status));
}
```

Program 8.3 Print a description of the exit status.

Unfortunately, there is no portable way to map the signal numbers from `WTERMSIG` into descriptive names. (See Section 10.21 for one method.) We have to look at the `<signal.h>` header to verify that `SIGABRT` has a value of 6, and `SIGFPE` has a value of 8. □

As we mentioned, if we have more than one child, `wait` returns on termination of any of the children. What if we want to wait for a specific process to terminate (assuming we know which process ID we want to wait for)? In older versions of Unix we would have to call `wait` and compare the returned process ID with the one we're interested in. If the terminated process isn't the one we want, we have to save the process ID and termination status and call `wait` again. We continue doing this until the desired process terminates. The next time we want to wait for a specific process we would go through the list of already-terminated processes to see if we had already waited for it, and if not, call `wait` again. What we need is a function that waits for a specific process. This functionality (and more) is provided by the POSIX.1 `waitpid` function.

The `waitpid` function is new with POSIX.1. It is provided by both SVR4 and 4.3+BSD. Earlier releases of System V and 4.3BSD, however, didn't support it.

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

int
main(void)
{
    pid_t   pid;
    int     status;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)          /* child */
        exit(7);

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)          /* child */
        abort();                /* generates SIGABRT */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)          /* child */
        status /= 0;            /* divide by 0 generates SIGFPE */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    exit(0);
}
```

Program 8.4 Demonstrate different `exit` statuses.

Constant	Description
WNOHANG	`waitpid` will not block if a child specified by *pid* is not immediately available. In this case the return value is 0.
WUNTRACED	If the implementation supports job control, the status of any child specified by *pid* that has stopped, and whose status has not been reported since it has stopped, is returned. The `WIFSTOPPED` macro determines if the return value corresponds to a stopped child process.

Figure 8.3 The *options* constants for `waitpid`.

The interpretation of the *pid* argument for `waitpid` depends on its value:

pid == –1	waits for any child process. In this respect, `waitpid` is equivalent to `wait`.
pid > 0	waits for the child whose process ID equals *pid*.
pid == 0	waits for any child whose process group ID equals that of the calling process.
pid < –1	waits for any child whose process group ID equals the absolute value of *pid*.

(We describe process groups in Section 9.4.) `waitpid` returns the process ID of the child that terminated, and its termination status is returned through *statloc*. With `wait` the only error is if the calling process has no children. (Another error return is possible, in case the function call is interrupted by a signal. We'll discuss this in Chapter 10.) With `waitpid`, however, it's also possible to get an error if the specified process or process group does not exist or is not a child of the calling process.

The *options* argument lets us further control the operation of `waitpid`. This argument is either 0 or is constructed from the bitwise OR of the constants in Figure 8.3.

> SVR4 supports two additional, but nonstandard, *option* constants. `WNOWAIT` has the system keep the process whose termination status is returned by `waitpid` in a wait state, so that it may be waited for again. With `WCONTINUED`, the status of any child specified by *pid* that has been continued, and whose status has not been reported, is returned.

The `waitpid` function provides three features that aren't provided by the `wait` function.

1. `waitpid` lets us wait for one particular process (whereas `wait` returns the status of any terminated child). We'll return to this feature when we discuss the `popen` function.
2. `waitpid` provides a nonblocking version of `wait`. There are times when we want to fetch a child's status, but we don't want to block.
3. `waitpid` supports job control (with the `WUNTRACED` option).

Example

Recall our discussion in Section 8.3 about zombie processes. If we want to write a process so that it `fork`s a child but we don't want to wait for the child to complete and we don't want the child to become a zombie until we terminate, the trick is to call `fork` twice. Program 8.5 does this.

We call `sleep` in the second child to assure that the first child terminates before printing the parent process ID. After a fork either the parent or child can continue executing—we never know which will resume execution first. If we didn't put the second child to sleep, and if it resumed execution after the `fork` before its parent, the parent process ID that it printed would be that of its parent, not process ID 1.

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

int
main(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {        /* first child */
        if ( (pid = fork()) < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit(0);    /* parent from second fork == first child */

        /* We're the second child; our parent becomes init as soon
           as our real parent calls exit() in the statement above.
           Here's where we'd continue executing, knowing that when
           we're done, init will reap our status. */

        sleep(2);
        printf("second child, parent pid = %d\n", getppid());
        exit(0);
    }

    if (waitpid(pid, NULL, 0) != pid)   /* wait for first child */
        err_sys("waitpid error");

    /* We're the parent (the original process); we continue executing,
       knowing that we're not the parent of the second child. */

    exit(0);
}
```

Program 8.5 Avoid zombie processes by forking twice.

Executing Program 8.5 gives us

```
$ a.out
$ second child, parent pid = 1
```

Note that the shell prints its prompt when the original process terminates, which is before the second child prints its parent process ID. □

8.7 `wait3` and `wait4` Functions

4.3+BSD provides two additional functions, `wait3` and `wait4`. The only feature provided by these two functions that isn't provided by the POSIX.1 functions `wait` and

`waitpid` is an additional argument that allows the kernel to return a summary of the resources used by the terminated process and all its child processes.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

pid_t wait3(int *statloc, int options, struct rusage *rusage);

pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```

Both return: process ID if OK, 0, or –1 on error

SVR4 also provides the `wait3` function in the BSD compatibility library.

The resource information includes information such as the amount of user CPU time, the amount of system CPU time, number of page faults, number of signals received, and the like. Refer to the `getrusage`(2) manual page for additional details. This resource information is available only for terminated child processes, not for stopped child processes. (This resource information differs from the resource limits we described in Section 7.11.) Figure 8.4 details the different arguments supported by the various wait functions.

Function	*pid*	*options*	*rusage*	POSIX.1	SVR4	4.3+BSD
`wait`				•	•	•
`waitpid`	•	•		•	•	•
`wait3`		•	•		•	•
`wait4`	•	•	•			•

Figure 8.4 Arguments supported by various wait functions on different systems.

8.8 Race Conditions

For our purposes a race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run. The `fork` function is a lively breeding ground for race conditions, if any of the logic after the `fork` either explicitly or implicitly depends on whether the parent or child runs first after the `fork`. In general we cannot predict which process runs first. Even if we knew which process would run first, what happens after that process starts running depends on the system load and the kernel's scheduling algorithm.

We saw a potential race condition in Program 8.5 when the second child printed its parent process ID. If the second child runs before the first child, then its parent process will be the first child. But if the first child runs first and has enough time to `exit`, then the parent process of the second child is `init`. Even calling `sleep`, as we did, guarantees nothing. If the system was heavily loaded, the second child could resume after

sleep returns, before the first child has a chance to run. Problems of this form can be hard to debug because they tend to work "most of the time."

If a process wants to wait for a child to terminate, it must call one of the wait functions. If a process wants to wait for its parent to terminate, as in Program 8.5, a loop of the following form could be used

```
while (getppid() != 1)
    sleep(1);
```

The problem with this type of loop (called *polling*) is that it wastes CPU time, since the caller is woken up every second to test the condition.

To avoid race conditions and to avoid polling, some form of signaling is required between multiple processes. Signals can be used, and we describe one way to do this in Section 10.16. Various forms of interprocess communication (IPC) can also be used. We'll discuss some of these in Chapters 14 and 15.

For a parent and child relationship, we often have the following scenario. After the fork both the parent and child have something to do. For example, the parent could update a record in a log file with the child's process ID, and the child might have to create a file for the parent. In this example we require that each process tell the other when it has finished its initial set of operations, and that each wait for the other to complete, before heading off on its own. The scenario is

```
#include  "ourhdr.h"

TELL_WAIT();    /* set things up for TELL_xxx & WAIT_xxx */

if ( (pid = fork()) < 0)
    err_sys("fork error");
else if (pid == 0) {            /* child */

    /* child does whatever is necessary ... */

    TELL_PARENT(getppid());     /* tell parent we're done */

    WAIT_PARENT();              /* and wait for parent */

    /* and the child continues on its way ... */
    exit(0);
}

/* parent does whatever is necessary ... */

TELL_CHILD(pid);            /* tell child we're done */

WAIT_CHILD();               /* and wait for child */

/* and the parent continues on its way ... */
exit(0);
```

We assume that the header ourhdr.h defines whatever variables are required. The five routines TELL_WAIT, TELL_PARENT, TELL_CHILD, WAIT_PARENT, and WAIT_CHILD can be either macros or actual functions.

We'll show various ways to implement these TELL and WAIT routines in later chapters: Section 10.16 shows an implementation using signals, Section 15.2 shows an implementation using stream pipes. Let's look at an example that uses these five routines.

Example

Program 8.6 outputs two strings: one from the child and one from the parent. It contains a race condition because the output depends on the order in which the processes are run by the kernel and for how long each process runs.

```
#include    <sys/types.h>
#include    "ourhdr.h"

static void charatatime(char *);

int
main(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
    }
    exit(0);
}

static void
charatatime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; c = *ptr++; )
        putc(c, stdout);
}
```

Program 8.6 Program with a race condition.

We set the standard output unbuffered, so that every character output generates a write. The goal in this example is to allow the kernel to switch between the two processes as often as possible to demonstrate the race condition. (If we didn't do this we might never see the type of output that follows. Not seeing the erroneous output doesn't mean that the race condition doesn't exist, it just means that we can't see it on this particular system.) The following actual output shows how the results can vary.

```
$ a.out
output from child
output from parent
$ a.out
oouuttppuutt  ffrroomm  cphairledn
t
$ a.out
oouuttppuutt  ffrroomm  pcahrielndt

$ a.out
ooutput from parent
utput from child
```

We need to change Program 8.6 to use the `TELL` and `WAIT` functions. Program 8.7 does this. The lines preceded by a plus sign are new lines.

```
  #include    <sys/types.h>
  #include    "ourhdr.h"

  static void charatatime(char *);

  int
  main(void)
  {
      pid_t   pid;

+     TELL_WAIT();
+
      if ( (pid = fork()) < 0)
          err_sys("fork error");
      else if (pid == 0) {
+         WAIT_PARENT();      /* parent goes first */
          charatatime("output from child\n");
      } else {
          charatatime("output from parent\n");
+         TELL_CHILD(pid);
      }
      exit(0);
  }

  static void
  charatatime(char *str)
  {
      char    *ptr;
      int     c;

      setbuf(stdout, NULL);           /* set unbuffered */
      for (ptr = str; c = *ptr++; )
          putc(c, stdout);
  }
```

Program 8.7 Modification of Program 8.6 to avoid race condition.

When we run this program the output is as we expect—there is no intermixing of output from the two processes.

In Program 8.7 the parent goes first. If we change the lines following the `fork` to be

```
else if (pid == 0) {
    charatatime("output from child\n");
    TELL_PARENT(getppid());
} else {
    WAIT_CHILD();        /* child goes first */
    charatatime("output from parent\n");
}
```

the child goes first. Exercise 8.3 continues this example. □

8.9 exec Functions

We mentioned in Section 8.3 that one use of the `fork` function was to create a new process (the child) that then causes another program to be executed by calling one of the `exec` functions. When a process calls one of the `exec` functions, that process is completely replaced by the new program, and the new program starts executing at its `main` function. The process ID does not change across an `exec` because a new process is not created. `exec` merely replaces the current process (its text, data, heap, and stack segments) with a brand new program from disk.

There are six different `exec` functions, but we'll often just refer to "the `exec` function," which means we could use any of the six different functions. These six functions round out the Unix process control primitives. With `fork` we can create new processes, and with the `exec` functions we can initiate new programs. The `exit` function and the two `wait` functions handle termination and waiting for termination. These are the only process control primitives we need. We'll use these primitives in later sections to build additional functions such as `popen` and `system`.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *) 0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *) 0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *) 0 */ );

int execvp(const char *filename, char *const argv[]);
```

All six return: –1 on error, no return on success

The first difference in these functions is that the first four take a pathname argument while the last two take a filename argument. When a *filename* argument is specified

- if *filename* contains a slash, it is taken as a pathname,
- otherwise, the executable file is searched for in the directories specified by the `PATH` environment variable.

The `PATH` variable contains a list of directories (called path prefixes) that are separated by colons. For example, the *name=value* environment string

```
PATH=/bin:/usr/bin:/usr/local/bin/:.
```

specifies four directories to search. (A zero-length prefix also means the current directory. It can be specified as a colon at the beginning of the *value*, two colons in a row, or a colon at the end of the *value*.)

> There are security reasons for *never* including the current directory in the search path. See Garfinkel and Spafford [1991].

If either of the two functions, `execlp` or `execvp`, finds an executable file using one of the path prefixes, but the file isn't a machine executable that was generated by the link editor, it assumes the file is a shell script and tries to invoke `/bin/sh` with the *filename* as input to the shell.

The next difference concerns the passing of the argument list (`l` stands for list and `v` stands for vector). The functions `execl`, `execlp`, and `execle` require each of the command-line arguments to the new program to be specified as separate arguments. We mark the end of the arguments with a null pointer. For the other three functions (`execv`, `execvp`, and `execve`) we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.

Before using ANSI C prototypes, the normal way to show the command-line arguments for the three functions `execl`, `execle`, and `execlp` was

```
char *arg0, char *arg1, ..., char *argn, (char *) 0
```

This specifically shows that the final command-line argument is followed by a null pointer. If this null pointer is specified by the constant `0`, we must explicitly cast it to a pointer, because if we don't it's interpreted as an integer argument. If the size of an integer is different from the size of a `char *`, the actual arguments to the `exec` function will be wrong.

The final difference is the passing of the environment list to the new program. The two functions whose name ends in an `e` (`execle` and `execve`) allow us to pass a pointer to an array of pointers to the environment strings. The other four functions, however, use the `environ` variable in the calling process to copy the existing environment for the new program. (Recall our discussion of the environment strings in Section 7.9 and Figure 7.5. We mentioned that if the system supported functions such as `setenv` and `putenv` we could change the current environment and the environment of any subsequent child processes, but we couldn't affect the environment of the parent

process.) Normally a process allows its environment to be propagated to its children, but there are cases when a process wants to specify a certain environment for a child. One example of the latter is the `login` program when a new login shell is initiated. Normally `login` creates a specific environment with only a few variables defined and lets us, through the shell start-up file, add variables to the environment when we log in.

Before using ANSI C prototypes, the arguments to `execle` were shown as

```
char *pathname, char *arg0, ..., char *argn, (char *) 0, char *envp[]
```

This specifically shows that the final argument is the address of the array of character pointers to the environment strings. The ANSI C prototype doesn't show this, since all the command-line arguments, the null pointer, and the *envp* pointer are shown with the ellipsis notation (`...`).

The arguments for these six `exec` functions are hard to remember. The letters in the function names help somewhat. The letter `p` means the function takes a *filename* argument and uses the `PATH` environment variable to find the executable file. The letter `l` means the function takes a list of arguments and is mutually exclusive with the letter `v`, which means it takes an *argv*`[]` vector. Finally the letter `e` means the function takes an *envp*`[]` array, instead of using the current environment. Figure 8.5 shows the differences between these six functions.

Function	*pathname*	*filename*	Arg list	*argv*`[]`	`environ`	*envp*`[]`
`execl`	•		•		•	
`execlp`		•	•		•	
`execle`	•		•			•
`execv`	•			•	•	
`execvp`		•		•	•	
`execve`	•			•		•
(letter in name)		`p`	`l`	`v`		`e`

Figure 8.5 Differences between the six `exec` functions.

Every system has a limit on the total size of the argument list and the environment list. From Figure 2.7 this limit is given by `ARG_MAX`. This value must be at least 4096 bytes on a POSIX.1 system. We sometimes encounter this limit when using the shell's filename expansion feature to generate a list of filenames. For example, the command

```
grep _POSIX_SOURCE /usr/include/*/*.h
```

can generate a shell error of the form

```
arg list too long
```

on some systems.

> Historically, System V has had a limit of 5120 bytes. 4.3BSD and 4.3+BSD are distributed with a limit of 20,480 bytes. The system used by the author (see the output from Program 2.1) allows up to a megabyte!

We've mentioned that the process ID does not change after an `exec`, but there are additional properties that the new program inherits from the calling process:

- process ID and parent process ID
- real user ID and real group ID
- supplementary group IDs
- process group ID
- session ID
- controlling terminal
- time left until alarm clock
- current working directory
- root directory
- file mode creation mask
- file locks
- process signal mask
- pending signals
- resource limits
- `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_ustime` values

The handling of open files depends on the value of the close-on-exec flag for each descriptor. Recall from Figure 3.2 and our mention of the `FD_CLOEXEC` flag in Section 3.13, that every open descriptor in a process has a close-on-exec flag. If this flag is set, the descriptor is closed across an `exec`. Otherwise the descriptor is left open across the `exec`. Note that the default is to leave the descriptor open across the `exec`, unless we specifically set the close-on-exec flag using `fcntl`.

POSIX.1 specifically requires that open directory streams (recall the `opendir` function from Section 4.21) be closed across an `exec`. This is normally done by the `opendir` function calling `fcntl` to set the close-on-exec flag for the descriptor corresponding to the open directory stream.

Note that the real user ID and the real group ID remain the same across the `exec`, but the effective IDs can change, depending on the status of the set-user-ID and the set-group-ID bits for the program file that is executed. If the set-user-ID bit is set for the new program, the effective user ID becomes the owner ID of the program file. Otherwise the effective user ID is not changed (it's not set to the real user ID). The group ID is handled in the same way.

In many Unix implementations only one of these six functions, `execve`, is a system call within the kernel. The other five are just library functions that eventually invoke this system call. We can picture the relationship between these six functions as shown in Figure 8.6. In this arrangement the library functions `execlp` and `execvp` process the `PATH` environment variable, looking for the first path prefix that contains an executable file named *filename*.

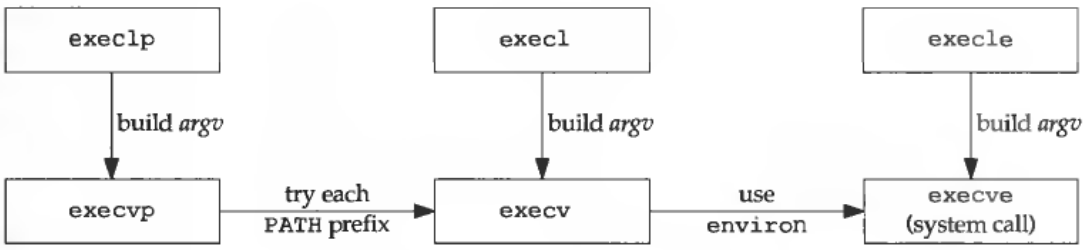


Figure 8.6 Relationship of the six exec functions.

Example

Program 8.8 demonstrates the exec functions.

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {    /* specify pathname, specify environment */
        if (execle("/home/stevens/bin/echoall",
                   "echoall", "myarg1", "MY ARG2", (char *) 0,
                   env_init) < 0)
            err_sys("execle error");
    }
    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {    /* specify filename, inherit environment */
        if (execlp("echoall",
                   "echoall", "only 1 arg", (char *) 0) < 0)
            err_sys("execlp error");
    }
    exit(0);
}
```

Program 8.8 Example of exec functions.

We first call `execle`, which requires a pathname and a specific environment. The next call is to `execlp`, which uses a filename and passes the caller's environment to the new program. The only reason the call to `execlp` works is because the directory `/home/stevens/bin` is one of the current path prefixes. Note also that we set the first argument, `argv[0]` in the new program, to be the filename component of the pathname. Some shells set this argument to be the complete pathname.

The program `echoall` that is `exec`ed twice in Program 8.8 is shown in Program 8.9. It is a trivial program that echoes all its command-line arguments and its entire environment list.

```
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    int         i;
    char        **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++)   /* and all env strings */
        printf("%s\n", *ptr);

    exit(0);
}
```

Program 8.9 Echo all command-line arguments and all environment strings.

When we execute Program 8.8 we get

```
$ a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
argv[0]: echoall
$ argv[1]: only 1 arg
USER=stevens
HOME=/home/stevens
LOGNAME=stevens
                                31 more lines that aren't shown
EDITOR=/usr/ucb/vi
```

Notice that the shell prompt appeared between the printing of `argv[0]` and `argv[1]` from the second `exec`. This is because the parent did not `wait` for this child process to finish. □

8.10 Changing User IDs and Group IDs

We can set the real user ID and effective user ID with the `setuid` function. Similarly we can set the real group ID and the effective group ID with the `setgid` function.

```
#include <sys/types.h>
#include <unistd.h>

int setuid(uid_t uid);

int setgid(gid_t gid);
```

Both return: 0 if OK, –1 on error

There are rules for who can change the IDs. Let's consider only the user ID for now. (Everything we describe for the user ID also applies to the group ID.)

1. If the process has superuser privileges, the `setuid` function sets the real user ID, effective user ID, and saved set-user-ID to *uid*.
2. If the process does not have superuser privileges, but *uid* equals either the real user ID or the saved set-user-ID, `setuid` sets only the effective user ID to *uid*. The real user ID and the saved set-user-ID are not changed.
3. If neither of these two conditions is true, `errno` is set to `EPERM` and an error is returned.

Here we are assuming that `_POSIX_SAVED_IDS` is true. If this feature isn't provided, then delete all references above to the saved set-user-ID.

FIPS 151–1 requires this feature.

SVR4 supports the `_POSIX_SAVED_IDS` feature.

We can make a couple of statements about the three user IDs that the kernel maintains.

1. Only a superuser process can change the real user ID. Normally the real user ID is set by the `login`(1) program when we log in and never changes. Since `login` is a superuser process, when it calls `setuid` it sets all three user IDs.
2. The effective user ID is set by the `exec` functions, only if the set-user-ID bit is set for the program file. If the set-user-ID bit is not set, the `exec` functions leave the effective user ID as its current value. We can call `setuid` at any time to set the effective user ID to either the real user ID or the saved set-user-ID. Naturally, we can't set the effective user ID to any random value.
3. The saved set-user-ID is copied from the effective user ID by `exec`. This copy is saved after `exec` stores the effective user ID from the file's user ID (if the file's set-user-ID bit is set).

Figure 8.7 summarizes the different ways these three user IDs can be changed.

ID	exec		setuid(*uid*)	
	set-user-ID bit off	set-user-ID bit on	superuser	unprivileged user
real user ID	unchanged	unchanged	set to *uid*	unchanged
effective user ID	unchanged	set from user ID of program file	set to *uid*	set to *uid*
saved set-user-ID	copied from effective user ID	copied from effective user ID	set to *uid*	unchanged

Figure 8.7 Different ways to change the three user IDs.

Note that we can obtain only the current value of the real user ID and the effective user ID with the functions `getuid` and `geteuid` from Section 8.2. We can't obtain the current value of the saved set-user-ID.

Example

To see the utility of the saved set-user-ID feature, let's examine the operation of a program that uses it. We'll look at the Berkeley `tip`(1) program. (The System V `cu`(1) program is similar.) Both programs connect to a remote system, either through a direct connection or by dialing a modem. When `tip` uses a modem, it has to obtain exclusive use of the modem through the use of a lock file. This lock file is also shared with the UUCP program, since both programs can want to use the same modem at the same time. The following steps take place.

1. The `tip` program file is owned by the user name `uucp` and has its set-user-ID bit set. When we `exec` it, we have

   real user ID = our user ID
   effective user ID = `uucp`
   saved set-user-ID = `uucp`

2. `tip` accesses the required lock files. These lock files are owned by the user name `uucp`, but since the effective user ID is `uucp`, file access is allowed.
3. `tip` executes `setuid(getuid())`. Since we are not a superuser process, this changes only the effective user ID. We have

   real user ID = our user ID (unchanged)
   effective user ID = our user ID
   saved set-user-ID = `uucp` (unchanged)

   Now the `tip` process is running with our user ID as its effective user ID. This means we can access only the files that we have normal access to. We have no additional permissions.

4. When we are done, `tip` executes `setuid(`*uucpuid*`)`, where *uucpuid* is the numerical user ID for the user name `uucp`. (This was probably saved by `tip` when it started by calling `geteuid`. We are not implying that it searches the password file for this numerical user ID.) This call is allowed because the argument to `setuid` equals the saved set-user-ID. (This is why we need the saved set-user-ID.) Now we have

   real user ID = our user ID (unchanged)
   effective user ID = `uucp`
   saved set-user-ID = `uucp` (unchanged)

5. `tip` can now operate on its lock files, to release them, since its effective user ID is `uucp`.

By using the saved set-user-ID in this fashion, we can use the extra privileges allowed us by the set-user-ID of the program file at the beginning of the process and at the end of the process. Most of the time the process is running, however, it runs with our normal permissions. If we weren't able to switch back to the saved set-user-ID at the end, we might be tempted to retain the extra permissions the whole time we were running (which is asking for trouble).

Let's look at what happens if `tip` spawns a shell for us while its running. (The shell is spawned using `fork` and `exec`.) Since the real user ID and effective user ID are both our normal user ID (step 3 above), the shell has no extra permissions. The shell can't access the saved set-user-ID that is set to `uucp` while `tip` is running, because the saved set-user-ID for the shell is copied from the effective user ID by `exec`. So in the child process that does the `exec`, all three user IDs are our normal user ID.

Our description of how the `setuid` function is used by `tip` is not correct if the program is set-user-ID to root. This is because a call to `setuid` with superuser privileges sets all three user IDs. For the example to work as described, we need `setuid` to set only the effective user ID. □

`setreuid` and `setregid` Functions

4.3+BSD supports the swapping of the real user ID and the effective user ID with the `setreuid` function.

```
#include <sys/types.h>
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);

int setregid(gid_t rgid, gid_t egid);
```

Both return: 0 if OK, –1 on error

The rule is simple: an unprivileged user can always swap between the real user ID and the effective user ID. This allows a set-user-ID program to swap to the user's normal permissions and swap back again later for set-user-ID operations. When the saved set-user-ID feature was introduced with POSIX.1, the rule was enhanced to also allow an unprivileged user to set its effective user ID to its saved set-user-ID.

SVR4 also provides these two functions in the BSD compatibility library.

4.3BSD didn't have the saved set-user-ID feature described earlier. It used `setreuid` and `setregid` instead. This allowed an unprivileged user to swap back and forth between the two values, and the `tip` program under 4.3BSD was written to use this feature. Be aware, however, that when this version of `tip` spawned a shell it had to set the real user ID to the normal user ID before the `exec`. If it didn't do this, the real user ID could be uucp (from the swap done by `setreuid`) and the shell process could call `setreuid` to swap the two and assume the permissions of uucp. `tip` sets both the real user ID and the effective user ID to the normal user ID in the child as a defensive programming measure.

`seteuid` and `setegid` Functions

A proposed change to POSIX.1 includes the two functions `seteuid` and `setegid`. Only the effective user ID or effective group ID is changed.

```
#include <sys/types.h>
#include <unistd.h>

int seteuid(uid_t uid);

int setegid(gid_t gid);
```

Both return: 0 if OK, –1 on error

An unprivileged user can set its effective user ID to either its real user ID or its saved set-user-ID. For a privileged user only the effective user ID is set to *uid*. (This differs from the `setuid` function, which changes all three user IDs.) This proposed POSIX.1 change also requires that the saved set-user-ID always be supported.

Both SVR4 and 4.3+BSD support these two functions.

Figure 8.8 summarizes all the functions that we've described in this section that modify the three different user IDs.

Group IDs

Everything that we've said so far in this section also applies in a similar fashion to group IDs. The supplementary group IDs are not affected by the `setgid` function.

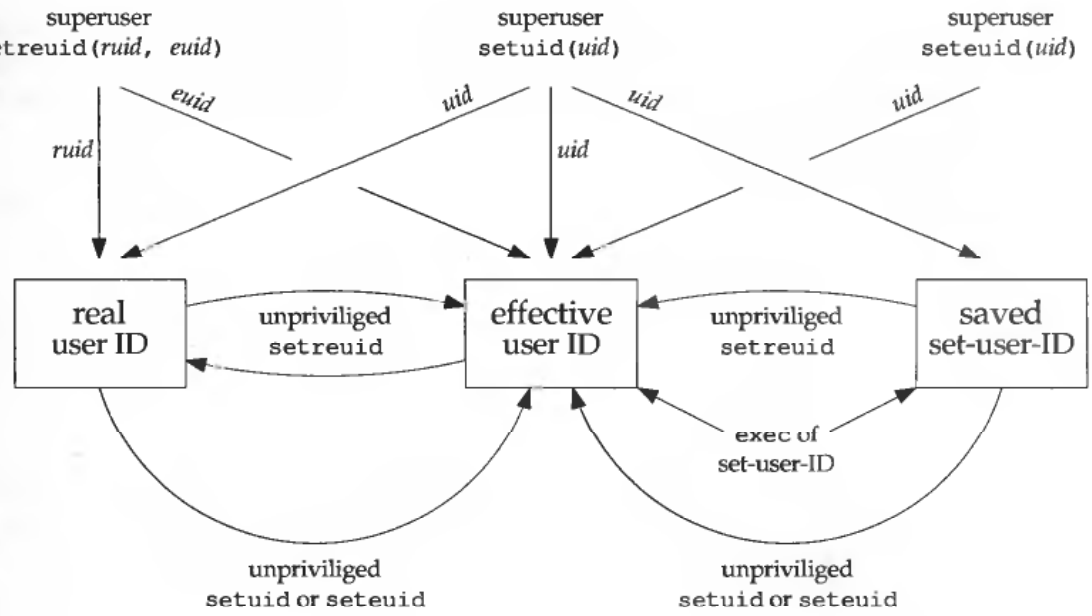


Figure 8.8 Summary of all the functions that set the different user IDs.

8.11 Interpreter Files

Both SVR4 and 4.3+BSD support interpreter files. These files are text files that begin with a line of the form

```
#! pathname [ optional-argument ]
```

The space between the exclamation point and the *pathname* is optional. The most common of these begin with the line

```
#!/bin/sh
```

The *pathname* is normally an absolute pathname, since no special operations are performed on it (i.e., `PATH` is not used). The recognition of these files is done within the kernel as part of processing the `exec` system call. The actual file that gets `exec`ed by the kernel is not the interpreter file, but the file specified by the *pathname* on the first line of the interpreter file. Be sure to differentiate between the interpreter file (a text file that begins with `#!`) and the interpreter (specified by the *pathname* on the first line of the interpreter file).

Be aware that many systems have a limit of 32 characters for the first line of an interpreter file. This includes the `#!`, the *pathname*, the optional argument, and any spaces.

Example

Let's look at an actual example, to see what the kernel does with the arguments to the exec function when the file being execed is an interpreter file and the optional argument on the first line of the interpreter file. Program 8.10 execs an interpreter file.

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

int
main(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {            /* child */
        if (execl("/home/stevens/bin/testinterp",
                  "testinterp", "myarg1", "MY ARG2", (char *) 0) < 0)
            err_sys("execl error");
    }

    if (waitpid(pid, NULL, 0) < 0)  /* parent */
        err_sys("waitpid error");
    exit(0);
}
```

Program 8.10 A program that execs an interpreter file.

The following shows the contents of the one line interpreter file that is execed, and the result from running Program 8.10:

```
$ cat /home/stevens/bin/testinterp
#!/home/stevens/bin/echoarg foo
$ a.out
argv[0]: /home/stevens/bin/echoarg
argv[1]: foo
argv[2]: /home/stevens/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

The program `echoarg` (the interpreter) just echoes each of its command-line arguments. (This is Program 7.2.) Note that when the kernel `execs` the interpreter (`/home/stevens/bin/echoarg`), `argv[0]` is the *pathname* of the interpreter, `argv[1]` is the optional argument from the interpreter file, and the remaining arguments are the *pathname* (`/home/stevens/bin/testinterp`) and the second and third arguments from the call to `execl` in Program 8.10 (`myarg1` and `MY ARG2`). `argv[1]` and `argv[2]` from the call to `execl` have been shifted right two positions. Note that the kernel takes the *pathname* from the `execl` call, instead of the first argument (`testinterp`), on the assumption that the *pathname* might contain more information than the first argument. □

Example

A common use for the optional argument following the interpreter *pathname* is to specify the `-f` option for programs that support this option. For example, an awk(1) program can be executed as

```
awk -f myfile
```

which tells awk to read the awk program from the file `myfile`.

> There are two versions of the awk language on many systems. awk is often called "old awk" and corresponds to the original version distributed with Version 7. nawk (new awk) contains numerous enhancements and corresponds to the language described in Aho, Kernighan, and Weinberger [1988]. This newer version provides access to the command-line arguments, which we need for the example that follows. SVR4 provides both, with awk and oawk being the same, and a note that awk will be nawk in a future release. The POSIX.2 draft specifies the new language as just awk, and that's what we'll use in this text.

Using the `-f` option with an interpreter file lets us write

```
#!/bin/awk -f
(awk program follows in the interpreter file)
```

For example, Program 8.11 shows `/usr/local/bin/awkexample` (an interpreter file).

```
#!/bin/awk -f

BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}
```

Program 8.11 An awk program as an interpreter file.

If one of the path prefixes is `/usr/local/bin`, we can execute Program 8.11 (assuming we've turned on the execute bit for the file) as

```
$ awkexample file1 FILENAME2 f3
ARGV[0] = /bin/awk
ARGV[1] = file1
ARGV[2] = FILENAME2
ARGV[3] = f3
```

When `/bin/awk` is executed, its command-line arguments are

```
/bin/awk -f /usr/local/bin/awkexample file1 FILENAME2 f3
```

The pathname of the interpreter file (`/usr/local/bin/awkexample`) is passed to the interpreter. The filename portion of this pathname (what we typed to the shell) isn't adequate, because the interpreter (`/bin/awk` in this example) can't be expected to use the `PATH` variable to locate files. When awk reads the interpreter file it ignores the first line, since the pound sign is awk's comment character.

We can verify these command-line arguments with the following commands.

```
$ su                                        become superuser
Password:                                   enter superuser password
# mv /bin/awk /bin/awk.save                 save the original program
# cp /home/stevens/bin/echoarg /bin/awk     and replace it temporarily
# suspend                                   suspend the superuser shell using job control
[1] + Stopped          su
$ awkexample file1 FILENAME2 f3
argv[0]: /bin/awk
argv[1]: -f
argv[2]: /usr/local/bin/awkexample
argv[3]: file1
argv[4]: FILENAME2
argv[5]: f3
$ fg                                        resume superuser shell using job control
su
# mv /bin/awk.save /bin/awk                 restore the original program
# exit                                      and exit the superuser shell
```

In this example the `-f` option for the interpreter is required. As we said, this tells awk where to look for the awk program. If we remove the `-f` option from the interpreter file, the results are

```
$ awkexample file1 FILENAME2 f3
/bin/awk: syntax error at source line 1
 context is
        >>> /usr/local <<< /bin/awkexample
/bin/awk: bailing out at source line 1
```

This is because the command-line arguments in this case are

```
/bin/awk /usr/local/bin/awkexample file1 FILENAME2 f3
```

and awk is trying to interpret the string `/usr/local/bin/awkexample` as an awk program. If we couldn't pass at least a single optional argument to the interpreter (`-f` in this case), these interpreter files would be usable only with the shells. □

Are interpreter files required? Not really. They provide an efficiency gain for the user at some expense in the kernel (since it's the kernel that recognizes these files). Interpreter files are useful for the following reasons.

1. They hide the fact that certain programs are scripts in some other language. For example, to execute Program 8.11 we just say

   ```
   awkexample optional-arguments
   ```

   instead of needing to know that the program is really an awk script that we would otherwise have to execute as

   ```
   awk -f awkexample optional-arguments
   ```

2. Interpreter scripts provide an efficiency gain. Consider the previous example again. We could still hide the fact that the program is an awk script, by wrapping it in a shell script:

```
awk 'BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}' $*
```

The problem with this solution is that more work is required. First the shell reads the command and tries to `execlp` the filename. Since the shell script is an executable file, but isn't a machine executable, an error is returned and `execlp` assumes the file is a shell script (which it is). Then `/bin/sh` is execed with the pathname of the shell script as its argument. The shell correctly runs our script, but to run the `awk` program, it does a `fork`, `exec`, and `wait`. There is more overhead in replacing an interpreter script with a shell script.

3. Interpreter scripts let us write shell scripts using shells other than `/bin/sh`. When `execlp` finds an executable file that isn't a machine executable it has to choose a shell to invoke, and it always uses `/bin/sh`. Using an interpreter script, however, we can just write

```
#!/bin/csh
(C shell script follows in the interpreter file)
```

Again, we could wrap this all in a `/bin/sh` script (that invokes the C shell), as we described earlier, but more overhead is required.

None of this would work as we've shown if the three shells and awk didn't use the pound sign as their comment character.

8.12 `system` Function

It is convenient to execute a command string from within a program. For example, assume we want to put a time and date stamp into a certain file. We could use the functions we describe in Section 6.9 to do this—call `time` to get the current calendar time, next call `localtime` to convert it to a broken-down time, then call `strftime` to format the result, and write the results to the file. It is much easier, however, to say

```
system("date > file");
```

ANSI C defines the `system` function, but its operation is strongly system dependent.

> The `system` function is not defined by POSIX.1 because it is not an interface to the operating system, but really an interface to a shell. Therefore it is being standardized by POSIX.2. The following description corresponds to Draft 11.2 of the POSIX.2 standard.

```
#include <stdlib.h>

int system(const char *cmdstring);
```

Returns: (see below)

If *cmdstring* is a null pointer, `system` returns nonzero only if a command processor is available. This feature determines if the `system` function is supported on a given operating system. Under Unix `system` is always available.

Since `system` is implemented by calling `fork`, `exec`, and `waitpid`, there are three different types of return values:

1. If either the `fork` fails or `waitpid` returns an error other than `EINTR`, `system` returns −1 with `errno` set to indicate the error.
2. If the `exec` fails (implying that the shell can't be executed) the return value is as if the shell had executed `exit(127)`.
3. Otherwise all three functions succeed (`fork`, `exec`, and `waitpid`) and the return value from `system` is the termination status of the shell, in the format specified for `waitpid`.

> Many current implementations of `system` return an error if `waitpid` is interrupted by a caught signal (`EINTR`). The requirement that `system` not return an error in this case was added to a recent draft of POSIX.2. (We discuss interrupted system calls in Section 10.5.)

Program 8.12 is an implementation of the `system` function. The one feature that it doesn't handle is signals. We'll update this function with signal handling in Section 10.18.

The shell's `-c` option tells it to take the next command-line argument (*cmdstring* in this case) as its command input (instead of reading from standard input or from a given file). The shell parses this null terminated C string and breaks it up into separate command-line arguments for the actual command. The actual command string that is passed to the shell can contain any valid shell commands. For example, input and output redirection using < and > can be used.

If we didn't use the shell to execute the command, but tried to execute the command ourself, it would be harder. First, we would want to call `execlp` instead of `execl`, to use the `PATH` variable, like the shell. We would also have to break up the null terminated C string into separate command-line arguments for the call to `execlp`. Finally, we wouldn't be able to use any of the shell metacharacters.

Note that we call `_exit` instead of `exit`. This is to prevent any standard I/O buffers (which would have been copied from the parent to the child across the `fork`) from being flushed in the child.

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    <errno.h>
#include    <unistd.h>

int
system(const char *cmdstring)    /* version without signal handling */
{
    pid_t   pid;
    int     status;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with Unix */

    if ( (pid = fork()) < 0) {
        status = -1;    /* probably out of processes */

    } else if (pid == 0) {              /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
        _exit(127);     /* execl error */

    } else {                            /* parent */
        while (waitpid(pid, &status, 0) < 0)
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
    }

    return(status);
}
```

Program 8.12 The system function (without signal handling).

We can test this version of system with Program 8.13. (The pr_exit function was defined in Program 8.3.) Running Program 8.13 gives us

```
$ a.out
Thu Aug 29 14:24:19 MST 1991
normal termination, exit status = 0      for date
sh: nosuchcommand: not found
normal termination, exit status = 1      for nosuchcommand
stevens  console Aug 25 11:49
stevens  ttyp0   Aug 29 05:56
stevens  ttyp1   Aug 29 05:56
stevens  ttyp2   Aug 29 05:56
normal termination, exit status = 44     for exit
```

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

int
main(void)
{
    int     status;

    if ( (status = system("date")) < 0)
        err_sys("system() error");
    pr_exit(status);

    if ( (status = system("nosuchcommand")) < 0)
        err_sys("system() error");
    pr_exit(status);

    if ( (status = system("who; exit 44")) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

Program 8.13 Calling the `system` function.

The advantage in using `system`, instead of using `fork` and `exec` directly, is that `system` does all the required error handling and (in our next version of this function in Section 10.18) all the required signal handling.

Earlier versions of Unix, including SVR3.2 and 4.3BSD, didn't have the `waitpid` function available. Instead, the parent waited for the child, using a statement such as

```
while ((lastpid = wait(&status)) != pid && lastpid != -1)
    ;
```

A problem occurs if the process that calls `system` has spawned its own children before calling `system`. Since the `while` statement above keeps looping until the child that was generated by `system` terminates, if any children of the process terminate before the process identified by `pid`, then the process ID and termination status of these other children is just discarded by the `while` statement. Indeed, this inability to `wait` for a specific child is one of the reasons given in the POSIX.1 Rationale for including the `waitpid` function. We'll see in Section 14.3 that the same problem occurs with the `popen` and `pclose` functions, if the system doesn't provide a `waitpid` function.

Set-User-ID Programs

What happens if we call `system` from a set-user-ID program? This is a security hole and should never be done. Program 8.14 is a simple program that just calls `system` for its command-line argument.

```
#include    "ourhdr.h"

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

    if (argc < 2)
        err_quit("command-line argument required");

    if ( (status = system(argv[1])) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

Program 8.14 Execute the command-line argument using `system`.

We'll compile this program into the executable file `tsys`.

Program 8.15 is another simple program that prints its real and effective user IDs.

```
#include    "ourhdr.h"

int
main(void)
{
    printf("real uid = %d, effective uid = %d\n", getuid(), geteuid());
    exit(0);
}
```

Program 8.15 Print real and effective user IDs.

We'll compile this program into the executable file `printuids`. Running both programs gives us the following.

```
$ tsys printuids                        normal execution, no special privileges
real uid = 224, effective uid = 224
normal termination, exit status = 0
$ su                                    become superuser
Password:                               enter superuser password
# chown root tsys                       change owner
# chmod u+s tsys                        make set-user-ID
# ls -l tsys                            verify file's permissions and owner
-rwsrwxr-x  1 root    105737 Aug 18 11:21 tsys
# exit                                  leave superuser shell
$ tsys printuids
real uid = 224, effective uid = 0       oops, this is a security hole
normal termination, exit status = 0
```

The superuser permissions that we gave the `tsys` program are retained across the `fork` and `exec` that are done by `system`.

If a process is running with special permissions (either set-user-ID or set-group-ID) and it wants to spawn another process, it should use `fork` and `exec` directly, being certain to change back to normal permissions after the `fork`, before calling `exec`. The `system` function should *never* be used from a set-user-ID or a set-group-ID program.

> One reason for this admonition is that `system` invokes the shell to parse the command string and the shell uses its `IFS` variable as the input field separator. Older versions of the shell didn't reset this variable to a normal set of characters when invoked. This allowed a malicious user to set `IFS` before `system` was called, causing `system` to execute a different program.

8.13 Process Accounting

Most Unix systems provide an option to do process accounting. When enabled the kernel writes an accounting record each time a process terminates. These accounting records are typically 32 bytes of binary data with the name of the command, the amount of CPU time used, the user ID and group ID, the starting time, and so on. We'll take a closer look at these accounting records in this section, since it gives us a chance to look at processes again, and a chance to use the `fread` function from Section 5.9.

> Process accounting is not specified by any of the standards. What we describe in this section corresponds to the implementation under SVR4 and 4.3+BSD. SVR4 provides numerous programs to process this raw accounting data—see `runacct` and `acctcom`, for example. 4.3+BSD provides the sa(8) command to process and summarize the raw accounting data.

A function we haven't described (`acct`) enables and disables process accounting. The only use of this function is from the SVR4 and 4.3+BSD `accton`(8) command. A superuser executes `accton` with a pathname argument to enable accounting. The pathname is usually `/var/adm/pacct`, although on older systems it is `/usr/adm/acct`. Accounting is turned off by executing `accton` without any arguments.

The structure of the accounting records is defined in the header `<sys/acct.h>` and looks like

```
typedef  u_short comp_t;  /* 3-bit base 8 exponent; 13-bit fraction */

struct  acct
{
  char    ac_flag;  /* flag (see Figure 8.9) */
  char    ac_stat;  /* termination status (signal & core flag only) */
                    /* (not provided by BSD systems) */
  uid_t   ac_uid;   /* real user ID */
  gid_t   ac_gid;   /* real group ID */
  dev_t   ac_tty;   /* controlling terminal */
  time_t  ac_btime; /* starting calendar time */
  comp_t  ac_utime; /* user CPU time (clock ticks) */
  comp_t  ac_stime; /* system CPU time (clock ticks) */
  comp_t  ac_etime; /* elapsed time (clock ticks) */
  comp_t  ac_mem;   /* average memory usage */
  comp_t  ac_io;    /* bytes transferred (by read and write) */
```

```
    comp_t  ac_rw;     /* blocks read or written */
    char    ac_comm[8];/* command name: [8] for SVR4, [10] for 4.3+BSD */
};
```

Historically, Berkeley systems, including 4.3+BSD, don't provide the `ac_stat` variable.

The `ac_flag` member records certain events during the execution of the process. These are described in Figure 8.9.

ac_flag	Description
AFORK	process is the result of `fork`, but never called exec
ASU	process used superuser privileges
ACOMPAT	process used compatibility mode (VAXes only)
ACORE	process dumped core (not in SVR4)
AXSIG	process was killed by a signal (not in SVR4)

Figure 8.9 Values for `ac_flag` from accounting record.

The data required for the accounting record (CPU times, number of characters transferred, etc.) are all kept by the kernel in the process table and initialized whenever a new process is created (e.g., in the child after a `fork`). Each accounting record is written when the process terminates. This means that the order of the records in the accounting file corresponds to the termination order of the processes, not the order in which they were started. To know the starting order we would have to go through the accounting file and sort by the starting calendar time. But this isn't perfect, since calendar times are in units of seconds (Section 1.10) and it's possible for many processes to be started in any given second. Alternatively, the elapsed time is given in clock ticks, which are usually between 50 and 100 ticks per second. But we don't know the ending time of a process, all we know is its starting time and ending order. This means that even though the elapsed time is more accurate than the starting time, we still can't reconstruct the exact starting order of various processes, given the data in the accounting file.

The accounting records correspond to processes, not programs. A new record is initialized by the kernel for the child after a `fork`, not when a new program is `exec`ed. Although `exec` doesn't create a new accounting record, the command name changes and the AFORK flag is cleared. This means that, if we have a chain of three programs (A `exec`s B, then B `exec`s C, and C `exit`s), only a single accounting record is written. The command name in the record corresponds to program C but the CPU times, for example, are the sum for programs A, B, and C.

Example

To have some accounting data to examine, we'll run Program 8.16, which calls `fork` four times. Each child does something different and then terminates. A picture of what this program is doing is shown in Figure 8.10.

Program 8.17 prints out selected fields from the accounting records.

```
#include    <signal.h>
#include    "ourhdr.h"

int
main(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {          /* parent */
        sleep(2);
        exit(2);                  /* terminate with exit status 2 */
    }

                                  /* first child */
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(4);
        abort();                  /* terminate with core dump */
    }

                                  /* second child */
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        execl("/usr/bin/dd", "dd", "if=/boot", "of=/dev/null", NULL);
        exit(7);                  /* shouldn't get here */
    }

                                  /* third child */
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(8);
        exit(0);                  /* normal exit */
    }

                                  /* fourth child */
    sleep(6);
    kill(getpid(), SIGKILL);      /* terminate with signal, no core dump */
    exit(6);                      /* shouldn't get here */
}
```

Program 8.16 Program to generate accounting data.

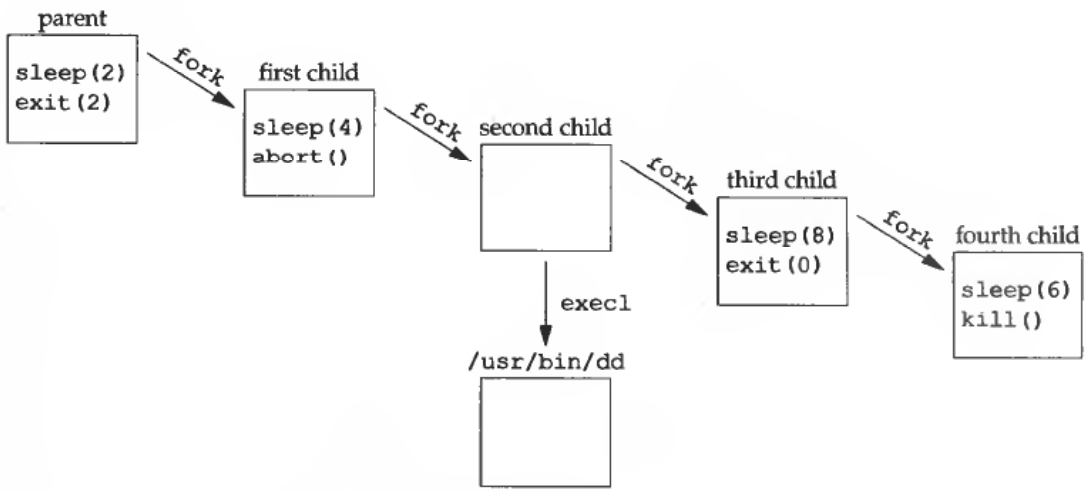


Figure 8.10 Process structure for accounting example.

We then do the following steps:

1. Become superuser and enable accounting, with the `accton` command. Note that, when this command terminates, accounting should be on, therefore the first record in the accounting file should be from this command.
2. Run Program 8.16. This should append five records to the accounting file (one for the parent and one for each of the four children).

   A new process is not created by the `execl` in the second child. There is only a single accounting record for the second child.
3. Become superuser and turn accounting off. Since accounting is off when this `accton` command terminates, it should not appear in the accounting file.
4. Run Program 8.17 to print the selected fields from the accounting file.

The output from step 4 follows. We have appended to each line the description of the process in italics, for the discussion later.

```
accton     e =      7, chars =      64, stat =   0:   S
dd         e =     37, chars =  221888, stat =   0:          second child
a.out      e =    128, chars =       0, stat =   0:          parent
a.out      e =    274, chars =       0, stat = 134: F   D X  first child
a.out      e =    360, chars =       0, stat =   9: F     X  fourth child
a.out      e =    484, chars =       0, stat =   0: F        third child
```

```
#include    <sys/types.h>
#include    <sys/acct.h>
#include    "ourhdr.h"

#define ACCTFILE        "/var/adm/pacct"
static unsigned long    compt2ulong(comp_t);

int
main(void)
{
    struct acct     acdata;
    FILE            *fp;

    if ( (fp = fopen(ACCTFILE, "r")) == NULL)
        err_sys("can't open %s", ACCTFILE);
    while (fread(&acdata, sizeof(acdata), 1, fp) == 1) {
        printf("%-*.*s  e = %6ld, chars = %7ld, "
               "stat = %3u: %c %c %c %c\n", sizeof(acdata.ac_comm),
                sizeof(acdata.ac_comm), acdata.ac_comm,
                compt2ulong(acdata.ac_etime), compt2ulong(acdata.ac_io),
                (unsigned char) acdata.ac_stat,
#ifdef  ACORE       /* SVR4 doesn't define ACORE */
                acdata.ac_flag & ACORE ? 'D' : ' ',
#else
                                                ' ',
#endif
#ifdef  AXSIG       /* SVR4 doesn't define AXSIG */
                acdata.ac_flag & AXSIG ? 'X' : ' ',
#else
                                                ' ',
#endif
                acdata.ac_flag & AFORK ? 'F' : ' ',
                acdata.ac_flag & ASU   ? 'S' : ' ');
    }
    if (ferror(fp))
        err_sys("read error");
    exit(0);
}

static unsigned long
compt2ulong(comp_t comptime)    /* convert comp_t to unsigned long */
{
    unsigned long   val;
    int             exp;

    val = comptime & 017777;    /* 13-bit fraction */
    exp = (comptime >> 13) & 7; /* 3-bit exponent (0-7) */
    while (exp-- > 0)
        val *= 8;
    return(val);
}
```

Program 8.17 Print selected fields from system's accounting file.

The elapsed time values are measured in units of CLK_TCK. From Figure 2.6 the value on this system is 60. For example, the `sleep(2)` in the parent corresponds to the elapsed time of 128 clock ticks. For the first child the `sleep(4)` becomes 274 clock ticks. Notice that the amount of time a process sleeps is not exact. (We'll return to the `sleep` function in Chapter 10.) Also, the calls to `fork` and `exit` take some amount of time.

Note that the `ac_stat` member is not the true termination status of the process. It corresponds to a portion of the termination status that we discussed in Section 8.6. The only information in this byte is a core-flag bit (usually the high-order bit) and the signal number (usually the seven low-order bits), if the process terminated abnormally. If the process terminated normally, we are not able to obtain the `exit` status from the accounting file. For the first child this value is 128+6. The 128 is the core flag bit and 6 happens to be the value on this system for SIGABRT (which is generated by the call to `abort`). The value 9 for the fourth child corresponds to the value of SIGKILL. We can't tell from the accounting data that the parent's argument to `exit` was 2, and the third child's argument to `exit` was 0.

The size of the file `/boot` that the `dd` process copies in the second child is 110,888 bytes. The number of characters of I/O is just over twice this value. It is twice the value since 110,888 bytes are read in, then 110,888 bytes are written out. Even though the output goes to the null device, they are still accounted for.

The `ac_flag` values are as we expect. The `F` flag is set for all the child processes except the second child that does the `execl`. The `F` flag is not set for the parent because the interactive shell that executed the parent did a `fork` and then an `exec` of the `a.out` file. The core dump flag (`D`) is on for the first child process that calls `abort`. Since `abort` generates a SIGABRT signal to generate the core dump, the `X` flag is also on for this process, since it was terminated by a signal. The `X` flag is also on for the fourth child, but the SIGKILL signal does not generate a core dump—it only terminates the process.

As a final note, the first child has a 0 count for the number of characters of I/O, yet this process generated a `core` file. It appears that the I/O required to write the `core` file is not charged to the process. □

8.14 User Identification

Any process can find out its real and effective user ID and group ID. Sometimes, however, we want to find out the login name of the user who's running the program. We could call `getpwuid(getuid())`, but what if a single user has multiple login names, each with the same user ID? (A person might have multiple entries in the password file with the same user ID to have a different login shell for each entry.) The system normally keeps track of the name we log in under (Section 6.7), and the `getlogin` function provides a way to fetch that login name.

```
#include <unistd.h>

char *getlogin(void);
```

Returns: pointer to string giving login name if OK, NULL on error

This function can fail if the process is not attached to a terminal that a user logged into. We normally call these processes *daemons*. We discuss them in Chapter 13.

Given the login name, we can then use it to look up the user in the password file (to determine the login shell, for example) using `getpwnam`.

> To find the login name, Unix systems have historically called the `ttyname` function (Section 11.9) and then tried to find a matching entry in the `utmp` file (Section 6.7). 4.3+BSD stores the login name in the process table entry and provides system calls to fetch and store this name.
>
> System V provided the `cuserid` function to return the login name. This function called `getlogin` and, if that failed, did a `getpwuid(getuid())`. The IEEE Std. 1003.1–1988 specified `cuserid`, but it called for the effective user ID to be used, instead of the real user ID. The final 1990 version of POSIX.1 dropped the `cuserid` function.
>
> FIPS 151–1 requires a login shell to define the environment variable LOGNAME with the user's login name. In 4.3+BSD this variable is set by `login` and inherited by the login shell. Realize, however, that a user can modify an environment variable, so we shouldn't use LOGNAME to validate the user in any way. Instead, `getlogin` should be used.

8.15 Process Times

In Section 1.10 we described three times that we can measure: wall clock time, user CPU time, and system CPU time. Any process can call the `times` function to obtain these values for itself and any terminated children.

```
#include <sys/times.h>

clock_t times(struct tms *buf);
```

Returns: elapsed wall clock time in clock ticks if OK, –1 on error

This function fills in the `tms` structure pointed to by *buf*.

```
struct tms {
  clock_t  tms_utime;  /* user CPU time */
  clock_t  tms_stime;  /* system CPU time */
  clock_t  tms_cutime; /* user CPU time, terminated children */
  clock_t  tms_cstime; /* system CPU time, terminated children */
};
```

Note that the structure does not contain any measurement for the wall clock time. Instead, the function returns the wall clock time as the value of the function, each time it's called. This value is measured from some arbitrary point in the past, so we can't use its absolute value, instead we use its relative value. For example, we call `times` and save the return value. At some later time we call `times` again, and subtract the earlier return value from the new return value. The difference is the wall clock time. (It is possible, though unlikely, for a long running process to overflow the wall clock time—see Exercise 1.6.)

The two structure fields for child processes contain values only for children that we have `wait`ed for.

All the `clock_t` values returned by this function are converted to seconds using the number of clock ticks per second—the `_SC_CLK_TCK` value returned by `sysconf` (Section 2.5.4).

> Berkeley-derived systems, including 4.3BSD, inherited a version of `times` from Version 7 that did not return the wall clock time. Instead, this older version returned 0 if OK or –1 on error. 4.3+BSD supports the POSIX.1 version.
>
> 4.3+BSD and SVR4 (in the BSD compatibility library) provide the `getrusage`(2) function. This function returns the CPU times, and 14 other values indicating resource usage.

Example

Program 8.18 executes each command-line argument as a shell command string, timing the command and printing the values from the `tms` structure. If we run this program we get:

```
$ a.out "sleep 5" "date"

command: sleep 5
  real:     5.25
  user:     0.00
  sys:      0.00
  child user:     0.02
  child sys:      0.13
normal termination, exit status = 0

command: date
Sun Aug 18 09:25:38 MST 1991
  real:     0.27
  user:     0.00
  sys:      0.00
  child user:     0.05
  child sys:      0.10
normal termination, exit status = 0
```

In these two examples all the CPU time appears in the child process, which is where the shell and the command execute.

```
#include    <sys/times.h>
#include    "ourhdr.h"

static void pr_times(clock_t, struct tms *, struct tms *);
static void do_cmd(char *);

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

    for (i = 1; i < argc; i++)
        do_cmd(argv[i]);    /* once for each command-line arg */
    exit(0);
}
static void
do_cmd(char *cmd)       /* execute and time the "cmd" */
{
    struct tms  tmsstart, tmsend;
    clock_t     start, end;
    int         status;

    fprintf(stderr, "\ncommand: %s\n", cmd);

    if ( (start = times(&tmsstart)) == -1)  /* starting values */
        err_sys("times error");

    if ( (status = system(cmd)) < 0)        /* execute command */
        err_sys("system() error");

    if ( (end = times(&tmsend)) == -1)      /* ending values */
        err_sys("times error");

    pr_times(end-start, &tmsstart, &tmsend);
    pr_exit(status);
}
static void
pr_times(clock_t real, struct tms *tmsstart, struct tms *tmsend)
{
    static long     clktck = 0;

    if (clktck == 0)    /* fetch clock ticks per second first time */
        if ( (clktck = sysconf(_SC_CLK_TCK)) < 0)
            err_sys("sysconf error");
    fprintf(stderr, "  real:  %7.2f\n", real / (double) clktck);
    fprintf(stderr, "  user:  %7.2f\n",
            (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
    fprintf(stderr, "  sys:   %7.2f\n",
            (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
    fprintf(stderr, "  child user:  %7.2f\n",
            (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
    fprintf(stderr, "  child sys:   %7.2f\n",
            (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}
```

Program 8.18 Time and execute all command-line arguments.

Let's rerun the example from Section 1.10.

```
$ a.out "cd /usr/include; grep _POSIX_SOURCE */*.h > /dev/null"

command: cd /usr/include; grep _POSIX_SOURCE */*.h > /dev/null
  real:    18.67
  user:     0.00
  sys:      0.02
  child user:     0.43
  child sys:      4.13
normal termination, exit status = 0
```

As we expect, all three values (the real time and the child CPU times) are similar to the values in Section 1.10. □

8.16 Summary

A thorough understanding of Unix process control is essential for advanced programming. There are only a few functions to master: `fork`, the `exec` family, `_exit`, `wait`, and `waitpid`. These primitives are used in many applications. The `fork` function also gave us an opportunity to look at race conditions.

Our examination of the `system` function and process accounting gave us another look at all these process control functions. We also looked at another variation of the `exec` functions: interpreter files and how they operate. An understanding of the different user IDs and group IDs that are provided (real, effective, and saved) is critical to writing safe set-user-ID programs.

Given an understanding of a single process and its children, in the next chapter we examine the relationship of a process to other processes—sessions and job control. We then complete our discussion of processes in Chapter 10 when we describe signals.

Exercises

8.1 In Program 8.2 we said that replacing the call to `_exit` with a call to `exit` causes the standard output to be closed. Modify the program to verify that `printf` does return –1.

8.2 Recall the typical arrangement of memory in Figure 7.3. Since the stack frames corresponding to each function call are usually stored in the stack, and since after a `vfork` the child runs in the address space of the parent, what happens if the call to `vfork` is from a function other than `main`, and the child does a return from this function after the `vfork`? Write a test program to verify this and draw a picture of what's happening.

8.3 When we execute Program 8.7 one time, as in

```
$ a.out
```

the output is correct. But if we execute the program multiple times, one right after the other, as in

```
$ a.out ; a.out ; a.out
output from parent
ooutput from parent
ouotuptut from child
put from parent
output from child
utput from child
```

the output is not correct. What's happening? How can we correct this? Can this problem happen if we let the child write its output first?

8.4 In Program 8.10 we call `execl`, specifying the *pathname* of the interpreter file. If we called `execlp` instead, specifying a *filename* of `testinterp`, and if the directory `/home/stevens/bin` was a path prefix, what would be printed as `argv[2]` when the program is run?

8.5 How can a process obtain its saved set-user-ID?

8.6 Write a program that creates a zombie and then call `system` to execute the `ps`(1) command to verify that the process is a zombie.

8.7 We mentioned in Section 8.9 that POSIX.1 requires that open directory streams be closed across an `exec`. Verify this as follows: call `opendir` for the root directory, peek at your system's implementation of the `DIR` structure, and print the close-on-exec flag. Then open the same directory for reading and print the close-on-exec flag.

9

Process Relationships

9.1 Introduction

We learned in the previous chapter that there are relationships between different processes. First, every process has a parent process. The parent is notified when the child terminates, and the parent can obtain the child's exit status. We also mentioned process groups when we described the `waitpid` function (Section 8.6) and how we can wait for any process in a process group to terminate.

In this chapter we'll look at process groups in more detail, and the new concept of sessions that was introduced by POSIX.1. We also look at the relationship between the login shell that is invoked for us when we log in and all the processes that we start from our login shell.

It is impossible to describe these relationships without talking about signals, and to talk about signals we need many of the concepts in this chapter. If you are unfamiliar with Unix signals you may want to skim through Chapter 10 at this point.

9.2 Terminal Logins

Let's start by looking at the programs that are executed when we log in to a Unix system. In early Unix systems, such as Version 7, users logged in using dumb terminals that were connected to the host with RS-232 connections. The terminals were either local (directly connected) or remote (connected through a modem). In either case, these logins came through a terminal device driver in the kernel. For example, on PDP-11s the common devices were DH-11s and DZ-11s. There were a fixed number of these terminal devices on a host, so there was a known upper limit on the number of simultaneous logins. The procedure that we now describe is used to log in to a Unix system using an RS-232 terminal.

4.3+BSD Terminal Logins

This procedure has not changed much over the past 15 years. The system administrator creates a file, usually `/etc/ttys`, that has one line per terminal device. Each line specifies the name of the device and other parameters that are passed to the `getty` program. These parameters specify the baud rate of the terminal, for example. When the system is bootstrapped the kernel creates process ID 1, the `init` process, and it is `init` that brings the system up multiuser. `init` reads the file `/etc/ttys` and, for every terminal device that allows a login, `init` does a `fork` followed by an `exec` of the program `getty`. This gives us the processes shown in Figure 9.1.

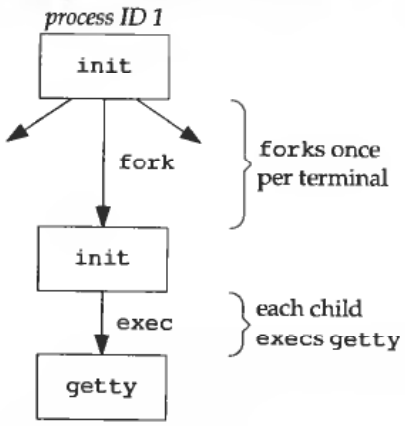


Figure 9.1 Processes invoked by `init` to allow terminal logins.

All the processes shown in Figure 9.1 have a real user ID of 0 and an effective user ID of 0 (i.e., they all have superuser privileges). `init` also execs the `getty` program with an empty environment.

It is `getty` that calls `open` for the terminal device. The terminal is opened for reading and writing. If the device is a modem, the open may delay inside the device driver until the modem is dialed and the call is answered. Once the device is open, file descriptors 0, 1, and 2 are set to the device. Then `getty` outputs something like `login:` and waits for us to enter our user name. If the terminal supports multiple speeds, `getty` can detect special characters that tell it to change the terminal's speed (baud rate). Consult your Unix manuals for additional details on the `getty` program and the data files (`gettytab`) that can drive its actions.

`getty` is done when we enter our user name. It then invokes the `login` program, similar to

```
execle("/usr/bin/login", "login", "-p", username, (char *) 0, envp);
```

(There can be options in the `gettytab` file to have it invoke other programs, but the default is the `login` program.) `init` invokes `getty` with an empty environment. `getty` creates an environment for `login` (the envp argument) with the name of the terminal (something like `TERM=foo`, where the type of terminal `foo` is taken from the `gettytab` file) and any environment strings that are specified in the `gettytab`. The

-p flag to `login` tells it to preserve the environment that it is passed and to add to that environment, not replace it. Figure 9.2 shows the state of these processes right after `login` has been invoked.

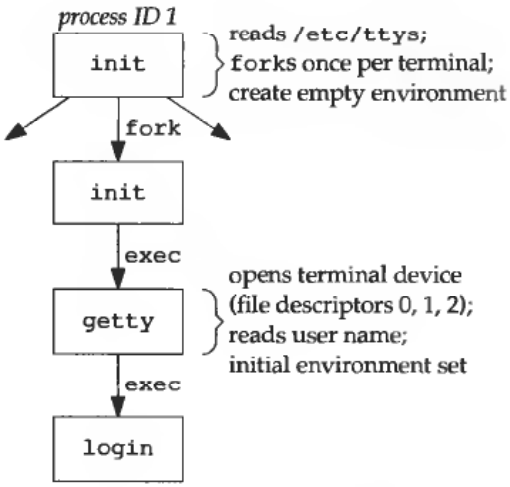


Figure 9.2 State of processes after `login` has been invoked.

All the processes shown in Figure 9.2 have superuser privileges, since the original `init` process has superuser privileges. The process ID of the bottom three processes in Figure 9.2 is the same, since the process ID does not change across an `exec`. Also, all the processes other than the original `init` process have a parent process ID of one.

Now `login` does many things. Since it has our user name it can call `getpwnam` to fetch our password file entry. It then calls `getpass`(3) to display the prompt `Password:` and read our password (with echoing disabled, of course). It calls `crypt`(3) to encrypt the password that we entered and compares the encrypted result with the `pw_passwd` field from our password file entry. If the login attempt fails because of an invalid password (after a few tries), `login` calls `exit` with an argument of 1. This termination will be noticed by the parent (`init`) and it will do another `fork` followed by an `exec` of `getty`, starting the procedure over again for this terminal.

If we log in correctly, `login` changes to our home directory (`chdir`). The ownership of our terminal device is changed (`chown`) so we are the owner and group owner. The access permissions are also changed for the terminal device, so that user-read, user-write, and group-read are enabled. Our group IDs are set, by calling `setgid` and then `initgroups`. The environment is then initialized with all the information that `login` has: our home directory (`HOME`), shell (`SHELL`), user name (`USER` and `LOGNAME`), and a default path (`PATH`). Finally it changes to our user ID (`setuid`) and invokes our login shell, as in

```
execl("/bin/sh", "-sh", (char *) 0);
```

The minus sign as the first character of `argv[0]` is a flag to all the shells that they are being invoked as a login shell. The shells can look at this character and modify their start-up accordingly.

`login` really does more than we've described here. It optionally prints the message-of-the-day file, checks for new mail, and does other functions. We're interested only in the features that we've described.

Recall from our discussion of the `setuid` function in Section 8.10 that since `setuid` is called by a superuser process it changes all three user IDs: the real user ID, effective user ID, and saved set-user-ID. The call to `setgid` that was done earlier by `login` has the same effect on all three group IDs.

At this point our login shell is running. Its parent process ID is the original `init` process (process ID 1), so when our login shell terminates, `init` is notified (it is sent a `SIGCHLD` signal) and it can start the whole procedure over again for this terminal. File descriptors 0, 1, and 2 for our login shell are set to the terminal device. Figure 9.3 shows this arrangement.

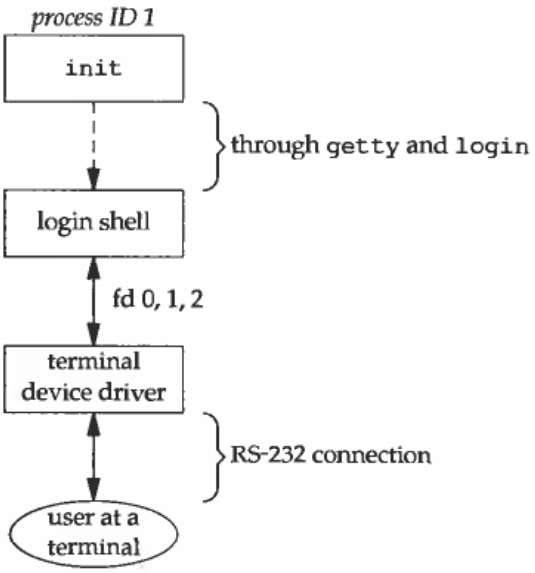


Figure 9.3 Arrangement of processes after everything is set for a terminal login.

Our login shell now reads its start-up files (`.profile` for the Bourne shell and KornShell, `.cshrc` and `.login` for the C shell). These start-up files usually change some of the environment variables and add many additional variables to the environment. For example, most users set their own `PATH` and often prompt for the actual terminal type (`TERM`). When the start-up files are done, we finally get the shell's prompt and can enter commands.

SVR4 Terminal Logins

SVR4 supports two forms of terminal logins: (a) `getty` style, as described previously for 4.3+BSD, and (b) `ttymon` logins, a new feature with SVR4. Normally `getty` is used for the console and `ttymon` is used for other terminal logins.

`ttymon` is part of a larger facility termed SAF, the Service Access Facility. For our purposes we end up with the same picture as in Figure 9.3, with a different set of steps

between `init` and the login shell. `init` is the parent of `sac` (the service access controller), which does a `fork` and `exec` of the `ttymon` program when the system enters multiuser state. `ttymon` monitors all the terminal ports listed in its configuration file and does a `fork` when we've entered our login name. This child of `ttymon` does an `exec` of `login`, and `login` prompts us for our password. Once this is done it `exec`s our login shell, and we're at the position shown in Figure 9.3. One difference is that the parent of our login shell is now `ttymon`, whereas the parent of the login shell from a `getty` login is `init`.

9.3 Network Logins

4.3+BSD Network Logins

With the terminal logins that we described in the previous section, `init` knows which terminal devices are enabled for logins and spawns a `getty` process for each device. In the case of network logins, however, all the logins come through the kernel's network interface drivers (e.g., the Ethernet driver), and we don't know ahead of time how many of these will occur. Instead of having a process waiting for each possible login, we now have to wait for a network connection request to arrive. In 4.3+BSD there is a single process that waits for most network connections, the `inetd` process, sometimes called the *Internet superserver*. In this section we'll look at the sequence of processes involved in network logins for a 4.3+BSD system. We are not interested in the detailed network programming aspects of these processes—refer to Stevens [1990] for all the details.

As part of the system start-up, `init` invokes a shell that executes the shell script `/etc/rc`. One of the daemons that is started by this shell script is `inetd`. Once the shell script terminates, the parent process of `inetd` becomes `init`. `inetd` waits for TCP/IP connection requests to arrive at the host, and when a connection request arrives for it to handle, it does a `fork` and `exec` of the appropriate program.

Let's assume that a TCP connection request arrives for the TELNET server. TELNET is a remote login application that uses the TCP protocol. A user on another host (that is connected to the server's host through a network of some form) or on the same host initiates the login by starting the TELNET client:

```
telnet hostname
```

The client opens a TCP connection to *hostname* and the program that's started on *hostname* is called the TELNET server. The client and server then exchange data across the TCP connection using the TELNET application protocol. What has happened is that the user who started the client program is now logged into the server's host. (This assumes, of course, that the user has a valid account on the server's host.) Figure 9.4 shows the sequence of processes that are involved in executing the TELNET server, called `telnetd`.

The `telnetd` process then opens a pseudo-terminal device and splits into two processes using `fork`. (In Chapter 19 we talk about pseudo terminals in detail.) The

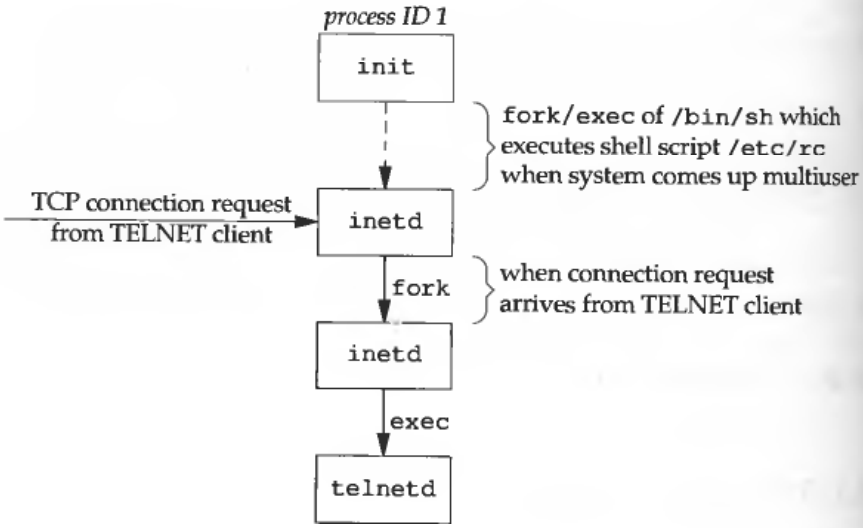


Figure 9.4 Sequence of processes involved in executing TELNET server.

parent handles the communication across the network connection, and the child does an `exec` of the `login` program. The parent and child are connected through the pseudo terminal. Before doing the `exec`, the child sets up file descriptors 0, 1, and 2 to the pseudo terminal. If we log in correctly, `login` performs the same steps we described in Section 9.2—it changes to our home directory, sets our group IDs and user ID, and our initial environment. Then `login` replaces itself with our login shell by calling `exec`. Figure 9.5 shows the arrangement of the processes at this point.

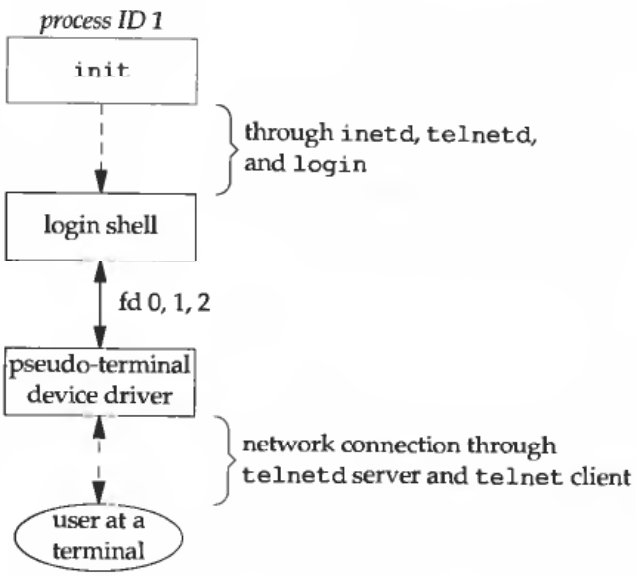


Figure 9.5 Arrangement of processes after everything is set for a network login.

Obviously a lot is going on between the pseudo-terminal device driver and the actual user at the terminal. We show all the processes involved in this type of arrangement in Chapter 19 when we talk about pseudo terminals in more detail.

The important thing to understand is whether we log in through a terminal (Figure 9.3) or a network (Figure 9.5) we have a login shell with its standard input, standard output, and standard error connected to either a terminal device or a pseudo-terminal device. We'll see in the coming sections that this login shell is the start of a POSIX.1 session, and the terminal or pseudo terminal is the controlling terminal for the session.

SVR4 Network Logins

The scenario for network logins under SVR4 is almost identical to the steps under 4.3+BSD. The same `inetd` server is used, but instead of its parent being `init`, under SVR4 `inetd` is invoked as a service by the service access controller, `sac`. We end up with the same overall picture as in Figure 9.5.

9.4 Process Groups

In addition to having a process ID, each process also belongs to a process group. We'll encounter process groups again when we discuss signals in Chapter 10.

A process group is a collection of one or more processes. Each process group has a unique process group ID. Process group IDs are similar to process IDs—they are positive integers and they can be stored in a `pid_t` data type. The function `getpgrp` returns the process group ID of the calling process.

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpgrp(void);
```

Returns: process group ID of calling process

In many Berkeley-derived systems, including 4.3+BSD, this function takes a *pid* argument and returns the process group for that process. The prototype shown is the POSIX.1 version.

Each process group can have a process group leader. The leader is identified by having its process group ID equal its process ID.

It is possible for a process group leader to create a process group, create processes in the group, and then terminate. The process group still exists, as long as there is at least one process in the group, regardless whether the group leader terminates or not. This is called the process group lifetime—the period of time that begins when the group is created and ends when the last remaining process in the group leaves the group. The last remaining process in the process group can either terminate or enter some other process group.

A process joins an existing process group, or creates a new process group by calling `setpgid`. (In the next section we'll see that `setsid` also creates a new process group.)

```
#include <sys/types.h>
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

Returns: 0 if OK, –1 on error

This sets the process group ID to *pgid* of the process *pid*. If the two arguments are equal, the process specified by *pid* becomes a process group leader.

A process can set the process group ID of only itself or one of its children. Furthermore, it can't change the process group ID of one of its children after that child has called one of the `exec` functions.

If *pid* is 0, the process ID of the caller is used. Also, if *pgid* is 0, the process ID specified by *pid* is used as the process group ID.

If the system does not support job control (we talk about job control in Section 9.8), `_POSIX_JOB_CONTROL` won't be defined, and this function returns an error with `errno` set to `ENOSYS`.

In most job-control shells this function is called after a `fork` to have the parent set the process group ID of the child, and to have the child set its own process group ID. One of these calls is redundant, but by doing both we are guaranteed that the child is placed into its own process group before either process assumes that is has happened. If we didn't do this we have a race condition, since it depends on which process executes first.

When we discuss signals we'll see how we can send a signal to either a single process (identified by its process ID) or to a process group (identified by its process group ID). Similarly, the `waitpid` function from Section 8.6 lets us wait for either a single process or one process from a specified process group.

9.5 Sessions

A session is a collection of one or more process groups. For example, we could have the arrangement shown in Figure 9.6. Here we have three process groups in a single session. The processes in a process group are usually grouped together into the process group by a shell pipeline. For example, the arrangement shown in Figure 9.6 could have been generated by shell commands of the form

```
proc1 | proc2 &
proc3 | proc4 | proc5
```

A process establishes a new session by calling the `setsid` function.

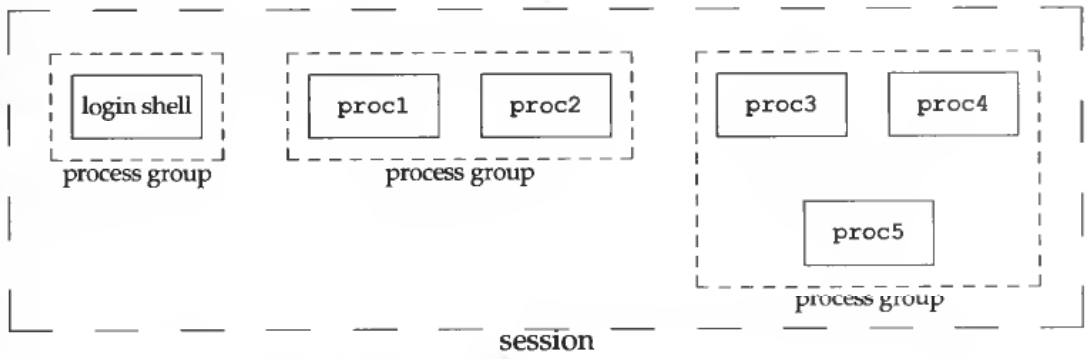


Figure 9.6 Arrangement of processes into process groups and sessions.

```
#include <sys/types.h>
#include <unistd.h>

pid_t setsid(void);
```

Returns: process group ID if OK, –1 on error

If the calling process is not a process group leader, this function creates a new session. Three things happen:

1. The process becomes the *session leader* of this new session. (A session leader is the process that creates a session.) The process is the only process in this new session.
2. The process becomes the process group leader of a new process group. The new process group ID is the process ID of the calling process.
3. The process has no controlling terminal. (We discuss controlling terminals in the next section.) If the process had a controlling terminal before calling `setsid`, that association is broken.

This function returns an error if the caller is already a process group leader. To ensure this is not the case, the usual practice is to call `fork` and have the parent terminate and the child continue. We are guaranteed that the child is not a process group leader because the process group ID of the parent is inherited by the child, but the child gets a new process ID. Hence it is impossible for the child's process ID to equal its inherited process group ID.

POSIX.1 talks only about a "session leader." There is no "session ID" similar to a process ID or a process group ID. Obviously a session leader is a single process that has a unique process ID, so we could talk about a session ID that is the process ID of the session leader. SVR4 does just this and both the SVID and the SVR4 manual page for `setsid`(2) talk about a session ID defined in this way. This is an implementation detail that is not part of POSIX.1 and is not supported by 4.3+BSD.

SVR4 has a `getsid` function that returns the session ID of a process. This function is not part of POSIX.1 and is not available under 4.3+BSD.

9.6 Controlling Terminal

There are a few other characteristics of sessions and process groups.

- A session can have a single *controlling terminal*. This is usually the terminal device (in the case of a terminal login) or pseudo-terminal device (in the case of a network login) on which we log in.
- The session leader that establishes the connection to the controlling terminal is called the *controlling process*.
- The process groups within a session can be divided into a single *foreground process group* and one or more *background process groups*.
- If a session has a controlling terminal, then it has a single foreground process group, and all other process groups in the session are background process groups.
- Whenever we type our terminal's interrupt key (often DELETE or Control-C) or quit key (often Control-backslash) this causes either the interrupt signal or the quit signal to be sent to all processes in the foreground process group.
- If a modem disconnect is detected by the terminal interface, the hang-up signal is sent to the controlling process (the session leader).

These characteristics are shown in Figure 9.7.

Usually we don't have to worry about our controlling terminal—it is established automatically for us when we log in.

How a system allocates a controlling terminal is left to the implementation by POSIX.1. We show the actual steps in Section 19.4.

SVR4 allocates the controlling terminal for a session when the session leader opens the first terminal device that is not already associated with a session. This assumes that the call to open by the session leader does not specify the `O_NOCTTY` flag (Section 3.3).

4.3+BSD allocates the controlling terminal for a session when the session leader calls `ioctl` with a *request* of `TIOCSCTTY` (the third argument is a null pointer). The session cannot already have a controlling terminal for this call to succeed. (Normally this call to `ioctl` follows a call to `setsid`, which guarantees that the process is a session leader without a controlling terminal.) The POSIX.1 `O_NOCTTY` flag to open is not used by 4.3+BSD.

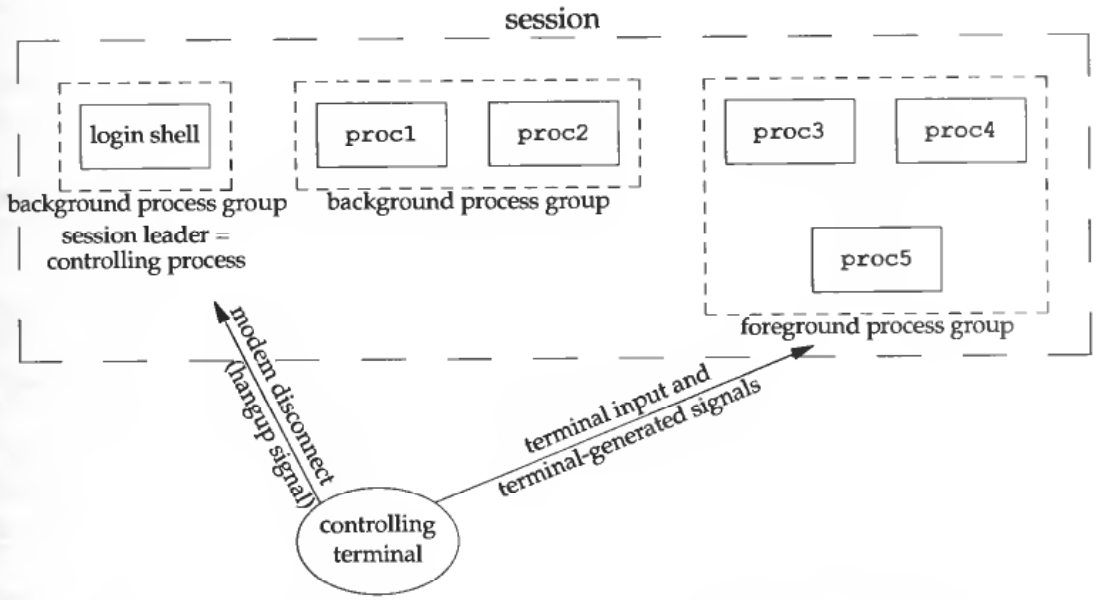


Figure 9.7 Process groups and sessions showing controlling terminal.

There are times when a program wants to talk to the controlling terminal, regardless whether the standard input or standard output is redirected. The way a program guarantees that it is talking to the controlling terminal is to open the file `/dev/tty`. This special file is a synonym within the kernel for the controlling terminal. Naturally, if the program doesn't have a controlling terminal, the open of this device will fail.

The classic example is the `getpass`(3) function that reads a password (with terminal echoing turned off, of course). This function is called by the `crypt`(1) program and can be used in a pipeline. For example

```
crypt < salaries | lpr
```

decrypts the file `salaries` and pipes the output to the print spooler. Because `crypt` reads its input file on its standard input, standard input can't be used to enter the password. Also, a design feature of `crypt` is that we should have to enter the encryption password each time we run the program, to prevent us from saving the password in a file.

There are known ways to break the encoding used by the `crypt` program. See Garfinkel and Spafford [1991] for more details on encrypting files.

9.7 `tcgetpgrp` and `tcsetpgrp` Functions

We need a way to tell the kernel which process group is the foreground process group, so that the terminal device driver knows where to send the terminal input and the terminal-generated signals (Figure 9.7).

```
#include <sys/types.h>
#include <unistd.h>

pid_t tcgetpgrp(int filedes);
```

Returns: process group ID of foreground process group if OK, –1 on error

```
int tcsetpgrp(int filedes, pid_t pgrpid);
```

Returns: 0 if OK, –1 on error

The function `tcgetpgrp` returns the process group ID of the foreground process group associated with the terminal open on *filedes*.

If the process has a controlling terminal, the process can call `tcsetpgrp` to set the foreground process group ID to *pgrpid*. The value of *pgrpid* must be the process group ID of a process group in the same session. *filedes* must refer to the controlling terminal of the session.

Most applications don't call these two functions directly. They are normally called by job-control shells. Both of these functions are defined only if `_POSIX_JOB_CONTROL` is defined. Otherwise they both return an error.

9.8 Job Control

Job control is a feature added by Berkeley around 1980. It allows us to start multiple jobs (groups of processes) from a single terminal and control which jobs can access the terminal and which jobs are to run in the background. Job control requires three forms of support.

1. A shell that supports job control.
2. The terminal driver in the kernel must support job control.
3. Support for certain job-control signals must be provided.

A different form of job control, termed shell layers, was provided by SVR3. The Berkeley form of job control, however, was selected by POSIX.1 and is what we describe here. Recall from Figure 2.7 that the constant `_POSIX_JOB_CONTROL` defines whether the system supports job control.

FIPS 151–1 requires POSIX.1 job control.

Both SVR4 and 4.3+BSD support POSIX.1 job control.

From our perspective, using job control from a shell, we can start a job in either the foreground of the background. A job is just a collection of processes, often a pipeline of processes. For example,

```
vi main.c
```

starts a job consisting of one process in the foreground. The commands

```
pr *.c | lpr &
make all &
```

start two jobs in the background. All the processes invoked by these background jobs are in the background.

As we said, we need to be using a shell that supports job control, to use the features provided by job control. With older systems it was simple to say which shells supported job control and which didn't. The C shell supported job control, the Bourne shell didn't, and it was an option with the KornShell, depending whether the host supported job control or not. But the C shell has been ported to systems that don't support job control (e.g., earlier versions of System V) and the SVR4 Bourne shell, when invoked by the name `jsh` instead of `sh`, supports job control. The KornShell continues to support job control if the host does. We'll just talk generically about a shell that supports job control, versus one that doesn't, when the difference between the various shells doesn't matter.

When we start a background job, the shell assigns it a job identifier and prints one or more of the process IDs. The following script shows how the KornShell handles this.

```
$ make all > Make.out &
[1]     1475
$ pr *.c | lpr &
[2]     1490
$                                   just press RETURN
[2] +  Done           pr *.c | lpr &
[1] +  Done           make all > Make.out &
```

The `make` is job number 1 and the starting process ID is 1475. The next pipeline is job number 2 and the process ID of the first process is 1490. When the jobs are done and when we press RETURN, the shell tells us that the jobs are complete. The reason we have to press RETURN is to have the shell print its prompt. The shell doesn't print the changed status of background jobs at any random time—only right before it prints its prompt, to let us enter a new command line. If it didn't do this, it could output while we were entering an input line.

The interaction with the terminal driver arises because there is a special terminal character that we can enter that affects the foreground job—the suspend key (typically Control-Z). Entering this character causes the terminal driver to send the `SIGTSTP` signal to all processes in the foreground process group. The jobs in any background process groups aren't affected. The terminal driver really looks for three special characters, which generate signals to the foreground process group:

- the interrupt character (typically DELETE or Control-C) generates `SIGINT`
- the quit character (typically Control-backslash) generates `SIGQUIT`
- the suspend character (typically Control-Z) generates `SIGTSTP`

In Chapter 11 we'll see how we can change these three characters to be any characters we choose, and how we can disable the terminal driver's processing of these special characters.

Another condition can arise with job control that must be handled by the terminal driver. Since we can have a foreground job and one or more background jobs, which of these receives the characters that we enter at the terminal? Only the foreground job receives terminal input. It is not an error for a background job to try to read from the terminal, but the terminal driver detects this and sends a special signal to the background job: `SIGTTIN`. This normally stops the background job, and by using the shell we are notified of this, and we can bring the job into the foreground so that it can read from the terminal. The following demonstrates this.

```
$ cat > temp.foo &              start in background, but it'll read from standard input
[1]     1681
$                               we press RETURN
[1] + Stopped (tty input)     cat > temp.foo &
$ fg %1                         bring job number 1 into the foreground
cat > temp.foo                  the shell tells us which job is now in the foreground
hello, world                    enter one line
^D                              type our end-of-file character
$ cat temp.foo                  check that the one line was put into the file
hello, world
```

The shell starts the `cat` process in the background, but when `cat` tries to read its standard input (the controlling terminal), the terminal driver, knowing that it is a background job, sends the `SIGTTIN` signal to the background job. The shell detects this change in status of its child (recall our discussion of the `wait` and `waitpid` function in Section 8.6) and tells us that the job has been stopped. We then move the stopped job into the foreground with the shell's `fg` command. (Refer to the manual page for the shell that you are using, for all the details on its job control commands, such as `fg` and `bg`, and the various ways to identify the different jobs.) Doing this causes the shell to place the job into the foreground process group (`tcsetpgrp`) and send the continue signal (`SIGCONT`) to the process group. Since the job is now in the foreground process group it can read from the controlling terminal.

What happens if a background job outputs to the controlling terminal? This is an option that we can allow or disallow. Normally we use the `stty`(1) command to change this option. (We'll see in Chapter 11 how we can change this option from a program.) The following shows how this works.

```
$ cat temp.foo &            execute in background
[1]     1719
$ hello, world              the output from the background job appears after the prompt
                            we press RETURN
[1] +  Done          cat temp.foo &
$ stty tostop               disable ability of background jobs to output to controlling terminal
$ cat temp.foo &            try it again in the background
[1]     1721
$                           we press RETURN and find the job is stopped
[1] + Stopped(tty output)     cat temp.foo &
$ fg %1                     resume stopped job in the foreground
cat temp.foo                the shell tells us which job is now in the foreground
hello, world                and here is its output
```

Figure 9.8 summarizes some of the features of job control that we've been describing.

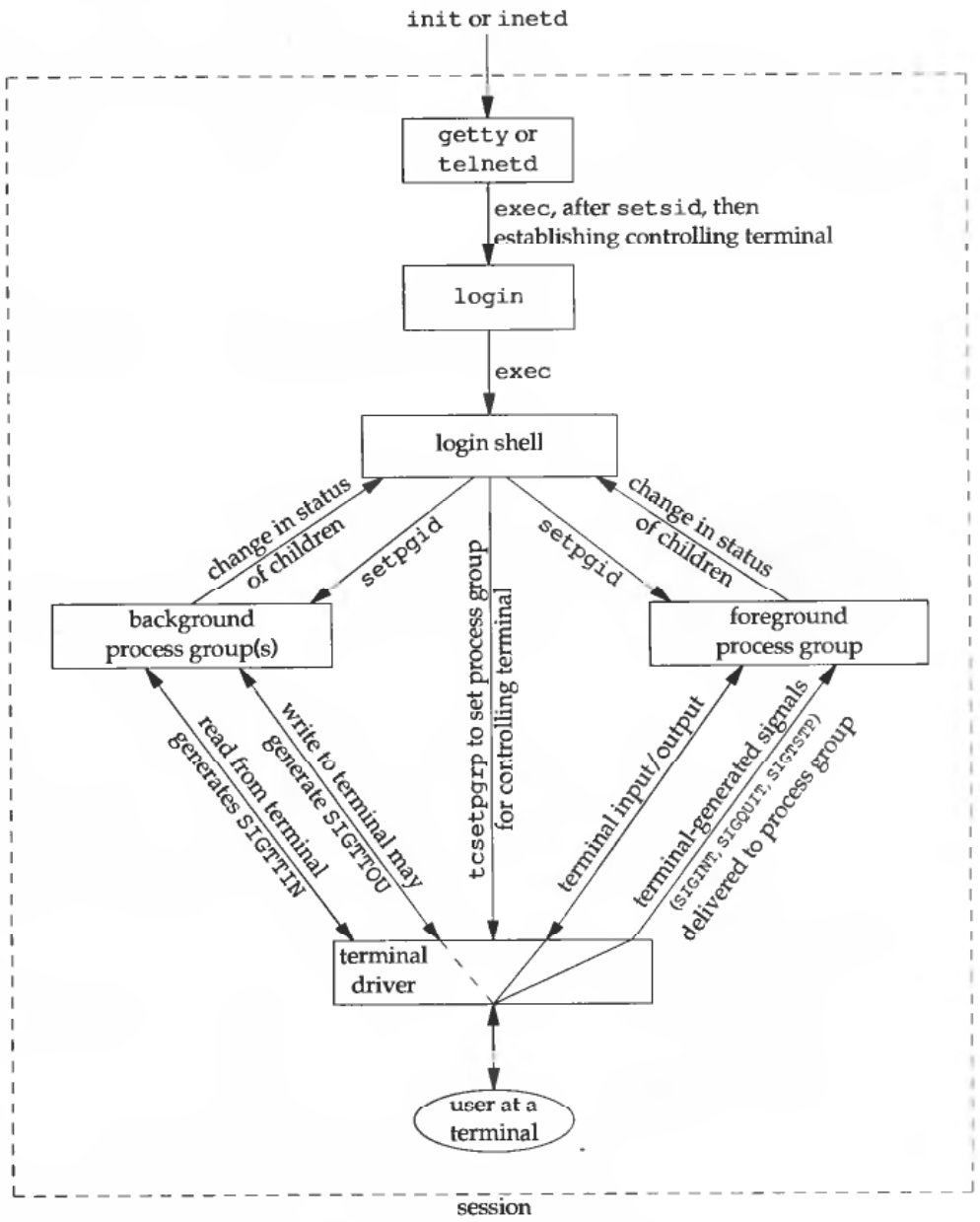


Figure 9.8 Summary of job control features with foreground and background jobs, and terminal driver.

The solid lines through the terminal driver box mean that the terminal I/O and the terminal-generated signals are always connected from the foreground process group to the actual terminal. The dashed line corresponding to the SIGTTOU signal means that whether the output from a process in the background process group appears on the terminal is an option.

Is job control necessary or desirable? This is a controversial topic that some users have strong opinions about. Job control was originally designed and implemented before windowing terminals were widespread. Some people claim that a well-designed windowing system removes any need for job control. Some complain that the implementation of job control—requiring support from the kernel, the terminal driver, the shell, and some applications—is a hack. Some use job control with a windowing system, claiming a need for both. Regardless of your opinion, job control is part of POSIX.1 and FIPS 151–1 and will be around for a while.

9.9 Shell Execution of Programs

Let's examine how the shells execute programs and how this relates to the concepts of process groups, controlling terminals, and sessions. To do this we'll use the `ps` command again.

First we'll use a shell that doesn't support job control—the classic Bourne shell. If we execute

```
ps -xj
```

the output is

```
PPID   PID  PGID   SID TPGID   COMMAND
   1   163   163   163   163   -sh
 163   168   163   163   163   ps
```

(We have removed some columns that don't interest us—the terminal name, user ID, and CPU time, for example.) Both the shell and the `ps` command are in the same session and foreground process group (163). We say 163 is the foreground process group because that is the process group shown under the TPGID column. The parent of the `ps` command is the shell, which we would expect. Note that the login shell is invoked by `login` with a hyphen as its first character.

> Unfortunately, the output of the `ps`(1) command differs in almost every version of Unix. Under SVR4 similar fields are output by the command `ps -jl`, although SVR4 never prints the TPGID field. Under 4.3+BSD the command is `ps -xj -Otpgid`.

Note that it is a misnomer to associate a process with a terminal process group ID (the TPGID column). A process does not have a terminal process control group. A process belongs to a process group, and the process group belongs to a session. The session may or may not have a controlling terminal. If it does have a controlling terminal, then the terminal device knows the process group ID of the foreground process. This value can be set in the terminal driver with the `tcsetpgrp` function, as we show in Figure 9.8. The foreground process group ID is an attribute of the terminal, not the process. This value from the terminal device driver is what `ps` prints as the TPGID. If `ps` finds that the session doesn't have a controlling terminal, it prints –1.

If we execute the command in the background,

```
ps -xj &
```

the only value that changes is the process ID of the command.

```
PPID   PID  PGID   SID TPGID   COMMAND
   1   163   163   163   163   -sh
 163   169   163   163   163   ps
```

The background job is not put into its own process group, and the controlling terminal isn't taken away from the background job, because this shell doesn't know about job control.

Let's now look at how the Bourne shell handles a pipeline. When we execute

```
ps -xj | cat1
```

the output is

```
PPID   PID  PGID   SID TPGID   COMMAND
   1   163   163   163   163   -sh
 163   200   163   163   163   cat1
 200   201   163   163   163   ps
```

(The program `cat1` is just a copy of the standard `cat` program, with a different name. We have another copy of `cat` with the name `cat2`, which we'll use later in this section. When we have two copies of `cat` in a pipeline, the different names let us differentiate between the two programs.) Note that the last process in the pipeline is the child of the shell, and the first process in the pipeline is a child of the last process. It appears that the shell `fork`s a copy of itself and this copy then `fork`s to make each of the previous processes in the pipeline.

If we execute the pipeline in the background

```
ps -xj | cat1 &
```

only the process IDs change. Since the shell doesn't handle job control, the process group ID of the background processes remains 163, as does the terminal process group ID.

What happens in this case if a background process tries to read from its controlling terminal? For example if we execute

```
cat > temp.foo &
```

With job control this is handled by placing the background job into a background process group, which causes the signal `SIGTTIN` to be generated if the background job tries to read from the controlling terminal. The way this is handled without job control is that the shell automatically redirects the standard input of a background process to `/dev/null`, if the process doesn't redirect standard input itself. A read from `/dev/null` generates an end of file. This means that our background `cat` process immediately reads an end of file and terminates normally.

The previous paragraph adequately handles the case of a background process accessing the controlling terminal through its standard input, but what happens if a background process specifically opens `/dev/tty` and reads from the controlling terminal? The answer is "it depends," but it's probably not what we want. For example,

```
crypt < salaries | lpr &
```

is such a pipeline. We run it in the background, but the `crypt` program opens `/dev/tty`, changes the terminal characteristics (to disable echoing), reads from the device, and resets the terminal characteristics. When we execute this background pipeline, the prompt `Password:` from `crypt` is printed on the terminal, but what we enter (the encryption password) is read by the shell and the shell tries to execute a command of that name. The next line we enter to the shell is taken as the password, and the file is not encrypted correctly, sending junk to the printer. Here we have two processes trying to read from the same device at the same time, and the result depends on the system. Job control, as we described earlier, handles this multiplexing of a single terminal between multiple processes in a better fashion.

Returning to our Bourne shell example, executing three processes in the pipeline

```
ps -xj | cat1 | cat2
```

lets us examine the process control employed by this shell.

```
PPID   PID  PGID   SID TPGID   COMMAND
   1   163   163   163   163   -sh
 163   202   163   163   163   cat2
 202   203   163   163   163   ps
 202   204   163   163   163   cat1
```

Again, the last process in the pipeline is the child of the shell, and all previous processes in the pipeline are children of the last process. Figure 9.9 shows what is happening.

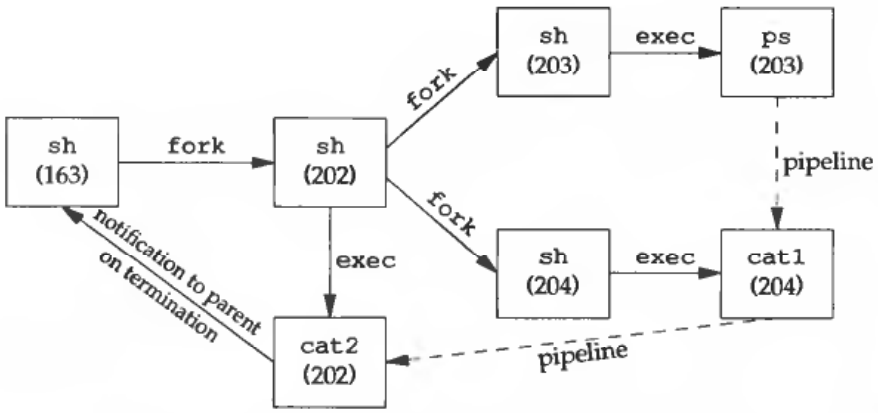


Figure 9.9 Processes in the pipeline "`ps -xj | cat1 | cat2`" when invoked by Bourne shell.

Since the last process in the pipeline is the child of the login shell, when that process (`cat2`) terminates, the shell is notified.

Now let's examine the same examples using a job-control shell. This shows the way these shells handle background jobs. We'll use the KornShell in this example—the C shell results are almost identical.

```
ps -xj
```

gives us

```
 PPID   PID  PGID   SID TPGID   COMMAND
    1   700   700   700   708   -ksh
  700   708   708   700   708   ps
```

(Starting with this example we show the foreground process group in a **bolder font**.) We immediately have a difference from our Bourne shell example. The KornShell places the foreground job (ps) into its own process group (708). The ps command is the process group leader and the only process in this process group. Furthermore this process group is the foreground process group since it has the controlling terminal. Our login shell is a background process group while the ps command executes. Note, however, that both process groups, 700 and 708, are members of the same session. Indeed, we'll see that the session never changes through our examples in this section.

Executing this process in the background

```
ps -xj &
```

gives us

```
 PPID   PID  PGID   SID TPGID   COMMAND
    1   700   700   700   700   -ksh
  700   709   709   700   700   ps
```

Again, the ps command is placed into its own process group but this time the process group (709) is no longer the foreground process group. It is a background process group. The TPGID of 700 indicates that the foreground process group is our login shell.

Executing two processes in a pipeline, as in

```
ps -xj | cat1
```

gives us

```
 PPID   PID  PGID   SID TPGID   COMMAND
    1   700   700   700   710   -ksh
  700   710   710   700   710   ps
  700   711   710   700   710   cat1
```

Both processes, ps and cat1, are placed into a new process group (710), and this is the foreground process group. We can also see another difference between this example and the similar Bourne shell example. The Bourne shell created the last process in the pipeline first, and this final process was the parent of the first process. Here the KornShell is the parent of both processes. But if we execute this pipeline in the background

```
ps -xj | cat1 &
```

the results show that now the KornShell generates the processes in the same fashion as the Bourne shell.

```
 PPID   PID  PGID   SID TPGID   COMMAND
    1   700   700   700   700   -ksh
  700   712   712   700   700   cat1
  712   713   712   700   700   ps
```

Both processes, 712 and 713, are placed into a background process group, 712.

9.10 Orphaned Process Groups

We've talked about the fact that a process whose parent terminates is called an orphan and is inherited by the `init` process. We now look at entire process groups that can be orphaned and how POSIX.1 handles this.

Example

Consider a process that `fork`s a child and then terminates. While this is nothing abnormal (it happens all the time), what happens if the child is stopped (using job control) when the parent terminates? How will the child ever be continued, and does the child know that it has been orphaned? Program 9.1 shows a specific example. There are some new features in this program that we describe below. Figure 9.10 shows the status after Program 9.1 has been started and the parent has `fork`ed the child.

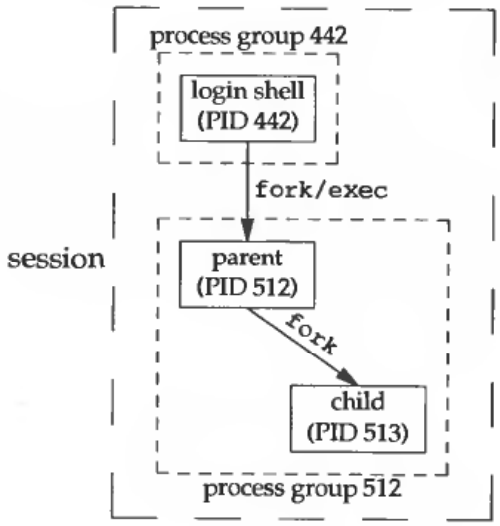


Figure 9.10 Example of a process group about to be orphaned.

Here we are assuming a job-control shell. Recall from the previous section that the shell places the foreground process into its own process group (512 in this example) and the shell stays in its own process group (442). The child inherits the process group of its parent (512). After the `fork`,

- The parent sleeps for 5 seconds. This is our (imperfect) way of letting the child execute before the parent terminates.
- The child establishes a signal handler for the hang-up signal (`SIGHUP`). This is so we can see if `SIGHUP` is sent to the child. (We discuss signal handlers in Chapter 10.)
- The child sends itself the stop signal (`SIGTSTP`) with the `kill` function. This stops the child, similar to our stopping a foreground job with our terminal's suspend character (Control-Z).

```
#include    <sys/types.h>
#include    <errno.h>
#include    <fcntl.h>
#include    <signal.h>
#include    "ourhdr.h"

static void sig_hup(int);
static void pr_ids(char *);

int
main(void)
{
    char    c;
    pid_t   pid;

    pr_ids("parent");
    if ( (pid = fork()) < 0)
        err_sys("fork error");

    else if (pid > 0) { /* parent */
        sleep(5);       /* sleep to let child stop itself */
        exit(0);        /* then parent exits */

    } else {            /* child */
        pr_ids("child");
        signal(SIGHUP, sig_hup);    /* establish signal handler */
        kill(getpid(), SIGTSTP);    /* stop ourself */
        pr_ids("child");    /* this prints only if we're continued */
        if (read(0, &c, 1) != 1)
            printf("read error from control terminal, errno = %d\n", errno);
        exit(0);
    }
}

static void
sig_hup(int signo)
{
    printf("SIGHUP received, pid = %d\n", getpid());
    return;
}

static void
pr_ids(char *name)
{
    printf("%s: pid = %d, ppid = %d, pgrp = %d\n",
            name, getpid(), getppid(), getpgrp());
    fflush(stdout);
}
```

Program 9.1 Creating an orphaned process group.

- When the parent terminates, the child is orphaned, so the child's parent process ID becomes 1, the `init` process ID.
- At this point the child is now a member of an *orphaned process group*. The POSIX.1 definition of an orphaned process group is one in which the parent of every member is either itself a member of the group or is not a member of the group's session. Another way of wording this is that the process group is not orphaned as long as there is a process in the group that has a parent in a different process group but in the same session. If the process group is not orphaned, there is a chance that one of those parents in a different process group but in the same session will restart a stopped process in the process group that is not orphaned.

  Here the parent of every process in the group (e.g., process 1 is the parent of process 513) belongs to another session.
- Since the process group is orphaned when the parent terminates, POSIX.1 requires that every process in the newly orphaned process group that is stopped (as our child is) be sent the hang-up signal (`SIGHUP`) followed by the continue signal (`SIGCONT`).
- This causes the child to be continued, after processing the hang-up signal. The default action for the hang-up signal is to terminate the process, which is why we have to provide a signal handler to catch the signal. We therefore expect the `printf` in the `sig_hup` function to appear before the `printf` in the `pr_ids` function.

Here is the output from Program 9.1.

```
$ a.out
parent: pid = 512, ppid = 442, pgrp = 512
child: pid = 513, ppid = 512, pgrp = 512
$ SIGHUP received, pid = 513
child: pid = 513, ppid = 1, pgrp = 512
read error from control terminal, errno = 5
```

Note that our shell prompt appears with the output from the child, since two processes, our login shell and the child, are writing to the terminal. As we expect, the parent process ID of the child has become 1.

Note that after calling `pr_ids` in the child, the program tries to read from standard input. As we saw earlier in this chapter, when a background process group tries to read from its controlling terminal, `SIGTTIN` is generated for the background process group. But here we have an orphaned process group—if the kernel were to stop it with this signal, the processes in the process group would probably never be continued. POSIX.1 specifies that the `read` is to return an error with `errno` set to `EIO` (whose value is 5 on this system).

Finally, notice that our child becomes a background process group when the parent terminates, since the parent was executed as a foreground job by the shell. □

We'll see another example of orphaned process groups in Section 19.5 with the `pty` program.

9.11 4.3+BSD Implementation

Having talked about the various attributes of a process, process group, session, and controlling terminal, it's worth looking at how all of this can be implemented. We'll look briefly at the implementation used by 4.3+BSD. Some details of the SVR4 implementation of these features can be found in Williams [1989]. Figure 9.11 shows the various data structures used by 4.3+BSD.

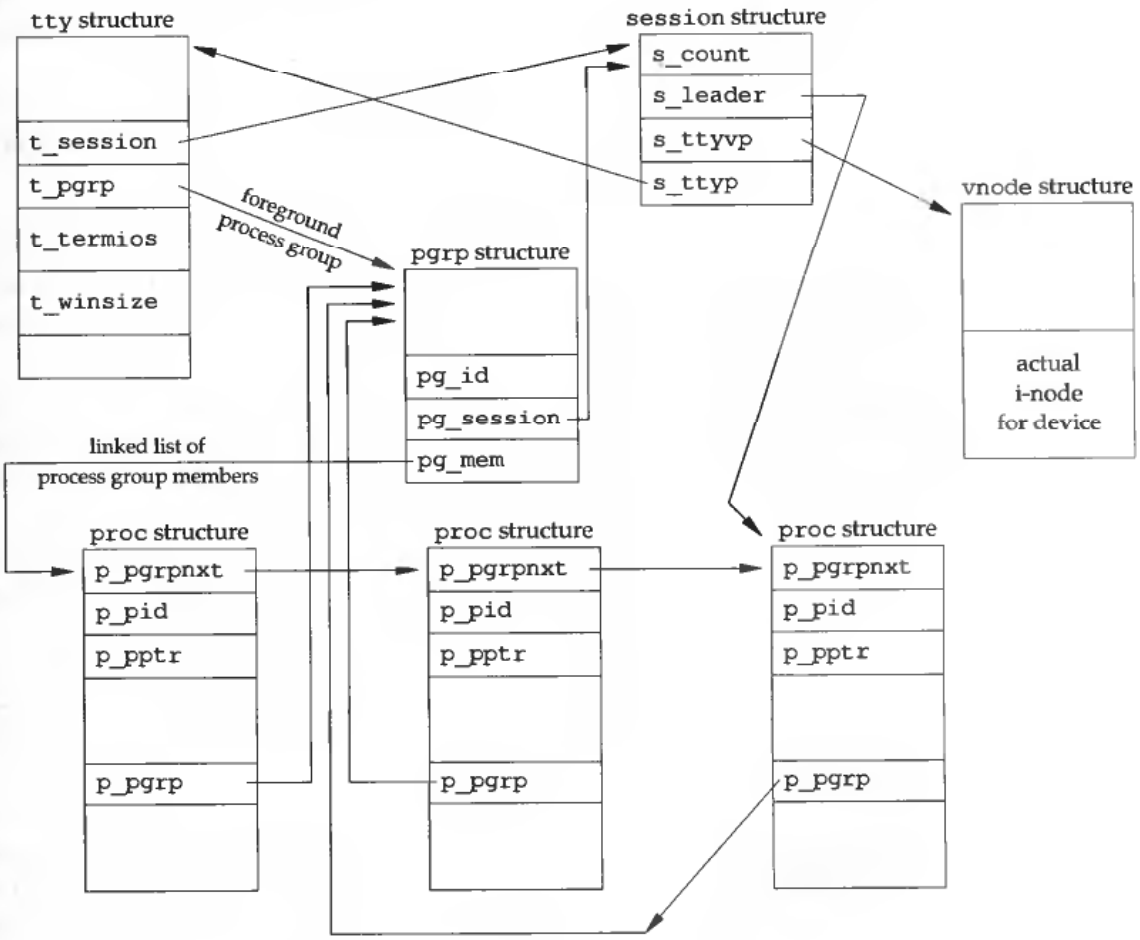


Figure 9.11 4.3+BSD implementation of sessions and process groups.

Let's look at all the fields that we've labeled. We'll start with the `session` structure. One of these structures is allocated for each session (e.g., each time `setsid` is called).

- `s_count` is the number of process groups in the session. When this counter is decremented to 0, the structure can be freed.

- `s_leader` is a pointer to the `proc` structure of the session leader. As we mentioned earlier, 4.3+BSD doesn't keep a session ID field, as SVR4 does.
- `s_ttyvp` is a pointer to the `vnode` structure of the controlling terminal.
- `s_ttyp` is a pointer to the `tty` structure of the controlling terminal.

When `setsid` is called, a new `session` structure is allocated within the kernel. `s_count` is set to 1, `s_leader` is set to point to the `proc` structure of the calling process, and `s_ttyvp` and `s_ttyp` are set to null pointers, since the new session doesn't have a controlling terminal.

Let's move to the `tty` structure. There is one of these structures in the kernel for each terminal device and each pseudo-terminal device. (We talk more about pseudo terminals in Chapter 19.)

- `t_session` points to the `session` structure that has this terminal as its controlling terminal. (Note that the `tty` structure points to the `session` structure and vice versa.) This pointer is used by the terminal to send a hang-up signal to the session leader if the terminal loses carrier (Figure 9.7).
- `t_pgrp` points to the `pgrp` structure of the foreground process group. This field is used by the terminal driver to send signals to the foreground process group. The three signals generated by entering special characters (interrupt, quit, and suspend) are sent to the foreground process group.
- `t_termios` is a structure containing all the special characters and related information for this terminal (e.g., baud rate, is echo on or off, etc.). We'll return to this structure in Chapter 11.
- `t_winsize` is a `winsize` structure that contains the current size of the terminal window. When the size of the terminal window changes, the `SIGWINCH` signal is sent to the foreground process group. We show how to set and fetch the terminal's current window size in Section 11.12.

Note that to find the foreground process group of a particular session the kernel has to start with the session structure, follow `s_ttyp` to get to the controlling terminal's `tty` structure, and then follow `t_pgrp` to get to the foreground process group's `pgrp` structure.

The `pgrp` structure contains the information for a particular process group.

- `pg_id` is the process group ID.
- `pg_session` points to the `session` structure that this process group belongs to.
- `pg_mem` is a pointer to the `proc` structure of the first process that is a member of this process group. The `p_pgrpnxt` in that `proc` structure points to the next process in the group, and so on, until a null pointer is encountered in the `proc` structure of the last process in the group.

The proc structure contains all the information for a single process.

- p_pid contains the process ID.
- p_pptr is a pointer to the proc structure of the parent process.
- p_pgrp points to the pgrp structure of the process group that this process belongs to.
- p_pgrpnxt is a pointer to the next process in the process group, as we mentioned earlier.

Finally we have the vnode structure. This structure is allocated when the controlling terminal device is opened. All references to /dev/tty in a process go through this vnode structure. We show the actual i-node as being part of the v-node. In Section 3.10 we said that this is the implementation used by 4.3+BSD, while SVR4 stores the v-node in the i-node.

9.12 Summary

This chapter has described the relationships between groups of processes—sessions, which are made up of process groups. Job control is a feature supported by many Unix systems today, and we've described how it's implemented by a shell that supports job control. The controlling terminal for a process, /dev/tty, is also involved in these process relationships.

We've made numerous references to the signals that are used in all these process relationships. The next chapter continues the discussion of signals, looking at all the Unix signals in detail.

Exercises

9.1 Refer back to our discussion of the utmp and wtmp files in Section 6.7. Why are the logout records written by the 4.3+BSD init process? Is this handled the same way for a network login?

9.2 Write a small program that calls fork and has the child create a new session. Verify that the child becomes a process group leader, and that the child no longer has a controlling terminal.

10

Signals

10.1 Introduction

Signals are software interrupts. Most nontrivial application programs need to deal with signals. Signals provide a way of handling asynchronous events: a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely.

Signals have been provided since the early versions of Unix, but the signal model provided with systems such as Version 7 was not reliable. Signals could get lost and it was hard for a process to turn off selected signals when executing critical regions of code. Both 4.3BSD and SVR3 made changes to the signal model, adding what are called *reliable signals*. But the changes made by Berkeley and AT&T were incompatible. Fortunately POSIX.1 standardizes the reliable signal routines, and that is what we describe in this chapter.

In this chapter we start with an overview of signals and a description of what each signal is normally used for. Then we look at the problems with earlier implementations. It is often important to understand what is wrong with an implementation, before seeing how to do things correctly. This chapter contains numerous examples that are not 100% correct and a discussion of the defects.

10.2 Signal Concepts

First, every signal has a name. These names all begin with the three characters `SIG`. For example, `SIGABRT` is the abort signal that is generated when a process calls the `abort` function. `SIGALRM` is the alarm signal that is generated when the timer set by the `alarm` function goes off. Version 7 had 15 different signals; SVR4 and 4.3+BSD both have 31 different signals.

These names are all defined by positive integer constants (the signal number) in the header <signal.h>. No signal has a signal number of 0. We'll see in Section 10.9 that the kill function uses the signal number of 0 for a special case. POSIX.1 calls this value the null signal.

Numerous conditions can generate a signal.

- The terminal-generated signals occur when users press certain terminal keys. Pressing the DELETE key on the terminal normally causes the interrupt signal to be generated (SIGINT). This is how to stop a runaway program. (We'll see in Chapter 11 how this signal can be mapped to any character on the terminal.)
- Hardware exceptions generate signals: divide by 0, invalid memory reference, and the like. These conditions are usually detected by the hardware, and the kernel is notified. The kernel then generates the appropriate signal for the process that was running at the time the condition occurred. For example, SIGSEGV is generated for a process that executes an invalid memory reference.
- The kill(2) function allows a process to send any signal to another process or process group. Naturally there are limitations: we have to be the owner of the process that we're sending the signal to, or we have to be the superuser.
- The kill(1) command allows us to send signals to other processes. This program is just an interface to the kill function. This command is often used to terminate a runaway background process.
- Software conditions can generate signals when something happens that the process should be made aware of. These aren't hardware-generated conditions (as is the divide-by-0 condition) but software conditions. Examples are SIGURG (generated when out-of-band data arrives over a network connection), SIGPIPE (generated when a process writes to a pipe after the reader of the pipe has terminated), and SIGALRM (generated when an alarm clock set by the process expires).

Signals are classic examples of asynchronous events. They occur at what appear to be random times to the process. The process can't just test a variable (such as errno) to see if a signal has occurred, instead the process has to tell the kernel "if and when this signal occurs, do the following."

There are three different things that we can tell the kernel to do when a signal occurs. We call this the *disposition* of the signal or the *action* associated with a signal.

1. Ignore the signal. This works for most signals, but there are two signals that can never be ignored: SIGKILL and SIGSTOP. The reason these two signals can't be ignored is to provide the superuser with a surefire way of either killing or stopping any process. Also, if we ignore some of the signals that are generated by a hardware exception (such as illegal memory reference or divide-by-0) the behavior of the process is undefined.

2. Catch the signal. To do this we tell the kernel to call a function of ours whenever the signal occurs. In our function we can do whatever we want to handle the condition. If we're writing a command interpreter, for example, when the user generates the interrupt signal at the keyboard, we probably want to return to the main loop of the program, terminating whatever command we were executing for the user. If the `SIGCHLD` signal is caught, it means a child process has terminated, so the signal-catching function can call `waitpid` to fetch the process ID of the child and its termination status. As another example, if the process has created temporary files we may want to write a signal-catching function for `SIGTERM` signal (the termination signal that is the default signal sent by the `kill` command) to clean up the temporary files.
3. Let the default action apply. Every signal has a default action, shown in Figure 10.1. Notice that the default action for most signals is to terminate the process.

Figure 10.1 lists the names of all the signals and an indication of which systems support the signal and the default action for the signal. The POSIX.1 column contains • if the signal is required, or "job" if the signal is a job-control signal (which is required only if job control is supported).

When the default action is labeled "terminate w/core" it means that a memory image of the process is left in the file named `core` of the current working directory of the process. (The fact that the file is named `core` shows how long this feature has been part of Unix.) This file can be used with most Unix debuggers to examine the state of the process at the time it terminated. The file will not be generated if (a) the process was set-user-ID and the current user is not the owner of the program file, or (b) the process was set-group-ID and the current user is not the group owner of the file, (c) the user does not have permission to write in the current working directory, or (d) the file is too big (recall the `RLIMIT_CORE` limit in Section 7.11). The permissions of the `core` file (assuming the file doesn't already exist) are usually user-read, user-write, group-read, and other-read.

> The generation of the `core` file is an implementation feature of most versions of Unix. It is not part of POSIX.1.
>
> Unix Version 6 didn't check for conditions (a) and (b) and the source code contained the comment "If you are looking for protection glitches, there are probably a wealth of them here when this occurs to a set-user-ID command."
>
> 4.3+BSD now generates a file with the name `core.`*prog*, where *prog* is the first 16 characters of the program name that was executed. This is a nice feature as it gives some identity to the core file.

The signals in Figure 10.1 with a description "hardware fault" correspond to implementation-defined hardware faults. Many of these names are taken from the original PDP-11 implementation of Unix. Check your system's manuals to determine exactly what type of error these signals correspond to.

Name	Description	ANSI C	POSIX.1	SVR4	4.3+BSD	Default action
SIGABRT	abnormal termination (abort)	•	•	•	•	terminate w/core
SIGALRM	time out (alarm)		•	•	•	terminate
SIGBUS	hardware fault			•	•	terminate w/core
SIGCHLD	change in status of child		job	•	•	ignore
SIGCONT	continue stopped process		job	•	•	continue/ignore
SIGEMT	hardware fault			•	•	terminate w/core
SIGFPE	arithmetic exception	•	•	•	•	terminate w/core
SIGHUP	hangup		•	•	•	terminate
SIGILL	illegal hardware instruction	•	•	•	•	terminate w/core
SIGINFO	status request from keyboard				•	ignore
SIGINT	terminal interrupt character	•	•	•	•	terminate
SIGIO	asynchronous I/O			•	•	terminate/ignore
SIGIOT	hardware fault			•	•	terminate w/core
SIGKILL	termination		•	•	•	terminate
SIGPIPE	write to pipe with no readers		•	•	•	terminate
SIGPOLL	pollable event (poll)			•		terminate
SIGPROF	profiling time alarm (setitimer)			•	•	terminate
SIGPWR	power fail/restart			•		ignore
SIGQUIT	terminal quit character		•	•	•	terminate w/core
SIGSEGV	invalid memory reference	•	•	•	•	terminate w/core
SIGSTOP	stop		job	•	•	stop process
SIGSYS	invalid system call			•	•	terminate w/core
SIGTERM	termination	•	•	•	•	terminate
SIGTRAP	hardware fault			•	•	terminate w/core
SIGTSTP	terminal stop character		job	•	•	stop process
SIGTTIN	background read from control tty		job	•	•	stop process
SIGTTOU	background write to control tty		job	•	•	stop process
SIGURG	urgent condition			•	•	ignore
SIGUSR1	user-defined signal		•	•	•	terminate
SIGUSR2	user-defined signal		•	•	•	terminate
SIGVTALRM	virtual time alarm (setitimer)			•	•	terminate
SIGWINCH	terminal window size change			•	•	ignore
SIGXCPU	CPU limit exceeded (setrlimit)			•	•	terminate w/core
SIGXFSZ	file size limit exceeded (setrlimit)			•	•	terminate w/core

Figure 10.1 Unix signals.

We'll now describe each of these signals in more detail.

SIGABRT This signal is generated by calling the abort function (Section 10.17). The process terminates abnormally.

SIGALRM This signal is generated when a timer that we've set with the alarm function expires. See Section 10.10 for more details.

This signal is also generated when an interval timer set by the setitimer(2) function expires.

SIGBUS This indicates an implementation-defined hardware fault.

SIGCHLD Whenever a process terminates or stops, the SIGCHLD signal is sent to the parent. By default this signal is ignored, so the parent must catch this signal if it wants to be notified whenever a child's status changes. The normal action in the signal-catching function is to call one of the wait functions to fetch the child's process ID and termination status.

Earlier releases of System V had a similar signal named SIGCLD (without the H). This signal had nonstandard semantics, and as far back as SVR2 the manual page warned that its use in new programs was strongly discouraged. Applications should use the standard SIGCHLD signal. We discuss these two signals in Section 10.7.

SIGCONT This job-control signal is sent to a stopped process when it is continued. If the process was stopped the default action is to continue the process, otherwise the default action is to ignore the signal. The vi editor, for example, catches this signal and redraws the terminal screen. Refer to Section 10.20 for additional details.

SIGEMT This indicates an implementation-defined hardware fault.

The name EMT comes from the PDP-11 "emulator trap" instruction.

SIGFPE This signals an arithmetic exception, such as divide-by-0, floating point overflow, and so on.

SIGHUP This signal is sent to the controlling process (session leader) associated with a controlling terminal if a disconnect is detected by the terminal interface. Referring to Figure 9.11 the signal is sent to the process pointed to by the s_leader field in the session structure. This signal is generated for this condition only if the terminal's CLOCAL flag is not set. (The CLOCAL flag for a terminal is set if the attached terminal is local. It tells the terminal driver to ignore all modem status lines. We describe how to set this flag in Chapter 11.) Note that the session leader that receives this signal may be in the background; see Figure 9.7 for an example. This differs from the normal terminal-generated signals (interrupt, quit, and suspend) that are always delivered to the foreground process group.

This signal is also generated if the session leader terminates. In this case the signal is sent to each process in the foreground process group.

This signal is commonly used to notify daemon processes (Chapter 13) to reread their configuration files. The reason SIGHUP is chosen for this is because a daemon should not have a controlling terminal and would normally never receive this signal.

SIGILL This signal indicates that the process has executed an illegal hardware instruction.

4.3BSD generated this signal from the abort function. SIGABRT is now used for this.

SIGINFO This 4.3+BSD signal is generated by the terminal driver when we type the status key (often Control-T). It is sent to all processes in the foreground process group (refer to Figure 9.8). This signal normally causes status information on processes in the foreground process group to be displayed on the terminal.

SIGINT This signal is generated by the terminal driver when we type the interrupt key (often DELETE or Control-C). It is sent to all processes in the foreground process group (refer to Figure 9.8). This signal is often used to terminate a runaway program, especially when it's generating a lot of output on the screen that we don't want.

SIGIO This signal indicates an asynchronous I/O event. We discuss it in Section 12.6.2.

> In Figure 10.1 we labeled the default action for SIGIO as either terminate or ignore. Unfortunately the default depends on the system. Under SVR4 SIGIO is identical to SIGPOLL, so its default action is to terminate the process. Under 4.3+BSD (the signal originated with 4.2BSD), the default is to be ignored.

SIGIOT This indicates an implementation-defined hardware fault.

> The name IOT comes from the PDP-11 mnemonic for the "input/output TRAP" instruction.
>
> Earlier versions of System V generated this signal from the abort function. SIGABRT is now used for this.

SIGKILL This signal is one of the two that can't be caught or ignored. It provides the system administrator a sure way to kill any process.

SIGPIPE If we write to a pipeline but the reader has terminated, SIGPIPE is generated. We describe pipes in Section 14.2. This signal is also generated when a process writes to a socket when the other end has terminated.

SIGPOLL This SVR4 signal can be generated when a specific event occurs on a pollable device. We describe this signal with the poll function in Section 12.5.2. It loosely corresponds to the 4.3+BSD SIGIO and SIGURG signals.

SIGPROF This signal is generated when a profiling interval timer set by the setitimer(2) function expires.

SIGPWR This SVR4 signal is system dependent. Its main use is on a system that has an uninterruptible power supply (UPS). If power fails, the UPS takes over and the software can usually be notified. Nothing needs to be done at this point, as the system continues running on battery power. But if the battery gets low (if the power is off for an extended period), the software is usually notified again, and at this point it behooves the system to shut everything down within about 15–30 seconds. This is when SIGPWR should be sent. Most systems have the process that is notified of the low battery condition send the SIGPWR signal to the init process, and init

handles the shutdown. Many System V implementations of `init` provide two entries in the `inittab` file for this purpose: `powerfail` and `powerwait`.

This signal is becoming more important with the availability of low cost UPS systems that can easily notify the computer of a low battery condition with an RS-232 serial connection.

`SIGQUIT` This signal is generated by the terminal driver when we type the terminal quit key (often Control-backslash). It is sent to all processes in the foreground process group (refer to Figure 9.8). This signal not only terminates the foreground process group (as does `SIGINT`), but it generates a `core` file.

`SIGSEGV` This signal indicates that the process has made an invalid memory reference.

The name SEGV stands for "segmentation violation."

`SIGSTOP` This job-control signal stops a process. It is like the interactive stop signal (`SIGTSTP`), but `SIGSTOP` cannot be caught or ignored.

`SIGSYS` This signals an invalid system call. Somehow the process executed a machine instruction that the kernel thought was a system call, but the parameter with the instruction that indicates the type of system call was invalid.

`SIGTERM` This is the termination signal sent by the `kill(1)` command by default.

`SIGTRAP` This indicates an implementation-defined hardware fault.

The signal name comes from the PDP-11 TRAP instruction.

`SIGTSTP` This interactive stop signal is generated by the terminal driver when we type the terminal suspend key (often Control-Z).† It is sent to all processes in the foreground process group (refer to Figure 9.8).

`SIGTTIN` This signal is generated by the terminal driver when a process in a background process group tries to read from its controlling terminal. (Refer to the discussion of this topic in Section 9.8.) As special cases, if either (a) the reading process is ignoring or blocking this signal or (b) the process group of the reading process is orphaned, then the signal is not generated and instead the read operation returns an error with `errno` set to `EIO`.

`SIGTTOU` This signal is generated by the terminal driver when a process in a background process group tries to write to its controlling terminal. (Refer to

† Unfortunately the term *stop* has different meanings. When discussing job control and signals we talk about stopping and continuing jobs. The terminal driver, however, has historically used the term stop to refer to stopping and starting the terminal output using the Control-S and Control-Q characters. Therefore the terminal driver calls the character that generates the interactive stop signal the suspend character, not the stop character.

the discussion of this topic in Section 9.8.) Unlike the SIGTTIN signal just described, a process has a choice of allowing background writes to the controlling terminal. We describe how to change this option in Chapter 11.

If background writes are not allowed, then like the SIGTTIN signal there are two special cases: if either (a) the writing process is ignoring or blocking this signal or (b) the process group of the writing process is orphaned, then the signal is not generated and instead the write operation returns an error with errno set to EIO.

Regardless whether background writes are allowed or not, certain terminal operations (other than writing) can also generate the SIGTTOU signal: tcsetattr, tcsendbreak, tcdrain, tcflush, tcflow, and tcsetpgrp. We describe these terminal operations in Chapter 11.

SIGURG This signal notifies the process that an urgent condition has occurred. This is optionally generated when out-of-band data is received on a network connection.

SIGUSR1 This is a user-defined signal, for use in application programs.

SIGUSR2 This is a user-defined signal, for use in application programs.

SIGVTALRM This signal is generated when a virtual interval timer set by the setitimer(2) function expires.

SIGWINCH The SVR4 and 4.3+BSD kernels maintain the size of the window associated with each terminal and pseudo terminal. A process can get and set the window size with the ioctl function, which we describe in Section 11.12. If a process changes the window size from its previous value, with the ioctl set-window-size command, the kernel generates the SIGWINCH signal for the foreground process group.

SIGXCPU SVR4 and 4.3+BSD support the concept of resource limits; refer to Section 7.11. If the process exceeds its soft CPU time limit, the SIGXCPU signal is generated.

SIGXFSZ This signal is generated by SVR4 and 4.3+BSD if the process exceeds its soft file size limit; refer to Section 7.11.

10.3 signal Function

The simplest interface to the signal features of Unix is the signal function.

```
#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);
```

Returns: previous disposition of signal (see following)

The signal function is defined by ANSI C. Since ANSI C doesn't involve multiple processes, process groups, terminal I/O, and the like, its definition of signals is vague enough to be almost useless for Unix systems. Indeed, the ANSI C description of signals takes 2 pages while the POSIX.1 description takes over 15 pages.

SVR4 also provides the signal function, but using it causes SVR4 to provide the old SVR2 unreliable signal semantics. (We describe these older semantics in Section 10.4.) This function is provided for backward compatibility for applications that require the older semantics. New applications should not use these unreliable signals.

4.3+BSD also provides the signal function, but it is defined in terms of the sigaction function (which we describe in Section 10.14), so using it under 4.3+BSD provides the newer reliable signal semantics.

When we describe the sigaction function we provide an implementation of signal that uses it. All the examples in this text use the signal function that we show in Program 10.12.

The *signo* argument is just the name of the signal from Figure 10.1. The value of *func* is either (a) the constant SIG_IGN, (b) the constant SIG_DFL, or (c) the address of a function to be called when the signal occurs. If we specify SIG_IGN we are telling the system to ignore the signal. (Remember that there are two signals, SIGKILL and SIGSTOP, that we cannot ignore.) By specifying SIG_DFL we are setting the action associated with the signal to its default action (see the final column in Figure 10.1). When we specify the address of a function to be called when the signal occurs, we call this "catching" the signal. We call the function either the *signal handler* or the *signal-catching function*.

The prototype for the signal function states that the function requires two arguments and returns a pointer to a function that returns nothing (void). The first argument, *signo*, is an integer. The second argument is a pointer to a function that takes a single integer argument and returns nothing. The function whose address is returned as the value of signal takes a single integer argument (the final (int)). In plain English, this declaration says that the signal handler is passed a single integer argument (the signal number) and it returns nothing. When we call signal to establish the signal handler, the second argument is a pointer to the function. The return value from signal is the pointer to the previous signal handler.

Many systems call the signal handler with additional, implementation-dependent arguments. We mention the optional SVR4 and 4.3+BSD arguments in Section 10.21.

The perplexing prototype shown at the beginning of this section for the signal function can be made much simpler through the use of the following typedef [Plauger 1992].

```
typedef void    Sigfunc(int);
```

Then the prototype becomes

```
Sigfunc *signal(int, Sigfunc *);
```

We've included this typedef in ourhdr.h (Appendix B) and use it with the functions in this chapter.

If we examine the system's header <signal.h> we probably find declarations of the form

```
#define SIG_ERR    (void (*)())-1
#define SIG_DFL    (void (*)())0
#define SIG_IGN    (void (*)())1
```

These constants can be used in place of the "pointer to a function that takes an integer argument and returns nothing," the second argument to signal, and the return value from signal. The three values used for these constants need not be –1, 0, and 1. They must be three values that can never be the address of any declarable function. Most Unix systems use the values shown.

Example

Program 10.1 shows a simple signal handler that catches either of the two user defined signals and prints the signal number. We describe the pause function in Section 10.10—it just puts the calling process to sleep.

```
#include    <signal.h>
#include    "ourhdr.h"

static void sig_usr(int);    /* one handler for both signals */

int
main(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR2");

    for ( ; ; )
        pause();
}

static void
sig_usr(int signo)      /* argument is signal number */
{
    if (signo == SIGUSR1)
        printf("received SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("received SIGUSR2\n");
    else
        err_dump("received signal %d\n", signo);
    return;
}
```

Program 10.1 Simple program to catch SIGUSR1 and SIGUSR2.

We invoke the program in the background and use the kill(1) command to send it signals. Note that the term *kill* in Unix is a misnomer. The kill(1) command and the kill(2) function just send a signal to a process or process group. Whether or not that signal terminates the process depends on which signal is sent and whether the process has arranged to catch the signal.

```
$ a.out &                          start process in background
[1]      4720                      job-control shell prints job number and process ID
$ kill -USR1 4720                  send it SIGUSR1
received SIGUSR1
$ kill -USR2 4720                  send it SIGUSR2
received SIGUSR2
$ kill 4720                        now send it SIGTERM
[1] + Terminated        a.out &
```

When we send the SIGTERM signal the process is terminated, since it doesn't catch the signal and the default action for the signal is termination. □

Program Start-up

When a program is execed the status of all signals is either default or ignore. Normally all signals are set to their default action, unless the process that calls exec is ignoring the signal. Specifically, the exec functions change the status of any signals that are being caught to their default action and leave the status of all other signals alone. (Naturally a signal that is being caught by a process that calls exec cannot be caught in the new program, since the address of the signal-catching function in the caller probably has no meaning in the new program file that is execed.)

One specific example that we encounter daily (but may not be cognizant of) is how an interactive shell treats the interrupt and quit signals for a background process. With a non-job-control shell, when we execute a process in the background, as in

```
cc main.c &
```

the shell automatically sets the disposition of the interrupt and quit signals in the background process to be ignored. This is so that if we type the interrupt character it doesn't affect the background process. If this weren't done, and we typed the interrupt character, not only would it terminate the foreground process, but it would also terminate all the background processes.

Many interactive programs that catch these two signals have code that looks like

```
int   sig_int(), sig_quit();

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, sig_int);
if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)
    signal(SIGQUIT, sig_quit);
```

Doing this, the process catches only the signal if the signal is not currently being ignored.

These two calls to `signal` also show a limitation of the `signal` function: we are not able to determine the current disposition of a signal without changing the disposition. We'll see later in this chapter how the `sigaction` function allows us to determine a signal's disposition without changing it.

Process Creation

When a process calls `fork` the child inherits the parent's signal dispositions. Here, since the child starts off with a copy of the parent's memory image, the address of a signal-catching function has meaning in the child.

10.4 Unreliable Signals

In earlier versions of Unix (such as Version 7), signals were unreliable. By this we mean that signals could get lost—a signal could occur and the process would never know about it. Also, a process had little control over a signal—it could catch the signal or ignore it. Sometimes we would like to tell the kernel to block a signal—don't ignore it, just remember if it occurs, and tell us later when we're ready.

> Changes were made with 4.2BSD to provide what are called *reliable signals*. A different set of changes was then made in SVR3 to provide reliable signals under System V. POSIX.1 chose the BSD model to standardize.

One problem with these early versions is that the action for a signal was reset to its default each time the signal occurred. (In the previous example, when we ran Program 10.1, we avoided this detail by catching each signal only once.) The classic example from many programming books that described these earlier systems concerns how to handle the interrupt signal. The code that was described usually looked like

```
int     sig_int();          /* my signal handling function */

...
signal(SIGINT, sig_int); /* establish handler */
...


sig_int()
{
    signal(SIGINT, sig_int);
                    /* reestablish handler for next occurrence */

    ...                         /* process the signal ... */
}
```

(The reason the signal handler is declared as returning an integer is that these early systems didn't support the ANSI C `void` data type.)

The problem with this code fragment is that there is a window of time after the signal has occurred, but before the call to `signal` in the signal handler, when the interrupt signal could occur another time. This second signal would cause the default action to

occur, which for this signal terminates the process. This is one of those conditions that works correctly most of the time, causing us to think that it is correct, when it isn't.

Another problem with these earlier systems is that the process was unable to turn a signal off when it didn't want the signal to occur. All the process could do was ignore the signal. There are times when we would like to tell the system "prevent the following signals from occurring, but remember if they do occur." The classic example that demonstrates this flaw is shown by a piece of code that catches a signal and sets a flag for the process that indicates that the signal occurred.

```
int   sig_int_flag;          /* set nonzero when signal occurs */

main()
{
    int    sig_int();        /* my signal handling function */
    ...
    signal(SIGINT, sig_int); /* establish handler */
    ...
    while (sig_int_flag == 0)
        pause();             /* go to sleep, waiting for signal */
    ...
}

sig_int()
{
    signal(SIGINT, sig_int); /* reestablish handler for next time */
    sig_int_flag = 1;        /* set flag for main loop to examine */
}
```

Here the process is calling the `pause` function to put it to sleep, until a signal is caught. When the signal is caught, the signal handler just sets the flag `sig_int_flag` nonzero. The process is automatically woken up by the kernel after the signal handler returns, notices the flag is nonzero, and does whatever it needs to do. But there is a window of time when things can go wrong. If the signal occurs after the test of `sig_int_flag`, but before the call to `pause`, the process could go to sleep forever (assuming the signal is never generated again). This occurrence of the signal is lost. This is another example of some code that isn't right, yet it works most of the time. Debugging this type of problem can be hard.

10.5 Interrupted System Calls

A characteristic of earlier Unix systems is that if a process caught a signal while the process was blocked in a "slow" system call, the system call was interrupted. The system call returned an error and `errno` was set to `EINTR`. This was done under the assumption that since a signal occurred and the process caught it, there is a good chance that something has happened that should wake up the blocked system call.

> Here we have to differentiate between a system call and a function. It is a system call within the kernel that is interrupted when some signal is caught.

To support this feature all the system calls are divided into two categories: the "slow" system calls, and all the others. The slow system calls are those that can block forever. Included in this category are

- reads from files that can block the caller forever if data isn't present (pipes, terminal devices, and network devices),
- writes to these same files that can block the caller forever if the data can't be accepted immediately,
- opens of files that block until some condition occurs (such as an open of a terminal device that waits until an attached modem answers the phone),
- `pause` (which by definition puts the calling process to sleep until a signal is caught) and `wait`,
- certain `ioctl` operations,
- some of the interprocess communication functions (Chapter 14).

The notable exception to these slow system calls is anything related to disk I/O. Although the read or a write of a disk file can block the caller temporarily (while the disk driver queues the request and then the request is executed), unless a hardware error occurs, the I/O operation always returns and unblocks the caller quickly.

One condition that is handled by interrupted system calls, for example, is when a process initiates a read from a terminal device and the user at the terminal walks away from the terminal for an extended period. In this example the process could be blocked for hours or days and would remain so unless the system was taken down.

The problem with interrupted system calls is that we now have to handle the error return explicitly. The typical code sequence (assuming a read operation and assuming we want to restart the read even if it's interrupted) would be

```
again:
    if ( (n = read(fd, buff, BUFFSIZE)) < 0) {
        if (errno == EINTR)
            goto again;     /* just an interrupted system call */
        /* handle other errors */
    }
```

To prevent the applications from having to handle interrupted system calls 4.2BSD introduced the automatic restarting of certain interrupted system calls. The system calls that were automatically restarted are `ioctl`, `read`, `readv`, `write`, `writev`, `wait`, and `waitpid`. As we've mentioned, the first five of these functions are interrupted by a signal only if they are operating on a slow device. `wait` and `waitpid` are always interrupted when a signal is caught. Since this caused a problem for some applications that didn't want the operation restarted if it was interrupted, 4.3BSD allowed the process to disable this feature on a per-signal basis.

POSIX.1 allows an implementation to restart system calls, but it is not required.

System V has never restarted system calls by default. But when `sigaction` is used with SVR4 (Section 10.14), the `SA_RESTART` option can be specified to restart system calls that are interrupted by that signal.

With 4.3+BSD the automatic restarting of system calls depends on which function is called to set the signal's disposition. The older, 4.3BSD-compatible `sigvec` function, causes system calls interrupted by that signal to be restarted automatically. But using the newer, POSIX.1-compatible `sigaction` causes them not to be restarted. As with SVR4, however, the `SA_RESTART` option can be used with `sigaction` to have the kernel restart system calls interrupted by that signal.

One of the reasons 4.2BSD introduced the automatic restart feature is because sometimes we don't know that the input or output device is a slow device. If the program we write can be used interactively, then it might be reading or writing a slow device, since terminals fall into this category. If we catch signals in this program, and if the system doesn't provide the restart capability, then we have to test every read or write for the interrupted error return and reissue the read or write.

Figure 10.2 summarizes the different signal functions and their semantics provided by the different implementations.

Functions	System	Signal handler remains installed	Ability to block signals	Automatic restart of interrupted system calls?
`signal`	V7, SVR2, SVR3, SVR4			never
`sigset, sighold, sigrelse sigignore, sigpause`	SVR3, SVR4	•	•	never
`signal, sigvec, sigblock`	4.2BSD	•	•	always
`sigsetmask, sigpause`	4.3BSD, 4.3+BSD	•	•	default
`sigaction, sigprocmask sigpending, sigsuspend`	POSIX.1	•	•	unspecified
	SVR4	•	•	optional
	4.3+BSD	•	•	optional

Figure 10.2 Features provided by different signal implementations.

Be aware that Unix systems from other vendors can have values different from those shown in this figure. For example, `sigaction` under SunOS 4.1.2 restarts an interrupted system call by default, different from both SVR4 and 4.3+BSD.

In Program 10.12 we provide our own version of the `signal` function that automatically tries to restart interrupted system calls (other than for the `SIGALRM` signal). In Program 10.13 we provide another function, `signal_intr`, that tries to never do the restart.

In all our code examples, we purposely show the `return` from a signal handler (if it returns) to remind ourselves that the return may interrupt a system call.

We talk more about interrupted system calls in Section 12.5 with regard to the `select` and `poll` functions.

10.6 Reentrant Functions

When a signal that is being caught is handled by a process, the normal sequence of instructions being executed by the process are temporarily interrupted by the signal handler. The process then continues executing, but the instructions in the signal handler are now executed. If the signal handler returns (instead of calling `exit` or `longjmp`, for example) then the normal sequence of instructions that the process was executing when the signal was caught continues executing. (This is similar to what happens when a hardware interrupt occurs.) But in the signal handler we can't tell where the process was executing when the signal was caught. What if the process was in the middle of allocating additional memory on its heap using `malloc`, and we call `malloc` from the signal handler? Or, what if the process was in the middle of a call to a function such as `getpwnam` (Section 6.2) that stores its result in a static location, and we call the same function from the signal handler? In the `malloc` example, havoc can result for the process, since `malloc` usually maintains a linked list of all its allocated areas, and it may have been in the middle of changing this list. In the case of `getpwnam` the information returned to the normal caller can get overwritten with the information returned to the signal handler.

POSIX.1 specifies the functions that are guaranteed to be reentrant. Figure 10.3 lists these reentrant functions. The four functions marked with an asterisk in this figure are not specified as being reentrant by POSIX.1, but are listed in the SVR4 SVID [AT&T 1989] as being reentrant.

_exit	fork	pipe	stat
abort*	fstat	read	sysconf
access	getegid	rename	tcdrain
alarm	geteuid	rmdir	tcflow
cfgetispeed	getgid	setgid	tcflush
cfgetospeed	getgroups	setpgid	tcgetattr
cfsetispeed	getpgrp	setsid	tcgetpgrp
cfsetospeed	getpid	setuid	tcsendbreak
chdir	getppid	sigaction	tcsetattr
chmod	getuid	sigaddset	tcsetpgrp
chown	kill	sigdelset	time
close	link	sigemptyset	times
creat	longjmp*	sigfillset	umask
dup	lseek	sigismember	uname
dup2	mkdir	signal*	unlink
execle	mkfifo	sigpending	utime
execve	open	sigprocmask	wait
exit*	pathconf	sigsuspend	waitpid
fcntl	pause	sleep	write

Figure 10.3 Reentrant functions that may be called from a signal handler.

Most functions that are not in Figure 10.3 are missing because (a) they are known to use static data structures, (b) they call `malloc` or `free`, or (c) they are part of the standard I/O library. Most implementations of the standard I/O library use global data structures in a nonreentrant way.

Be aware that even if we call a function listed in Figure 10.3 from a signal handler, there is only one `errno` variable per process, and we might modify its value. Consider a signal handler that is invoked right after `main` has set `errno`. If the signal handler calls `read`, for example, this call can change the value of `errno`, wiping out the value that was just stored in `main`. Therefore, as a general rule, when calling the functions listed in Figure 10.3 from a signal handler, we should save and restore `errno`. (Be aware that a commonly caught signal is `SIGCHLD` and its signal handler usually calls one of the `wait` functions. All the `wait` functions can change `errno`.)

POSIX.1 does not include `longjmp` and `siglongjmp` in Figure 10.3. (We describe the latter function in Section 10.15.) This is because the signal may have occurred while the main routine was updating a data structure in a nonreentrant way. Not returning from the signal handler, but calling `siglongjmp` instead, could leave this data structure half updated. If the application is going to do things such as update global data structures as we describe here, while catching signals that cause `sigsetjmp` to be executed, then we need to block the signal while we're updating the data structure.

Example

Program 10.2 calls the nonreentrant function `getpwnam` from a signal handler that is called every second. We describe the `alarm` function in Section 10.10. We use it here to generate a `SIGALRM` every second.

When this program was run the results were random. Usually the program would be terminated by a `SIGSEGV` signal when the signal handler returned the first time. An examination of the `core` file showed that the `main` function had called `getpwnam`, but some internal pointers had been corrupted when the signal handler called the same function. Occasionally the program would run for several seconds before crashing with a `SIGSEGV` error. When the `main` function did run correctly after the signal had been caught, sometimes the return value was corrupted and sometimes it was fine. Once the call to `getpwnam` from the signal handler returned an error of `EBADF` (invalid file descriptor).

As shown by this example, if we call a nonreentrant function from a signal handler, the results are unpredictable. □

10.7 SIGCLD Semantics

Two signals that continually generate confusion are `SIGCLD` and `SIGCHLD`. First, `SIGCLD` (without the `H`) is the System V name, and this signal has different semantics from the BSD signal, named `SIGCHLD`. The POSIX.1 signal is also named `SIGCHLD`.

The semantics of the BSD `SIGCHLD` signal are normal, in that its semantics are similar to all other signals. When the signal occurs, the status of a child has changed and we need to call one of the `wait` functions to determine what has happened.

System V, however, has traditionally handled the `SIGCLD` signal differently from other signals. SVR4 continues this questionable tradition (i.e., compatibility constraint), if we set its disposition using either `signal` or `sigset` (the older, SVR3-compatible

```
#include    <pwd.h>
#include    <signal.h>
#include    "ourhdr.h"

static void my_alarm(int);

int
main(void)
{
    struct passwd   *ptr;

    signal(SIGALRM, my_alarm);
    alarm(1);

    for ( ; ; ) {
        if ( (ptr = getpwnam("stevens")) == NULL)
            err_sys("getpwnam error");
        if (strcmp(ptr->pw_name, "stevens") != 0)
            printf("return value corrupted!, pw_name = %s\n",
                    ptr->pw_name);
    }
}

static void
my_alarm(int signo)
{
    struct passwd   *rootptr;

    printf("in signal handler\n");
    if ( (rootptr = getpwnam("root")) == NULL)
            err_sys("getpwnam(root) error");
    alarm(1);
    return;
}
```

Program 10.2 Call a nonreentrant function from a signal handler.

functions to set the disposition of a signal). This older handling of SIGCLD consists of the following:

1. If the process specifically sets its disposition to SIG_IGN, children of the calling process will not generate zombie processes. Note that this is different from its default action (SIG_DFL), which from Figure 10.1 is to be ignored. Instead, on termination the status of these child processes is just discarded. If the calling process subsequently calls one of the wait functions, it will block until all of its children have terminated, and then wait returns –1 with errno set to ECHILD. (The default disposition of this signal is to be ignored, but this default will not cause the above semantics to occur. Instead, we specifically have to set its disposition to SIG_IGN.)

POSIX.1 does not specify what happens when SIGCHLD is ignored, so this behavior is allowed.

4.3+BSD always generates zombies if SIGCHLD is ignored. If we want to avoid zombies, we have to wait for our children.

With SVR4, if either signal or sigset is called to set the disposition of SIGCHLD to be ignored, zombies are never generated. Also, with the SVR4 version of sigaction, we can set the SA_NOCLDWAIT flag (Figure 10.5) to avoid zombies.

2. If we set the disposition of SIGCLD to be caught, the kernel immediately checks if there are any child processes ready to be waited for and, if so, calls the SIGCLD handler.

Item 2 changes the way we have to write a signal handler for this signal.

Example

Recall in Section 10.4 we said the first thing to do on entry to a signal handler is to call signal again, to reestablish the handler. (This was to minimize the window of time when the signal is reset back to its default, and could get lost.) We show this in Program 10.3. This program doesn't work. If we compile and run it under SVR2 the output is a continual string of SIGCLD received lines. Eventually the process runs out of stack space and terminates abnormally.

The problem with this program is that the call to signal at the beginning of the signal handler invokes item 2 from the preceding discussion—the kernel checks if there is a child that needs to be waited for (which there is, since we're processing a SIGCLD signal), so it generates another call to the signal handler. The signal handler calls signal, and the whole process starts over again.

To fix this program we have to move the call to signal after the call to wait. By doing this we call signal after fetching the child's termination status—the signal is generated again by the kernel only if some other child has since terminated.

POSIX.1 states that when we establish a signal handler for SIGCHLD, and there exists a terminated child who we have not yet waited for, it is unspecified whether the signal is generated. This allows the behavior described previously. But since POSIX.1 does not reset a signal's disposition to its default when the signal occurs (assuming we're using the POSIX.1 sigaction function to set its disposition), there is no need for us to ever establish a signal handler for SIGCHLD within that handler.

□

Be cognizant of the semantics that your implementation associates with the SIGCHLD signal. Be especially aware of some systems that #define SIGCHLD to be SIGCLD or vice versa. Changing the name may allow you to compile a program that was written for another system, but if that program depends on the other semantics, it may not work.

```
#include    <sys/types.h>
#include    <signal.h>
#include    <stdio.h>

static void sig_cld();

int
main()
{
    pid_t   pid;

    if (signal(SIGCLD, sig_cld) == -1)
        perror("signal error");

    if ( (pid = fork()) < 0)
        perror("fork error");
    else if (pid == 0) {        /* child */
        sleep(2);
        _exit(0);
    }
    pause();    /* parent */
    exit(0);
}

static void
sig_cld()
{
    pid_t   pid;
    int     status;

    printf("SIGCLD received\n");
    if (signal(SIGCLD, sig_cld) == -1)  /* reestablish handler */
        perror("signal error");

    if ( (pid = wait(&status)) < 0)     /* fetch child status */
        perror("wait error");
    printf("pid = %d\n", pid);
    return;     /* interrupts pause() */
}
```

Program 10.3 System V `SIGCLD` handler that doesn't work.

10.8 Reliable Signal Terminology and Semantics

There are terms used throughout our discussion of signals that we need to define. First, a signal is *generated* for a process (or sent to a process) when the event that causes the signal occurs. The event could be a hardware exception (e.g., divide by 0), a software condition (e.g., an `alarm` timer expiring), a terminal-generated signal, or a call to the `kill` function. When the signal is generated the kernel usually sets a flag of some form in the process table.

We say that a signal is *delivered* to a process when the action for a signal is taken. During the time between the generation of a signal and its delivery, the signal is said to be *pending*.

A process has the option of *blocking* the delivery of a signal. If a signal that is blocked is generated for a process, and if the action for that signal is either the default action or to catch the signal, then the signal remains pending for the process until the process either (a) unblocks the signal or (b) changes the action to ignore the signal. The system determines what to do with a blocked signal when the signal is delivered, not when it's generated. This allows the process to change the action for the signal before it's delivered. The `sigpending` function (Section 10.13) can be called by a process to determine which signals are blocked and pending.

What happens if a blocked signal is generated more than once before the process unblocks the signal? POSIX.1 allows the system to deliver the signal either once or more than once. If the system delivers the signal more than once, we say that the signals are queued. Most Unix systems, however, do *not* queue signals. Instead the Unix kernel just delivers the signal once.

> The manual pages for earlier versions of System V claimed that the `SIGCLD` signal was queued, but it really wasn't. Instead the signal was regenerated by the kernel as we described in Section 10.7.
>
> The `sigaction`(2) manual page in AT&T [1990e] claims that the `SA_SIGINFO` flag (Figure 10.5) causes signals to be reliably queued. This is wrong. Apparently this feature exists within the kernel, but it is not enabled in SVR4.

What happens if more than one signal is ready to be delivered to a process? POSIX.1 does not specify the order in which the signals are delivered to the process. The Rationale for POSIX.1 does suggest, however, that signals related to the current state of the process, such as `SIGSEGV`, be delivered before other signals.

Each process has a *signal mask* that defines the set of signals currently blocked from delivery to that process. We can think of this mask as having one bit for each possible signal. If the bit is on for a given signal, that signal is currently blocked. A process can examine and change its current signal mask by calling `sigprocmask`, which we describe in Section 10.12.

Since it is possible for the number of signals to exceed the number of bits in an integer, POSIX.1 defines a new data type, `sigset_t` that holds a *signal set*. The signal mask, for example, is stored in one of these signal sets. We describe five functions that operate on signal sets in Section 10.11.

10.9 `kill` and `raise` Functions

The `kill` function sends a signal to a process or a group of processes. The `raise` function allows a process to send a signal to itself.

> `raise` is defined by ANSI C, not POSIX.1. Since ANSI C does not deal with multiple processes it could not define a function such as `kill` that requires a process ID argument.

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int signo);

int raise(int signo);
```

Both return: 0 if OK, −1 on error

There are four different conditions for the *pid* argument to `kill`.

pid > 0 The signal is sent to the process whose process ID is *pid*.

pid == 0 The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal.

The term "all processes" excludes an implementation-defined set of system processes. For most Unix systems this set of system processes includes the swapper (pid 0), `init` (pid 1), and the pagedaemon (pid 2).

pid < 0 The signal is sent to all processes whose process group ID equals the absolute value of *pid* and for which the sender has permission to send the signal.

Again, the set of "all processes" excludes certain system processes, as described earlier.

pid == −1 POSIX.1 leaves this condition as unspecified.

> SVR4 and 4.3+BSD use this for what they call *broadcast signals*. These broadcast signals are never sent to the set of system processes described previously. 4.3+BSD also never sends a broadcast signal to the process sending the signal. If the caller is the superuser, the signal is sent to all processes. If the caller is not the superuser, the signal is sent to all processes whose real user ID or saved set-user-ID equals the real user ID or effective user ID of the caller. These broadcast signals should be used only for administrative purposes (such as a superuser process that is about to shut down the system).

As we've mentioned, a process needs permission to send a signal to some other process. The superuser can send a signal to any process. For others, the basic rule is that the real or effective user ID of the sender has to equal the real or effective user ID of the receiver. If the implementation supports `_POSIX_SAVED_IDS` (as does SVR4) then the saved set-user-ID of the receiver is checked instead of its effective user ID.

There is also one special case for the permission testing: if the signal being sent is `SIGCONT` then a process can send it to any other process that is a member of the same session.

POSIX.1 defines signal number 0 as the null signal. If the *signo* argument is 0, then the normal error checking is performed by `kill`, but no signal is sent. This is often used to determine if a specific process still exists. If we send the process the null signal

and it doesn't exist, kill returns –1 and errno is set to ESRCH. Be aware, however, that Unix systems recycle process IDs after some amount of time, so the existence of a process with a given process ID does not mean it's the process that you think it is.

If the call to kill causes the signal to be generated for the calling process and, if the signal is not blocked, either *signo* or some other pending, unblocked signal is delivered to the process before kill returns.

10.10 alarm and pause Functions

The alarm function allows us to set a timer that will expire at a specified time in the future. When the timer expires, the SIGALRM signal is generated. If we ignore or don't catch this signal, its default action is to terminate the process.

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

Returns: 0 or number of seconds until previously set alarm

The *seconds* value is the number of clock seconds in the future when the signal should be generated. Be aware that when that time occurs, the signal is generated by the kernel, but there could be additional time before the process gets control to handle the signal because of processor scheduling delays.

> Earlier versions of Unix warned that the signal could also be sent up to 1 second early. POSIX.1 does not allow this.

There is only one of these alarm clocks per process. If, when we call alarm, there is a previously registered alarm clock for the process that has not yet expired, the number of seconds left for that alarm clock is returned as the value of this function. That previously registered alarm clock is replaced by the new value.

If there is a previously registered alarm clock for the process that has not yet expired and if the *seconds* value is 0, the previous alarm clock is cancelled. The number of seconds left for that previous alarm clock is still returned as the value of the function.

Although the default action for SIGALRM is to terminate the process, most processes that use an alarm clock catch this signal. If the process then wants to terminate, it can perform whatever cleanup is required before terminating.

The pause function suspends the calling process until a signal is caught.

```
#include <unistd.h>

int pause(void);
```

Returns: –1 with errno set to EINTR

The only time pause returns is if a signal handler is executed and that handler returns. In that case, pause returns –1 with errno set to EINTR.

Example

Using `alarm` and `pause` we can put ourself to sleep for a specified amount of time. The `sleep1` function in Program 10.4 does this.

```
#include    <signal.h>
#include    <unistd.h>

static void
sig_alrm(int signo)
{
    return; /* nothing to do, just return to wake up the pause */
}

unsigned int
sleep1(unsigned int nsecs)
{
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(nsecs);
    alarm(nsecs);       /* start the timer */
    pause();            /* next caught signal wakes us up */
    return( alarm(0) ); /* turn off timer, return unslept time */
}
```

Program 10.4 Simple, incomplete implementation of `sleep`.

This function looks like the `sleep` function, which we describe in Section 10.19, but this simple implementation has problems.

1. If the caller already has an alarm set, that alarm is erased by the first call to `alarm`.

   We can correct this by looking at the return value from the first call to `alarm`. If the number of seconds until some previously set alarm is less than the argument, then we should wait only until the previously set alarm expires. If the previously set alarm will go off after ours, then before returning we should reset this alarm to occur at its designated time in the future.

2. We have modified the disposition for `SIGALRM`. If we're writing a function for others to call, we should save the disposition when we're called and restore it when we're done.

   We can correct this by saving the return value from `signal` and resetting the disposition before we return.

3. There is a race condition between the first call to `alarm` and the call to `pause`. It's possible on a busy system for the alarm to go off and the signal handler be called before we call `pause`. If that happens, the caller is suspended forever in the call to `pause` (assuming some other signal isn't caught).

Earlier implementations of `sleep` looked like our program, with problems 1 and 2 corrected as described. There are two ways to correct problem 3. The first uses `setjmp`,

which we show later. The other uses sigprocmask and sigsuspend, and we describe it in Section 10.19. □

Example

The SVR2 implementation of sleep used setjmp and longjmp (Section 7.10) to avoid the race condition described in problem 3 earlier. A simple version of this function, called sleep2 is shown in Program 10.5. (To reduce the size of this example, we don't handle problems 1 and 2 described earlier.)

```
#include    <setjmp.h>
#include    <signal.h>
#include    <unistd.h>

static jmp_buf  env_alrm;

static void
sig_alrm(int signo)
{
    longjmp(env_alrm, 1);
}

unsigned int
sleep2(unsigned int nsecs)
{
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(nsecs);
    if (setjmp(env_alrm) == 0) {
        alarm(nsecs);       /* start the timer */
        pause();            /* next caught signal wakes us up */
    }
    return( alarm(0) );     /* turn off timer, return unslept time */
}
```

Program 10.5 Another (imperfect) implementation of sleep.

In this function the race condition from Program 10.4 has been avoided. Even if the pause is never executed, when the SIGALRM occurs, the sleep2 function returns.

There is, however, another subtle problem with the sleep2 function that involves its interaction with other signals. If the SIGALRM interrupts some other signal handler, when we call longjmp it aborts the other signal handler. Program 10.6 shows this scenario. The loop in the SIGINT handler was written so that it executes for longer than 5 seconds on the system used by the author. We just want it to execute longer than the argument to sleep2. The integer j is declared volatile to prevent an optimizing compiler from discarding the loop. Executing Program 10.6 gives us

```
$ a.out
^?                              we type our interrupt character
sig_int starting
sleep2 returned: 0
```

```
#include    <signal.h>
#include    "ourhdr.h"

unsigned int    sleep2(unsigned int);
static void     sig_int(int);

int
main(void)
{
    unsigned int    unslept;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");

    unslept = sleep2(5);
    printf("sleep2 returned: %u\n", unslept);

    exit(0);
}

static void
sig_int(int signo)
{
    int             i;
    volatile int    j;

    printf("\nsig_int starting\n");
    for (i = 0; i < 2000000; i++)
        j += i * i;
    printf("sig_int finished\n");
    return;
}
```

Program 10.6 Calling `sleep2` from a program that catches other signals.

We can see that the `longjmp` from the `sleep2` function aborted the other signal handler, `sig_int`, even though it wasn't finished. This is what you'll encounter if you mix the SVR2 `sleep` function with other signal handling. See Exercise 10.3. □

The purpose of these two examples, the `sleep1` and `sleep2` functions, is to show the pitfalls in dealing naively with signals. The following sections will show ways around all these problems, so we can handle signals reliably, without interfering with other pieces of code.

Example

A common use for `alarm`, in addition to implementing the `sleep` function, is to put an upper time limit on operations that can block. For example, if we have a read operation

on a device that can block (a "slow" device, as described in Section 10.5) we might want the read to timeout after some amount of time. Program 10.7 does this, reading one line from standard input and writing it to standard output.

```
#include    <signal.h>
#include    "ourhdr.h"

static void sig_alrm(int);

int
main(void)
{
    int     n;
    char    line[MAXLINE];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");
    alarm(10);
    if ( (n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("read error");
    alarm(0);

    write(STDOUT_FILENO, line, n);

    exit(0);
}

static void
sig_alrm(int signo)
{
    return; /* nothing to do, just return to interrupt the read */
}
```

Program 10.7 Calling read with a time out.

This sequence of code is seen in many Unix applications, but there are two problems with this program.

1. Program 10.7 has the same flaw that we described in Program 10.4: there is a race condition between the first call to `alarm` and the call to `read`. If the kernel blocks the process between these two function calls for longer than the alarm period, the `read` could block forever. Most operations of this type use a long alarm period, such as a minute or more, making this unlikely, but nevertheless it is a race condition.
2. If system calls are automatically restarted, the `read` is not interrupted when the `SIGALRM` signal handler returns. In this case the time out does nothing.

Here we specifically want a slow system call to be interrupted. POSIX.1, however, does not give us a portable way to do this. □

Example

Let's redo the preceding example using `longjmp`. This way we don't need to worry whether a slow system call is interrupted or not.

```
#include    <setjmp.h>
#include    <signal.h>
#include    "ourhdr.h"

static void     sig_alrm(int);
static jmp_buf  env_alrm;

int
main(void)
{
    int     n;
    char    line[MAXLINE];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");

    if (setjmp(env_alrm) != 0)
        err_quit("read timeout");

    alarm(10);
    if ( (n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("read error");
    alarm(0);

    write(STDOUT_FILENO, line, n);

    exit(0);
}

static void
sig_alrm(int signo)
{
    longjmp(env_alrm, 1);
}
```

Program 10.8 Calling `read` with a time out, using `longjmp`.

This version works as expected, regardless of whether the system restarts interrupted system calls or not. Realize, however, that we still have the problem of interactions with other signal handlers, as in Program 10.5. □

If we want to set a time limit on an I/O operation we need to use `longjmp`, as shown previously, realizing its possible interaction with other signal handlers. Another option is to use the `select` or `poll` functions, described in Sections 12.5.1 and 12.5.2.

10.11 Signal Sets

We need a data type to represent multiple signals—a *signal set*. We'll use this with functions such as `sigprocmask` (in the next section) to tell the kernel not to allow any of the signals in the set to occur. As we mentioned earlier, the number of different signals can exceed the number of bits in an integer, so in general we can't use one bit per signal in an integer. POSIX.1 defines the data type `sigset_t` to contain a signal set and the following five functions to manipulate signal sets.

```
#include <signal.h>

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signo);

int sigdelset(sigset_t *set, int signo);
```

All four return: 0 if OK, –1 on error

```
int sigismember(const sigset_t *set, int signo);
```

Returns: 1 if true, 0 if false

The function `sigemptyset` initializes the signal set pointed to by *set* so that all signals are excluded. The function `sigfillset` initializes the signal set so that all signals are included. All applications have to call either `sigemptyset` or `sigfillset` once for each signal set, before using the signal set. This is because we cannot assume that the C initialization for external and static variables (0) corresponds to the implementation of signal sets on a given system.

Once we have initialized a signal set, we can add and delete specific signals in the set. The function `sigaddset` adds a single signal to an existing set, and `sigdelset` removes a single signal from a set. We'll see in all the functions that take a signal set as an argument that we always pass the address of the signal set as the argument.

Implementation

If the implementation has fewer signals than bits in an integer, a signal set can be implemented using one bit per signal. Most implementations of 4.3+BSD, for example, have 31 signals and 32-bit integers. `sigemptyset` zeroes the integer and `sigfillset` turns on all the bits in the integer. These two functions can be implemented as macros in the `<signal.h>` header:

```
#define sigemptyset(ptr)   ( *(ptr) = 0 )
#define sigfillset(ptr)    ( *(ptr) = ~(sigset_t)0, 0 )
```

Note that `sigfillset` must return 0, in addition to setting all the bits on in the signal set, so we use C's comma operator, which returns the value after the comma as the value of the expression.

Using this implementation `sigaddset` turns on a single bit and `sigdelset` turns off a single bit. `sigismember` tests a certain bit. Since there is never a signal numbered 0, we subtract 1 from the signal number to obtain the bit to manipulate. Program 10.9 implements these functions.

```
#include    <signal.h>
#include    <errno.h>

#define SIGBAD(signo)    ((signo) <= 0 || (signo) >= NSIG)
    /* <signal.h> usually defines NSIG to include signal number 0 */

int
sigaddset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    *set |= 1 << (signo - 1);       /* turn bit on */
    return(0);
}

int
sigdelset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    *set &= ~(1 << (signo - 1));    /* turn bit off */
    return(0);
}

int
sigismember(const sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    return( (*set & (1 << (signo - 1))) != 0 );
}
```

Program 10.9 An implementation of `sigaddset`, `sigdelset`, and `sigismember`.

We might be tempted to implement these three functions as one-line macros in the `<signal.h>` header, but POSIX.1 requires us to check the signal number argument for validity and set `errno` if it is invalid. This is harder to do in a macro than a function.

10.12 `sigprocmask` Function

Recall from Section 10.8 that the signal mask of a process is the set of signals currently blocked from delivery to that process. A process can examine or change (or both) its signal mask by calling the following function.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

Returns: 0 if OK, –1 on error

First, if *oset* is a nonnull pointer, the current signal mask for the process is returned through *oset*.

Second, if *set* is a nonnull pointer, then the *how* argument indicates how the current signal mask is modified. Figure 10.4 describes the different values for *how*. `SIG_BLOCK` is an inclusive-OR operation while `SIG_SETMASK` is an assignment.

how	Description
`SIG_BLOCK`	The new signal mask for the process is the union of its current signal mask and the signal set pointed to by *set*. That is, *set* contains the additional signals that we want to block.
`SIG_UNBLOCK`	The new signal mask for the process is the intersection of its current signal mask and the complement of the signal set pointed to by *set*. That is, *set* contains the signals that we want to unblock.
`SIG_SETMASK`	The new signal mask for the process is the value pointed to by *set*.

Figure 10.4 Ways to change current signal mask using `sigprocmask`.

If *set* is a null pointer, the signal mask of the process is not changed, and the value of *how* is not significant.

If there are any pending, unblocked signals after the call to `sigprocmask`, at least one of these signals is delivered to the process before `sigprocmask` returns.

Example

Program 10.10 shows a function that prints the names of the signals in the signal mask of the calling process. We call this function from Program 10.14 and Program 10.15. To save space we don't test the signal mask for every signal that we listed in Figure 10.1. (See Exercise 10.9.) □

10.13 `sigpending` Function

`sigpending` returns the set of signals that are blocked from delivery and currently pending for the calling process. The set of signals is returned through the *set* argument.

```
#include <signal.h>

int sigpending(sigset_t *set);
```

Returns: 0 if OK, –1 on error

```
#include    <errno.h>
#include    <signal.h>
#include    "ourhdr.h"

void
pr_mask(const char *str)
{
    sigset_t    sigset;
    int         errno_save;

    errno_save = errno;     /* we can be called by signal handlers */
    if (sigprocmask(0, NULL, &sigset) < 0)
        err_sys("sigprocmask error");

    printf("%s", str);
    if (sigismember(&sigset, SIGINT))   printf("SIGINT ");
    if (sigismember(&sigset, SIGQUIT))  printf("SIGQUIT ");
    if (sigismember(&sigset, SIGUSR1))  printf("SIGUSR1 ");
    if (sigismember(&sigset, SIGALRM))  printf("SIGALRM ");
        /* remaining signals can go here */
    printf("\n");
    errno = errno_save;
}
```

Program 10.10 Print the signal mask for the process.

Example

Program 10.11 shows many of the signal features that we've been describing. The process blocks `SIGQUIT`, saving its current signal mask (to reset later), and then goes to sleep for 5 seconds. Any occurrence of the quit signal during this period is blocked and won't be delivered until the signal is unblocked. At the end of the 5 second sleep we check if the signal is pending and unblock the signal.

Note that we saved the old mask when we blocked the signal. To unblock the signal we did a `SIG_SETMASK` of the old mask. Alternately, we could `SIG_UNBLOCK` only the signal that we had blocked. Be aware, however, if we write a function that can be called by others and if we need to block a signal in our function, we can't use `SIG_UNBLOCK` to unblock the signal. In this case we have to use `SIG_SETMASK` and reset the signal mask to its prior value, because it's possible that the caller had specifically blocked this signal before calling our function. We'll see an example of this in the `system` function in Section 10.18.

If we generate the quit signal during this sleep period, the signal is now pending and unblocked, so it is delivered before `sigprocmask` returns. We'll see this occur because the `printf` in the signal handler is output before the `printf` that follows the call to `sigprocmask`.

```
#include    <signal.h>
#include    "ourhdr.h"

static void sig_quit(int);

int
main(void)
{
    sigset_t    newmask, oldmask, pendmask;

    if (signal(SIGQUIT, sig_quit) == SIG_ERR)
        err_sys("can't catch SIGQUIT");

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
                    /* block SIGQUIT and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    sleep(5);       /* SIGQUIT here will remain pending */

    if (sigpending(&pendmask) < 0)
        err_sys("sigpending error");
    if (sigismember(&pendmask, SIGQUIT))
        printf("\nSIGQUIT pending\n");

                    /* reset signal mask which unblocks SIGQUIT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    printf("SIGQUIT unblocked\n");

    sleep(5);       /* SIGQUIT here will terminate with core file */

    exit(0);
}

static void
sig_quit(int signo)
{
    printf("caught SIGQUIT\n");

    if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
        err_sys("can't reset SIGQUIT");
    return;
}
```

Program 10.11 Example of signal sets and `sigprocmask`.

The process then goes to sleep for another 5 seconds. If we generate the quit signal during this sleep period, it should terminate the process, since we reset the handling of the signal to its default when we caught it. In the following output, the terminal prints ^\ when we input Control-backslash, the terminal quit character.

```
$ a.out
^\                          generate signal once (before 5 seconds are up)
SIGQUIT pending             after return from sleep
caught SIGQUIT              in signal handler
SIGQUIT unblocked           after return from sigprocmask
^\Quit(coredump)            generate signal again
$ a.out
^\^\^\^\^\^\^\^\^\^\        generate signal 10 times (before 5 seconds are up)
SIGQUIT pending
caught SIGQUIT              signal is generated only once
SIGQUIT unblocked
^\Quit(coredump)            generate signal again
```

(The message `Quit(coredump)` is printed by the shell when it sees that its child terminated abnormally.) Notice that when we run the program the second time we generate the quit signal 10 times while the process is asleep, yet the signal is delivered only once to the process when it's unblocked. This demonstrates that signals are not queued on this system. □

10.14 `sigaction` Function

The `sigaction` function allows us to examine or modify (or both) the action associated with a particular signal. This function supersedes the `signal` function from earlier releases of Unix. Indeed, at the end of this section we show an implementation of `signal` using `sigaction`.

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *act,
                         struct sigaction *oact);
```

Returns: 0 if OK, –1 on error

The argument *signo* is the signal number whose action we are examining or modifying. If the *act* pointer is nonnull, we are modifying the action. If the *oact* pointer is nonnull, the system returns the previous action for the signal. This function uses the following structure

```
struct sigaction {
  void     (*sa_handler)(); /* addr of signal handler,
                               or SIG_IGN, or SIG_DFL */
  sigset_t sa_mask;         /* additional signals to block */
  int      sa_flags;        /* signal options, Figure 10.5 */
};
```

When changing the action for a signal, if the sa_handler points to a signal-catching function (as opposed to the constants SIG_IGN or SIG_DFL) then the sa_mask field specifies a set of signals that are added to the signal mask of the process before the signal-catching function is called. If and when the signal-catching function returns, the signal mask of the process is reset to its previous value. This way we are able to block certain signals whenever a signal handler is invoked. This new signal mask that is installed by the system when the signal handler is invoked automatically includes the signal being delivered. Hence, we are guaranteed that whenever we are processing a given signal, another occurrence of that same signal is blocked until we're finished processing the first occurrence. Recall from Section 10.8 that additional occurrences of the same signal are usually not queued. If the signal occurs five times while its blocked, when we unblock the signal the signal-handling function for that signal will usually be invoked only one time.

Once we install an action for a given signal, that action remains installed until we explicitly change it by calling sigaction. Unlike earlier systems with their unreliable signals, POSIX.1 requires that a signal handler remain installed until explicitly changed.

The sa_flags field of the *act* structure specifies various options for the handling of this signal. Figure 10.5 details the meaning of these options when set.

Option	POSIX.1	SVR4	4.3+BSD	Description
SA_NOCLDSTOP	•	•	•	If *signo* is SIGCHLD, do not generate this signal when a child process stops (job control). This signal is still generated, of course, when a child terminates (but see the SVR4-specific SA_NOCLDWAIT option below).
SA_RESTART		•	•	System calls interrupted by this signal are automatically restarted. (Refer to Section 10.5.)
SA_ONSTACK		•	•	If an alternate stack has been declared with sigaltstack(2), this signal is delivered to the process on the alternate stack.
SA_NOCLDWAIT		•		If *signo* is SIGCHLD, this option causes the system not to create zombie processes when children of the calling process terminate. If the calling process subsequently calls wait, it blocks until all its child processes have terminated and then returns –1 with errno set to ECHILD. (Recall Section 10.7.)
SA_NODEFER		•		When this signal is caught, the signal is not automatically blocked by the system while the signal-catching function executes. Note that this type of operation corresponds to the earlier unreliable signals.
SA_RESETHAND		•		The disposition for this signal is reset to SIG_DFL on entry to the signal-catching function. Note that this type of operation corresponds to the earlier unreliable signals.
SA_SIGINFO		•		This option provides additional information to a signal handler. Refer to Section 10.21 for additional details.

Figure 10.5 Option flags (sa_flags) for the handling of each signal.

Example—`signal` Function

Let's now implement the `signal` function using `sigaction`. This is what 4.3+BSD does (and what a note in the POSIX.1 Rationale states was the intent of POSIX). SVR4, on the other hand, provides a `signal` function that provides the older, unreliable signal semantics. Unless you specifically require these older, unreliable semantics (for backward compatibility), under SVR4 you should use the following implementation of `signal` or call `sigaction` directly. (As you might guess, an implementation of `signal` under SVR4 with the old semantics could call `sigaction` specifying `SA_RESETHAND` and `SA_NODEFER`.) All the examples in this text that call `signal` call the function shown in Program 10.12.

```
/* Reliable version of signal(), using POSIX sigaction().  */

#include    <signal.h>
#include    "ourhdr.h"

Sigfunc *
signal(int signo, Sigfunc *func)
{
    struct sigaction    act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (signo == SIGALRM) {
#ifdef  SA_INTERRUPT
        act.sa_flags |= SA_INTERRUPT;    /* SunOS */
#endif
    } else {
#ifdef  SA_RESTART
        act.sa_flags |= SA_RESTART;      /* SVR4, 4.3+BSD */
#endif
    }
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

Program 10.12 An implementation of `signal` using `sigaction`.

Note that we must use `sigemptyset` to initialize the `sa_mask` member of the structure. We're not guaranteed that

```
act.sa_mask = 0;
```

does the same thing.

We intentionally try to set the `SA_RESTART` flag, for all signals other than `SIGALRM`, so that any system call interrupted by these other signal is automatically restarted. The reason we don't want `SIGALRM` restarted is to allow us to set a time out for I/O operations. (Recall the discussion of Program 10.7.)

Some systems (such as SunOS) define the SA_INTERRUPT flag. These systems restart interrupted system calls by default, so specifying this flag causes system calls to be interrupted. □

Example—signal_intr Function

Program 10.13 is a version of the signal function that tries to prevent any interrupted system calls from being restarted.

```
#include    <signal.h>
#include    "ourhdr.h"

Sigfunc *
signal_intr(int signo, Sigfunc *func)
{
    struct sigaction    act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
#ifdef  SA_INTERRUPT    /* SunOS */
    act.sa_flags |= SA_INTERRUPT;
#endif
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

Program 10.13 The signal_intr function.

We specify the SA_INTERRUPT flag, if defined by the system, to prevent interrupted system calls from being restarted. □

10.15 sigsetjmp and siglongjmp Functions

In Section 7.10 we described the setjmp and longjmp functions that can be used for nonlocal branching. The longjmp function is often called from a signal handler to return to the main loop of a program, instead of returning from the handler. Indeed, the ANSI C standard states that a signal handler can either return or call abort, exit, or longjmp. We saw this in Programs 10.5 and 10.8.

There is a problem in calling longjmp. When a signal is caught, the signal-catching function is entered with the current signal automatically being added to the signal mask of the process. This prevents subsequent occurrences of that signal from interrupting the signal handler. If we longjmp out of the signal handler, what happens to the signal mask for the process?

> Under 4.3+BSD setjmp and longjmp save and restore the signal mask. SVR4, however, does not do this. 4.3+BSD provides the functions _setjmp and _longjmp that do not save and restore the signal mask.

To allow either form of behavior, POSIX.1 does not specify the effect of `setjmp` and `longjmp` on signal masks. Instead, two new functions, `sigsetjmp` and `siglongjmp`, are defined by POSIX.1. These two functions should always be used when branching from a signal handler.

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);

            Returns: 0 if called directly, nonzero if returning from a call to siglongjmp

void siglongjmp(sigjmp_buf env, int val);
```

The only difference between these functions and the `setjmp` and `longjmp` functions is that `sigsetjmp` has an additional argument. If *savemask* is nonzero then `sigsetjmp` also saves the current signal mask of the process in *env*. When `siglongjmp` is called, if the *env* argument was saved by a call to `sigsetjmp` with a nonzero *savemask*, then `siglongjmp` restores the saved signal mask.

Example

Program 10.14 demonstrates how the signal mask that is installed by the system when a signal handler is invoked automatically includes the signal being caught. It also illustrates the use of the `sigsetjmp` and `siglongjmp` functions.

```
#include    <signal.h>
#include    <setjmp.h>
#include    <time.h>
#include    "ourhdr.h"

static void                     sig_usr1(int), sig_alrm(int);
static sigjmp_buf               jmpbuf;
static volatile sig_atomic_t    canjump;

int
main(void)
{
    if (signal(SIGUSR1, sig_usr1) == SIG_ERR)
        err_sys("signal(SIGUSR1) error");
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");
    pr_mask("starting main: ");     /* Program 10.10 */

    if (sigsetjmp(jmpbuf, 1)) {
        pr_mask("ending main: ");
        exit(0);
    }
    canjump = 1;    /* now sigsetjmp() is OK */

    for ( ; ; )
        pause();
}
```

```
static void
sig_usr1(int signo)
{
    time_t  starttime;

    if (canjump == 0)
        return;     /* unexpected signal, ignore */

    pr_mask("starting sig_usr1: ");

    alarm(3);               /* SIGALRM in 3 seconds */

    starttime = time(NULL);
    for ( ; ; )             /* busy wait for 5 seconds */
        if (time(NULL) > starttime + 5)
            break;

    pr_mask("finishing sig_usr1: ");

    canjump = 0;
    siglongjmp(jmpbuf, 1);  /* jump back to main, don't return */
}

static void
sig_alrm(int signo)
{
    pr_mask("in sig_alrm: ");
    return;
}
```

Program 10.14 Example of signal masks, `sigsetjmp`, and `siglongjmp`.

This program demonstrates another technique that should be used whenever `siglongjmp` is being called from a signal handler. We set the variable `canjump` nonzero only after we've called `sigsetjmp`. This variable is also examined in the signal handler, and `siglongjmp` is called only if the flag `canjump` is nonzero. This provides protection against the signal handler being called at some earlier or later time, when the jump buffer isn't initialized by `sigsetjmp`. (In this trivial program we terminate quickly after the `siglongjmp`, but in larger programs the signal handler may remain installed long after the `siglongjmp`.) Providing this type of protection usually isn't required with `longjmp` in normal C code (as opposed to a signal handler). Since a signal can occur at *any* time, however, we need the added protection in a signal handler.

Here we use the data type `sig_atomic_t`, which is defined by the ANSI C standard to be the type of variable that can be written without being interrupted. By this we mean that a variable of this type should not extend across page boundaries on a system with virtual memory and can be accessed with a single machine instruction, for example. We always include the ANSI type qualifier `volatile` for these data types too, since the variable is being accessed by two different threads of control—the `main` function and the asynchronously executing signal handler.

Figure 10.6 shows a time line for this program. We can divide Figure 10.6 into three parts: the left part (corresponding to `main`), the center part (`sig_usr1`), and the right part (`sig_alrm`). While the process is executing in the left part its signal mask is 0 (no

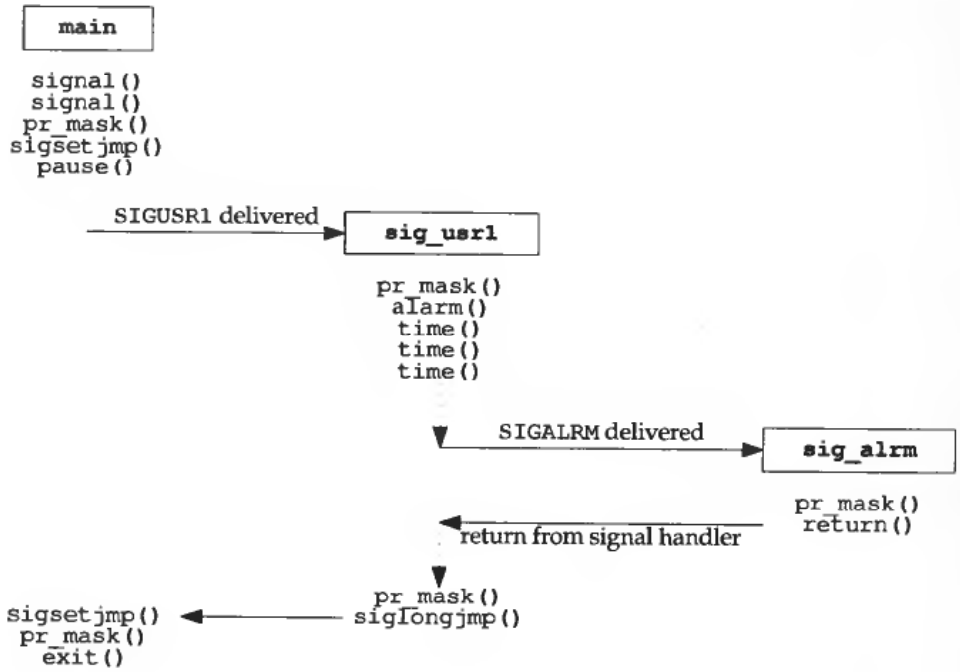


Figure 10.6 Time line for example program handling two signals.

signals are blocked). While executing in the center part its signal mask is `SIGUSR1`. While executing in the right part its signal mask is `SIGUSR1 | SIGALRM`.

Let's examine the actual output when Program 10.14 is executed.

```
$ a.out &                              start process in background
starting main:
[1]   531                              the job-control shell prints its process ID
$ kill -USR1 531                       send the process SIGUSR1
starting sig_usr1: SIGUSR1
$ in sig_alrm: SIGUSR1 SIGALRM
finishing sig_usr1: SIGUSR1
ending main:
                                       just press RETURN
[1] +  Done          a.out &
```

The output is as we expect: when a signal handler is invoked, the signal being caught is added to the current signal mask of the process. The original mask is restored when the signal handler returns. Also, `siglongjmp` restores the signal mask that was saved by `sigsetjmp`.

If we change Program 10.14 so that the calls to `sigsetjmp` and `siglongjmp` are replaced with calls to `_setjmp` and `_longjmp` instead, under 4.3+BSD the final line of output becomes

```
ending main: SIGUSR1
```

This means that the main function is executing with the SIGUSR1 signal blocked, after the call to _setjmp. This probably isn't what we want. □

10.16 sigsuspend Function

We have seen how we can change the signal mask for a process to block and unblock selected signals. We can use this to protect critical regions of code that we don't want interrupted by a signal. What if we want to unblock a signal and then pause, waiting for the previously blocked signal to occur? Assuming the signal is SIGINT, the incorrect way to do this is

```
sigset_t    newmask, oldmask;

sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
                /* block SIGINT and save current signal mask */
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
    err_sys("SIG_BLOCK error");

/* critical region of code */

                /* reset signal mask, which unblocks SIGINT */
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
    err_sys("SIG_SETMASK error");

pause();        /* wait for signal to occur */

/* continue processing */
```

There is a problem if the signal occurs between the unblocking and the pause. Any occurrence of the signal in this window of time is lost. This is another problem with the earlier unreliable signals.

To correct this problem we need a way to both reset the signal mask and put the process to sleep in a single atomic operation. This feature is provided by the sigsuspend function.

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

Returns: −1 with errno set to EINTR

The signal mask of the process is set to the value pointed to by *sigmask*. The process is also suspended until a signal is caught or until a signal occurs that terminates the process. If a signal is caught and if the signal handler returns, then sigsuspend returns and the signal mask of the process is set to its value before the call to sigsuspend.

Note that there is no successful return from this function. If it returns to the caller, it always returns −1 with errno set to EINTR (indicating an interrupted system call).

Example

Program 10.15 shows the correct way to protect a critical region of code from a specific signal.

```
#include    <signal.h>
#include    "ourhdr.h"

static void sig_int(int);

int
main(void)
{
    sigset_t    newmask, oldmask, zeromask;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");

    sigemptyset(&zeromask);

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGINT);
                    /* block SIGINT and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

                    /* critical region of code */
    pr_mask("in critical region: ");

                    /* allow all signals and pause */
    if (sigsuspend(&zeromask) != -1)
        err_sys("sigsuspend error");
    pr_mask("after return from sigsuspend: ");

                    /* reset signal mask which unblocks SIGINT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");

                    /* and continue processing ... */
    exit(0);
}

static void
sig_int(int signo)
{
    pr_mask("\nin sig_int: ");
    return;
}
```

Program 10.15 Protecting a critical region from a signal.

Note that when sigsuspend returns it sets the signal mask to its value before the call. In this example the SIGINT signal will be blocked. We therefore reset the signal mask to the value that we saved earlier (oldmask).

Running Program 10.15 produces the following output.

```
$ a.out
in critical region: SIGINT
^?                                    type our interrupt character
in sig_int: SIGINT
after return from sigsuspend: SIGINT
```

We can see that when sigsuspend returns, it restores the signal mask to its value before the call. □

Example

Another use of sigsuspend is to wait for a signal handler to set a global variable. In Program 10.16 we catch both the interrupt signal and the quit signal, but want only to wake up the main routine when the quit signal is caught. Sample output from this program is

```
$ a.out
^?                    type our interrupt character
interrupt
^?                    type our interrupt character again
interrupt
^?                    and again
interrupt
^\ $                  now terminate with quit character
```

□

For portability between non-POSIX systems that support ANSI C, and POSIX.1 systems, the only thing we should do within a signal handler is assign a value to a variable of type sig_atomic_t, and nothing else. POSIX.1 goes farther and specifies a list of functions that are safe to call from within a signal handler (Figure 10.3), but if we do this our code may not run correctly on non-POSIX systems.

Example

As another example of signals we show how signals can be used to synchronize a parent and child. Program 10.17 implements the five routines TELL_WAIT, TELL_PARENT, TELL_CHILD, WAIT_PARENT, and WAIT_CHILD from Section 8.8. We use the two user-defined signals: SIGUSR1 is sent by the parent to the child, and SIGUSR2 is sent by the child to the parent. In Program 14.3 we show another implementation of these five functions using pipes. □

```
#include    <signal.h>
#include    "ourhdr.h"

volatile sig_atomic_t   quitflag;   /* set nonzero by signal handler */

int
main(void)
{
    void        sig_int(int);
    sigset_t    newmask, oldmask, zeromask;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    if (signal(SIGQUIT, sig_int) == SIG_ERR)
        err_sys("signal(SIGQUIT) error");

    sigemptyset(&zeromask);

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
                    /* block SIGQUIT and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    while (quitflag == 0)
        sigsuspend(&zeromask);

    /* SIGQUIT has been caught and is now blocked; do whatever */
    quitflag = 0;
                    /* reset signal mask which unblocks SIGQUIT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");

    exit(0);
}

void
sig_int(int signo)  /* one signal handler for SIGINT and SIGQUIT */
{
    if (signo == SIGINT)
        printf("\ninterrupt\n");
    else if (signo == SIGQUIT)
        quitflag = 1;   /* set flag for main loop */
    return;
}
```

Program 10.16 Using sigsuspend to wait for a global variable to be set.

```
#include    <signal.h>
#include    "ourhdr.h"

static volatile sig_atomic_t    sigflag;
                                /* set nonzero by signal handler */
static sigset_t         newmask, oldmask, zeromask;

static void
sig_usr(int signo)  /* one signal handler for SIGUSR1 and SIGUSR2 */
{
    sigflag = 1;
    return;
}

void
TELL_WAIT()
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("signal(SIGQUIT) error");

    sigemptyset(&zeromask);

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGUSR1);
    sigaddset(&newmask, SIGUSR2);
        /* block SIGUSR1 and SIGUSR2, and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");
}

void
TELL_PARENT(pid_t pid)
{
    kill(pid, SIGUSR2);     /* tell parent we're done */
}

void
WAIT_PARENT(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask);  /* and wait for parent */

    sigflag = 0;
            /* reset signal mask to original value */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}
```

```
void
TELL_CHILD(pid_t pid)
{
    kill(pid, SIGUSR1);          /* tell child we're done */
}

void
WAIT_CHILD(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask);  /* and wait for child */

    sigflag = 0;
            /* reset signal mask to original value */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}
```

Program 10.17 Routines to allow a parent and child to synchronize.

The `sigsuspend` function is fine if we want to go to sleep while waiting for a signal to occur (as we've shown in the previous two examples), but what if we want to call other system functions while we're waiting? Unfortunately there is no bulletproof solution to this problem.

In Program 17.13 we encounter this scenario. We catch both `SIGINT` and `SIGALRM`, setting a global variable in each of the signal handlers if the signal occurs. Both signal handlers are installed using the `signal_intr` function, so that they interrupt any slow system call that is blocked. The signals are most likely to occur when we're blocked in a call to the `select` function (Section 12.5.1), waiting for input from a slow device. (This is especially true for `SIGALRM`, since we set the alarm clock to prevent us from blocking forever waiting for input.) The best we can do is the following.

```
if (intr_flag)      /* flag set by our SIGINT handler */
    handle_intr();
if (alrm_flag)      /* flag set by our SIGALRM handler */
    handle_alrm();
            /* signals occurring in here are lost */
while (select( ... ) < 0) {
    if (errno == EINTR) {
        if (alrm_flag)
            handle_alrm();
        else if (intr_flag)
            handle_intr();
    } else
        /* some other error */
}
```

We test each of the global flags before calling `select` and again if `select` returns an interrupted system call error. The problem occurs if either signal is caught between the first two `if` statements and the subsequent call to `select`. Signals occurring in here are lost, as indicated by the code comment. The signal handlers are called, and they set the appropriate global variable, but the `select` never returns (unless some data is ready to be read).

What we would like to be able to do is the following sequence of steps, in order.

1. Block `SIGINT` and `SIGALRM`.
2. Test the two global variables to see if either signal has occurred and, if so, handle the condition.
3. Call `select` (or any other system function, such as `read`) and unblock the two signals, as an atomic operation.

The `sigsuspend` function helps us only if step 3 is a `pause` operation.

10.17 `abort` Function

We mentioned earlier that the `abort` function causes abnormal program termination.

```
#include <stdlib.h>

void abort(void);
```

This function never returns

This function sends the `SIGABRT` signal to the process. A process should not ignore this signal.

ANSI C requires that if the signal is caught and the signal handler returns, `abort` still doesn't return to its caller. If this signal is caught, the only way the signal handler can't return is if it calls `exit`, `_exit`, `longjmp`, or `siglongjmp`. (Section 10.15 discusses the differences between `longjmp` and `siglongjmp`.) POSIX.1 also specifies that `abort` overrides the blocking or ignoring of the signal by the process.

The intent of letting the process catch the `SIGABRT` is to allow it to perform any cleanup that it wants to do, before the process terminates. If the process doesn't terminate itself from this signal handler, POSIX.1 states that, when the signal handler returns, `abort` terminates the process.

The ANSI C specification of this function leaves it up to the implementation whether output streams are flushed and whether temporary files (Section 5.13) are deleted. POSIX.1 goes further and requires that if the call to `abort` terminates the process, then it shall have the effect of calling `fclose` on all open standard I/O streams. But if the call to `abort` doesn't terminate the process, then it should have no effect on open streams. As we see later, this requirement is hard to implement.

Earlier versions of System V generated the SIGIOT signal from the abort function. Furthermore it was possible for a process to ignore this signal or to catch it and return from the signal handler, in which case abort returned to its caller.

4.3BSD generated the SIGILL signal. Before doing this the 4.3BSD function unblocked the signal and reset its disposition to SIG_DFL (terminate with core file). This prevented a process from either ignoring the signal or catching it.

SVR4 closes all standard I/O streams before generating the signal. On the other hand, 4.3+BSD does not. For defensive programming, if we want standard I/O streams to be flushed, we specifically do it before calling abort. We do this in the err_dump function (Appendix B).

Since most Unix implementations of tmpfile call unlink immediately after creating the file, the ANSI C warning about temporary files does not usually concern us.

Example

Program 10.18 implements the abort function, as specified by POSIX.1. The required handling of open standard I/O streams is hard to accomplish. We first see if the default action will occur, and if so we flush all the standard I/O streams. This is not equivalent to an fclose on all the open streams (since it just flushes them and doesn't close them), but when the process terminates the system closes all open files. If the process catches the signal and returns, we flush all the streams. (If the process catches the signal and doesn't return, we're not supposed to touch the standard I/O streams.) The only condition we don't handle is if the process catches the signal and calls _exit. In this case any unflushed standard I/O buffers in memory are discarded. We assume that a caller who catches the signal and specifically calls _exit, doesn't want the buffers flushed.

Recall from Section 10.9 that if calling kill causes the signal to be generated for the caller, and if the signal is not blocked (which we guarantee in Program 10.18), then the signal is delivered to the process before kill returns. This way we know that, if the call to kill returns, the process caught the signal and the signal handler returned. □

10.18 system Function

In Section 8.12 we showed an implementation of the system function. That version, however, did not do any signal handling. POSIX.2 requires that system ignore SIGINT and SIGQUIT and block SIGCHLD. Before showing a version that correctly handles these signals, let's see why we need to worry about signal handling.

```
#include <sys/signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void
abort(void)          /* POSIX-style abort() function */
{
    sigset_t            mask;
    struct sigaction    action;

        /* caller can't ignore SIGABRT, if so reset to default */
    sigaction(SIGABRT, NULL, &action);
    if (action.sa_handler == SIG_IGN) {
        action.sa_handler = SIG_DFL;
        sigaction(SIGABRT, &action, NULL);
    }

    if (action.sa_handler == SIG_DFL)
        fflush(NULL);           /* flush all open stdio streams */

        /* caller can't block SIGABRT; make sure it's unblocked */
    sigfillset(&mask);
    sigdelset(&mask, SIGABRT);  /* mask has only SIGABRT turned off */
    sigprocmask(SIG_SETMASK, &mask, NULL);

    kill(getpid(), SIGABRT);    /* send the signal */

        /* if we're here, process caught SIGABRT and returned */
    fflush(NULL);               /* flush all open stdio streams */

    action.sa_handler = SIG_DFL;
    sigaction(SIGABRT, &action, NULL);  /* reset disposition to default */
    sigprocmask(SIG_SETMASK, &mask, NULL);  /* just in case ... */

    kill(getpid(), SIGABRT);                /* and one more time */

    exit(1);    /* this should never be executed ... */
}
```

Program 10.18 POSIX.1 implementation of abort.

Example

Program 10.19 uses the version of `system` from Section 8.12 to invoke the ed(1) editor. (This editor has been part of Unix systems for a long time. We use it here because it is an interactive program that catches the interrupt and quit signals. If we invoke ed from a shell, and type the interrupt character, it catches the interrupt signal and prints a question mark. It also sets the disposition of the quit signal so that it is ignored.)

```
#include    <sys/types.h>
#include    <signal.h>
#include    "ourhdr.h"

static void sig_int(int), sig_chld(int);

int
main(void)
{
    int     status;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    if (signal(SIGCHLD, sig_chld) == SIG_ERR)
        err_sys("signal(SIGCHLD) error");

    if ( (status = system("/bin/ed")) < 0)
        err_sys("system() error");
    exit(0);
}

static void
sig_int(int signo)
{
    printf("caught SIGINT\n");
    return;
}

static void
sig_chld(int signo)
{
    printf("caught SIGCHLD\n");
    return;
}
```

Program 10.19 Using `system` to invoke the ed editor.

Program 10.19 catches both `SIGINT` and `SIGCHLD`. If we invoke it we get

```
$ a.out
a                                   append text to the editor's buffer
Here is one line of text
and another
.                                   period on a line by itself stops append mode
```

```
1,$p                          print first through last lines of buffer to see what's there
Here is one line of text
and another
w temp.foo                    write the buffer to a file
37                            editor says it wrote 37 bytes
q                             and leave the editor
caught SIGCHLD
```

What is happening when the editor terminates is that SIGCHLD is generated for the parent (the a.out process). We catch it and return from the signal handler. But if the parent is catching the SIGCHLD signal, it should be doing so because it has created its own children, so that it knows when its children have terminated. The catching of this signal in the parent should be blocked while the system function is executing. Indeed, this is what POSIX.2 specifies. Otherwise, when the child created by system terminates, it would fool the caller of system into thinking that one of its own children terminated.

If we execute the program again, this time sending an interrupt signal to the editor, we get

```
$ a.out
a                             append text to the editor's buffer
hello, world
.                             period on a line by itself stops append mode
1,$p                          print first through last lines to see what's there
hello, world
w temp.foo                    write the buffer to a file
13                            editor says it wrote 13 bytes
^?                            type our interrupt character
?                             editor catches signal, prints question mark
caught SIGINT                 and so does the parent process
q                             leave editor
caught SIGCHLD
```

Recall from Section 9.6 that typing the interrupt character causes the interrupt signal to be sent to all the processes in the foreground process group. Figure 10.7 shows the arrangement of the processes when the editor is running.

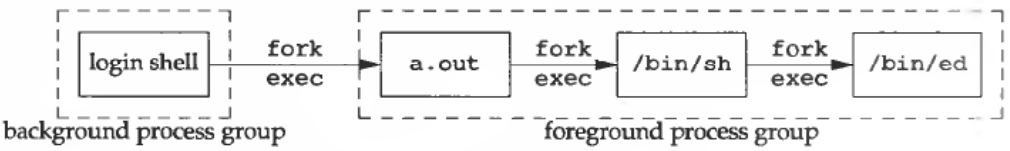


Figure 10.7 Foreground and background process groups for Program 10.19.

In this example SIGINT is sent to all three foreground processes. (The shell ignores it.) As we can see from the output, both the a.out process and the editor catch the signal. But when we're running another program with the system function, we shouldn't have both the parent and the child catching the two terminal-generated signals: interrupt and quit. These two signals should really be sent to the program that is running: the child. Since the command that is executed by system can be an interactive command (as is

the ed program in this example) and since the caller of system gives up control while the program executes, waiting for it to finish, the caller of system should not be receiving these two terminal-generated signals. This is why POSIX.2 specifies that the caller of system should ignore these two signals. □

Example

Program 10.20 shows an implementation of the system function with the required signal handling.

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    <errno.h>
#include    <signal.h>
#include    <unistd.h>

int
system(const char *cmdstring)    /* with appropriate signal handling */
{
    pid_t               pid;
    int                 status;
    struct sigaction    ignore, saveintr, savequit;
    sigset_t            chldmask, savemask;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with Unix */

    ignore.sa_handler = SIG_IGN;     /* ignore SIGINT and SIGQUIT */
    sigemptyset(&ignore.sa_mask);
    ignore.sa_flags = 0;
    if (sigaction(SIGINT, &ignore, &saveintr) < 0)
        return(-1);
    if (sigaction(SIGQUIT, &ignore, &savequit) < 0)
        return(-1);

    sigemptyset(&chldmask);          /* now block SIGCHLD */
    sigaddset(&chldmask, SIGCHLD);
    if (sigprocmask(SIG_BLOCK, &chldmask, &savemask) < 0)
        return(-1);

    if ( (pid = fork()) < 0) {
        status = -1;     /* probably out of processes */

    } else if (pid == 0) {           /* child */
            /* restore previous signal actions & reset signal mask */
        sigaction(SIGINT, &saveintr, NULL);
        sigaction(SIGQUIT, &savequit, NULL);
        sigprocmask(SIG_SETMASK, &savemask, NULL);

        execl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
        _exit(127);      /* exec error */
```

```
    } else {                            /* parent */
        while (waitpid(pid, &status, 0) < 0)
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
    }

        /* restore previous signal actions & reset signal mask */
    if (sigaction(SIGINT, &saveintr, NULL) < 0)
        return(-1);
    if (sigaction(SIGQUIT, &savequit, NULL) < 0)
        return(-1);
    if (sigprocmask(SIG_SETMASK, &savemask, NULL) < 0)
        return(-1);

    return(status);
}
```

Program 10.20 Correct POSIX.2 implementation of `system` function.

Many older texts show the ignoring of the interrupt and quit signals as follows:

```
if ( (pid = fork()) < 0)
    err_sys("fork error");
else if (pid == 0) {        /* child */
    execl(...);
    _exit(127);
}
/* parent */
old_intr = signal(SIGINT, SIG_IGN);
old_quit = signal(SIGQUIT, SIG_IGN);
waitpid(pid, &status, 0)
signal(SIGINT, old_intr);
signal(SIGQUIT, old_quit);
```

The problem with this sequence of code is that we have no guarantee after the `fork` whether the parent or child runs first. If the child runs first and the parent doesn't run for some time after, it's possible for an interrupt signal to be generated before the parent is able to change its disposition to be ignored. For this reason, in Program 10.20 we change the disposition of the signals before the `fork`.

Notice that we have to reset the dispositions of these two signals in the child before the call to `execl`. This allows `execl` to change their dispositions to the default, based on the caller's dispositions, as we described in Section 8.9. □

Return Value from `system`

Beware of the return value from `system`. It is the termination status of the shell, which isn't always the termination status of the command string. We saw some examples in

Program 8.13, and the results were as we expected: if we execute a simple command such as `date`, the termination status is 0. Executing the shell command `exit 44` gave us a termination status of 44. What happens with signals?

Let's run Program 8.14 and send some signals to the command that's executing.

```
$ tsys "sleep 30"
^?normal termination, exit status = 130    we type our interrupt key
$ tsys "sleep 30"
^\sh: 946 Quit                             we type our quit key
normal termination, exit status = 131
```

When we terminate the `sleep` with the interrupt signal, the `pr_exit` function (Program 8.3) thinks it terminated normally. The same thing happens when we kill the `sleep` with the quit key. What is happening here is that the Bourne shell has a poorly documented feature that its termination status is 128 plus the signal number, when the command it was executing is terminated by a signal. We can see this with the shell interactively.

```
$ sh                                       make sure we're running the Bourne shell
$ sh -c "sleep 30"
^?                                         type our interrupt key
$ echo $?                                  print termination status of last command
130
$ sh -c "sleep 30"
^\sh: 962 Quit - core dumped               type our quit key
$ echo $?                                  print termination status of last command
131
$ exit                                     leave Bourne shell
```

On the system being used, `SIGINT` has a value of 2 and `SIGQUIT` has a value of 3, giving us the shell's termination statuses of 130 and 131.

Let's try a similar example, but this time we'll send a signal directly to the shell and see what gets returned by `system`.

```
$ tsys "sleep 30" &                        start it in background this time
[1]     980
$ ps                                       look at the process IDs
  PID TT STAT  TIME COMMAND
  980 p3 S     0:00 tsys sleep 30
  981 p3 S     0:00 sh -c sleep 30
  982 p3 S     0:00 sleep 30
  985 p3 R     0:00 ps
$ kill -KILL 981                           kill the shell itself
abnormal termination, signal number = 9
[1] +  Done       tsys "sleep 30" &
```

Here we can see that the return value from `system` reports an abnormal termination only when the shell itself abnormally terminates.

When writing programs that use the `system` function, be sure to interpret the return value correctly. If we call `fork`, `exec`, and `wait` ourself, the termination status is different than if we call `system`.

10.19 sleep Function

We've used the sleep function in numerous examples throughout the text, and we showed two flawed implementations of it in Programs 10.4 and 10.5.

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Returns: 0 or number of unslept seconds

This function causes the calling process to be suspended until either

1. the amount of wall clock time specified by *seconds* has elapsed, or
2. a signal is caught by the process and the signal handler returns.

As with an alarm signal, the actual return may be at a time later than requested, because of other system activity.

In case 1 the return value is 0. When sleep returns early, because of some signal being caught (case 2), the return value is the number of unslept seconds (the requested time minus the actual time slept).

sleep can be implemented with the alarm function (Section 10.10), but this isn't required. If alarm is used, however, there can be interactions between the two functions. The POSIX.1 standard leaves all these interactions unspecified. For example, if we do an alarm(10) and 3 wall clock seconds later do a sleep(5), what happens? The sleep will return in 5 seconds (assuming some other signal isn't caught in that time), but will another SIGALRM be generated 2 seconds later? These details depend on the implementation.

SVR4 implements sleep using alarm. The sleep(3) manual page says that a previously scheduled alarm is properly handled. For example, in the preceding scenario, before sleep returns it will reschedule the alarm to happen 2 seconds later. sleep returns 0 in this case. (Obviously, sleep must save the address of the signal handler for SIGALRM and reset it before returning.) Also, if we do an alarm(6) and 3 wall clock seconds later do a sleep(5), the sleep returns in 3 seconds (when the alarm goes off), not in 5 seconds. Here the return value from sleep is 2 (the number of unslept seconds).

4.3+BSD, on the other hand, uses another technique: the interval timer provided by setitimer(2). This timer is independent of the alarm function, but there can still be interaction between a previously set interval timer and sleep. Also, even though the alarm timer (alarm) and interval timer (setitimer) are separate, they (unfortunately) use the same SIGALRM signal. Since sleep temporarily changes the address of the signal handler for this signal to its own function, there can still be unwanted interactions between alarm and sleep.

The moral in all this is to be intimately aware of how your system implements sleep if you have any intentions of mixing calls to sleep with any other timing functions.

Previous Berkeley-derived implementations of sleep did not provide any useful return information. This has been fixed in 4.3+BSD.

Example

Program 10.21 shows an implementation of the POSIX.1 `sleep` function. This function is a modification of Program 10.4 that handles signals reliably, avoiding the race condition in the earlier implementation. We still do not handle any interactions with previously set alarms. (As we mentioned, these interactions are explicitly undefined by POSIX.1.)

```
#include    <signal.h>
#include    <stddef.h>
#include    "ourhdr.h"

static void
sig_alrm(void)
{
    return; /* nothing to do, just returning wakes up sigsuspend() */
}

unsigned int
sleep(unsigned int nsecs)
{
    struct sigaction    newact, oldact;
    sigset_t            newmask, oldmask, suspmask;
    unsigned int        unslept;

    newact.sa_handler = sig_alrm;
    sigemptyset(&newact.sa_mask);
    newact.sa_flags = 0;
    sigaction(SIGALRM, &newact, &oldact);
                /* set our handler, save previous information */

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGALRM);
                /* block SIGALRM and save current signal mask */
    sigprocmask(SIG_BLOCK, &newmask, &oldmask);

    alarm(nsecs);

    suspmask = oldmask;
    sigdelset(&suspmask, SIGALRM);  /* make sure SIGALRM isn't blocked */

    sigsuspend(&suspmask);          /* wait for any signal to be caught */

    /* some signal has been caught, SIGALRM is now blocked */

    unslept = alarm(0);
    sigaction(SIGALRM, &oldact, NULL);  /* reset previous action */

                /* reset signal mask, which unblocks SIGALRM */
    sigprocmask(SIG_SETMASK, &oldmask, NULL);

    return(unslept);
}
```

Program 10.21 Reliable implementation of `sleep`.

It takes more code to write this reliable implementation than Program 10.4. We don't use any form of nonlocal branching (as we did in Program 10.5 to avoid the race condition between the `alarm` and `pause`), so there is no effect on other signal handlers that may be executing when the `SIGALRM` is handled. □

10.20 Job-Control Signals

From Figure 10.1 there are six signals that POSIX.1 considers the job-control signals.

`SIGCHLD`	Child process has stopped or terminated.
`SIGCONT`	Continue process, if stopped.
`SIGSTOP`	Stop signal (can't be caught or ignored).
`SIGTSTP`	Interactive stop signal.
`SIGTTIN`	Read from controlling terminal by member of a background process group.
`SIGTTOU`	Write to controlling terminal by member of a background process group.

Although POSIX.1 requires the system to support `SIGCHLD` only if the system supports job control, almost every version of Unix supports the signal. We have already described how this signal is generated when a child process terminates.

Most application programs don't handle these signals—interactive shells usually do all the work required to handle these signals. When we type the suspend character (usually Control-Z), `SIGTSTP` is sent to all processes in the foreground process group. When we tell the shell to resume a job in the foreground or background, the shell sends all the processes in the job the `SIGCONT` signal. Similarly, if `SIGTTIN` or `SIGTTOU` is delivered to a process, the process is stopped by default, and the job-control shell recognizes this and notifies us.

An exception is a process that is managing the terminal—the `vi`(1) editor, for example. It needs to know when the user wants to suspend it, so that it can restore the terminal's state to the way it was when `vi` was started. Also, when it resumes in the foreground it needs to set the terminal state back to way it wants it, and it needs to redraw the terminal screen. We see how a program such as `vi` handles this in the example that follows.

There are some interactions between the job-control signals. When any of the four stop signals are generated for a process (`SIGTSTP`, `SIGSTOP`, `SIGTTIN`, or `SIGTTOU`), any pending `SIGCONT` signal for that process is discarded. Similarly, when the `SIGCONT` signal is generated for a process, any pending stop signals for that same process are discarded.

Notice that the default action for `SIGCONT` is to continue the process, if it is stopped, otherwise the signal is ignored. Normally we don't have to do anything with this signal. When `SIGCONT` is generated for a process that is stopped, the process is continued, even if the signal is blocked or ignored.

Example

Program 10.22 demonstrates the normal sequence of code used when a program handles job control. This program just copies its standard input to its standard output, but comments are given in the signal handler for typical actions performed by a program that manages a screen. When Program 10.22 starts, it arranges to catch the `SIGTSTP` signal only if the signal's disposition is `SIG_DFL`. The reason is that when the program is started by a shell that doesn't support job control (`/bin/sh`, for example), the signal's disposition should be set to `SIG_IGN`. Actually, the shell doesn't explicitly ignore this signal, `init` sets the disposition of the three job-control signals `SIGTSTP`, `SIGTTIN`, and `SIGTTOU` to `SIG_IGN`. This disposition is then inherited by all login shells. Only a job-control shell should reset the disposition of these three signals to `SIG_DFL`.

When we type the suspend character, the process receives the `SIGTSTP` signal, and the signal handler is invoked. At this point we would do any terminal-related processing: move the cursor to the lower left corner, restore the terminal mode, and so on. We then send ourself the same signal, `SIGTSTP`, after resetting its disposition to its default (stop the process) and unblocking the signal. We have to unblock it since we're currently handling that same signal, and the system blocks it automatically while it's being caught. At this point the system stops the process. It is continued only when someone (usually the job-control shell, in response to an interactive `fg` command) sends it a `SIGCONT` signal. We don't catch `SIGCONT`. Its default disposition is to continue the stopped process, and when this happens the program continues as though it returned from the `kill` function. When the program is continued we reset the disposition for the `SIGTSTP` signal and do whatever terminal processing we want (we could redraw the screen, for example). □

We'll see another way to handle the special job-control suspend character in Chapter 18, when we don't use the signal, but recognize the special character ourself.

10.21 Additional Features

In this section we describe some additional implementation-dependent features of signals.

Signal Names

Some systems provide the array

```
extern char *sys_siglist[];
```

The array index is the signal number, giving a pointer to the character string name of the signal.

```
#include    <signal.h>
#include    "ourhdr.h"

#define BUFFSIZE    1024

static void sig_tstp(int);

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    /* only catch SIGTSTP if we're running with a job-control shell */
    if (signal(SIGTSTP, SIG_IGN) == SIG_DFL)
        signal(SIGTSTP, sig_tstp);

    while ( (n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");
    if (n < 0)
        err_sys("read error");

    exit(0);
}

static void
sig_tstp(int signo) /* signal handler for SIGTSTP */
{
    sigset_t    mask;

    /* ... move cursor to lower left corner, reset tty mode ... */

    /* unblock SIGTSTP, since it's blocked while we're handling it */
    sigemptyset(&mask);
    sigaddset(&mask, SIGTSTP);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);

    signal(SIGTSTP, SIG_DFL);   /* reset disposition to default */

    kill(getpid(), SIGTSTP);    /* and send the signal to ourself */

        /* we won't return from the kill until we're continued */

    signal(SIGTSTP, sig_tstp);  /* reestablish signal handler */

    /* ... reset tty mode, redraw screen ... */
    return;
}
```

Program 10.22 How to handle SIGTSTP.

These systems normally provide the function `psignal` also.

```
#include <signal.h>

void psignal(int signo, const char *msg);
```

The string *msg* (which is normally the name of the program) is output to the standard error, followed by a colon and a space, followed by a description of the signal, followed by a newline. This function is similar to `perror` (Section 1.7).

Both SVR4 and 4.3+BSD provide `sys_siglist` and the `psignal` function.

Additional Arguments to SVR4 Signal Handler

When we call `sigaction` to set the disposition for a signal, we can specify an `sa_flags` value of `SA_SIGINFO` (Figure 10.5). This causes two additional arguments to be passed to the signal handler. The integer signal number is always passed as the first argument. The second argument is either a null pointer or a pointer to a `siginfo` structure. (The third argument provides information about different threads of control within a single process, which we don't discuss.)

```
struct siginfo {
  int    si_signo;  /* signal number */
  int    si_errno;  /* if nonzero, errno value from <errno.h> */
  int    si_code;   /* additional info (depends on signal) */
  pid_t  si_pid;    /* sending process ID */
  uid_t  si_uid;    /* sending process real user ID */
  /* other fields also */
};
```

For hardware generated signals, such as `SIGFPE`, the `si_code` value gives additional information: `FPE_INTDIV` means integer divide by 0, `FPE_FLTDIV` means floating point divide by 0, and so on. If `si_code` is less than or equal to 0, it means the signal was generated by a user process that called `kill`(2). In this case the two elements `si_pid` and `si_uid` give additional information on the process that sent us the signal. Other information is available that depends on the signal being caught; see the SVR4 `siginfo`(5) manual page.

Additional Arguments to 4.3+BSD Signal Handler

4.3+BSD always calls a signal handler with three arguments.

```
handler(int signo, int code, struct sigcontext *scp);
```

The argument *signo* is the signal number, and *code* gives additional information for certain signals. For example, a *code* of `FPE_INTDIV_TRAP` for `SIGFPE` means integer divide by 0. The third argument, *scp*, is hardware dependent.

10.22 Summary

Signals are used in most nontrivial applications. An understanding of the hows and whys of signal handling is essential to advanced Unix programming. This chapter has been a long and thorough look at Unix signals. We started by looking at the warts in previous implementations of signals and how they manifest themselves. We then proceeded to the POSIX.1 reliable signal concept and all the related functions. Once we covered all these details, we were able to provide implementations of the POSIX.1 `abort`, `system`, and `sleep` functions. We finished with a look at the job-control signals.

Exercises

10.1 In Program 10.1 remove the `for (;;)` statement? What happens and why?

10.2 Implement the `raise` function.

10.3 Draw pictures of the stack frames when we run Program 10.5.

10.4 In Program 10.8 we showed a technique that's often used to set a time out on an I/O operation using `setjmp` and `longjmp`. The following code has also been seen:

```
signal(SIGALRM, sig_alrm);
alarm(60);
if (setjmp(env_alrm) != 0) {
    /* handle time out */
    ...
}
...
```

What else is wrong with this sequence of code?

10.5 Using only a single timer (either `alarm` or the higher precision `setitimer`) provide a set of functions that allows a process to set any number of timers.

10.6 Write the following program to test the parent–child synchronization functions in Program 10.17. The process creates a file and writes the integer 0 to the file. The process then calls `fork` and the parent and child alternate incrementing the counter in the file. Each time the counter is incremented, print which process is doing the increment (parent or child).

10.7 In Program 10.18 if the caller catches `SIGABRT` and returns from the signal handler, why do we go to the trouble of resetting the disposition to its default and call `kill` the second time, instead of just calling `_exit`?

10.8 Why do you think the designers of the SVR4 `siginfo` feature (Section 10.21) chose to pass the real user ID, instead of the effective user ID, in the `si_uid` field?

10.9 Rewrite Program 10.10 to handle all the signals from Figure 10.1. The function should consist of a single loop that iterates once for every signal in the current signal mask (not once for every possible signal).

10.10 Write a program that calls `sleep(60)` in an infinite loop. Every five times through the loop (every 5 minutes) fetch the current time-of-day and print the `tm_sec` field. Run the program overnight and explain the results. How would a program such as the BSD `cron` daemon, which runs every minute on the minute, handle this?

10.11 Modify Program 3.3 as follows: (a) change `BUFFSIZE` to 100; (b) catch the `SIGXFSZ` signal using the `signal_intr` function, printing a message when it's caught, and returning from the signal handler; and (c) print the return value from `write` if the requested number of bytes weren't written. Modify the soft `RLIMIT_FSIZE` resource limit (Section 7.11) to 1024 bytes and run your new program, copying a file that is larger than 1024 bytes. (Try to set the soft resource limit from your shell. If you can't do this from your shell, call `setrlimit` directly from the program.) Run this program on the different systems that you have access to. What happens and why?

10.12 Write a program that calls `fwrite` with a large buffer (a few megabytes). Before calling `fwrite`, call `alarm` to schedule a signal in 1 second. In your signal handler print that the signal was caught and return. Does the call to `fwrite` complete? What's happening?

11

Terminal I/O

11.1 Introduction

The handling of terminal I/O is a messy area, regardless of the operating system. Unix is no exception. The manual page for terminal I/O is usually one of the longest in most editions of the Unix manuals. The `termio` manual page in the SVID exceeds 16 pages.

With Unix, a schism formed in the late 1970s when System III developed a different set of terminal routines from Version 7. The System III style of terminal I/O continued through System V, and the Version 7 style became the standard for the Berkeley-derived systems. As with signals, this difference between the two worlds has been conquered by POSIX.1. In this chapter we look at all the POSIX.1 terminal functions, and some of the SVR4 and 4.3+BSD additions.

Part of the complexity of the terminal I/O system is because people use terminal I/O for so many different things: terminals, hardwired lines between computers, modems, printers, and so on. In later chapters we develop two programs to demonstrate terminal I/O: one communicates with a PostScript printer (Chapter 17) and the other allows us to talk to a modem and log in to a remote computer (Chapter 18).

11.2 Overview

There are two different modes for terminal I/O:

1. Canonical mode input processing. In this mode terminal input is processed as lines. The terminal driver returns at most one line per read request.
2. Noncanonical mode input processing. The input characters are not assembled into lines.

If we don't do anything special, canonical mode is the default. For example, if the shell redirects standard input to the terminal and we copy standard input to standard output using `read` and `write`, the terminal is in the canonical mode and each `read` returns at most one line. Programs that manipulate the entire screen, such as the `vi` editor, use noncanonical mode, since the commands may be single characters and are not terminated by newlines. Also, this editor doesn't want processing by the system of the special characters since the special characters may overlap with the editor commands. For example, the Control-D character is often the end-of-file character for the terminal, but it's also a `vi` command to scroll down one-half screen.

> The Version 7 and BSD-style terminal drivers support three different modes for terminal input: (a) cooked mode (the input is collected into lines and the special characters are processed), (b) raw mode (the input is not assembled into lines and there is no processing of special characters), and (c) cbreak mode (the input is not assembled into lines, but some of the special characters are processed). Program 11.10 shows a POSIX.1 function that places a terminal in cbreak or raw mode.

POSIX.1 defines 11 special input characters, 9 of which we can change. We've been using some of these throughout the text: the end-of-file character (usually Control-D) and the suspend character (usually Control-Z), for example. Section 11.3 describes each of these characters.

We can think of a terminal device as being controlled by a terminal driver, probably within the kernel. Each terminal device has an input queue and an output queue, shown in Figure 11.1.

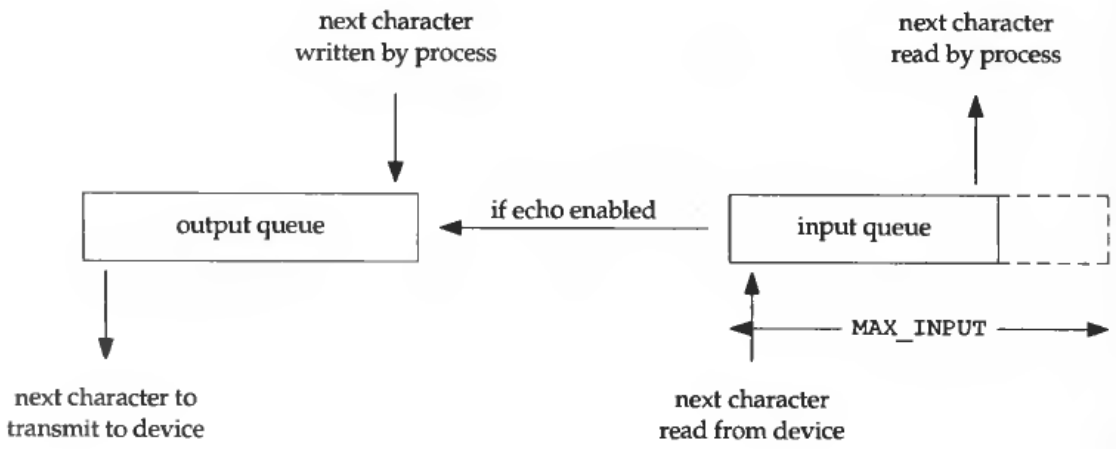


Figure 11.1 Logical picture of input and output queues for a terminal device.

There are several points to consider from this picture.

- There is an implied link between the input queue and the output queue, if echoing is enabled.

- The size of the input queue, `MAX_INPUT` (refer to Figure 2.5), may be finite. What the system does when the input queue for a particular device fills is implementation dependent. Most Unix systems echo the bell character when this happens.
- There is another input limit that we don't show here, `MAX_CANON`. This is the maximum number of bytes in a canonical input line.
- Although the output queue is normally of a finite size, there are no constants defining its size that are accessible to the program. This is because when the output queue starts to fill up, the kernel just puts the writing process to sleep until room is available.
- We'll see how the `tcflush` flush function allows us to flush either the input queue or the output queue. Similarly, when we describe the `tcsetattr` function, we'll see how we can tell the system to change the attributes of a terminal device only after the output queue is empty. (We want to do this, for example, if we're changing the output attributes.) We can also tell the system that when it changes the terminal attributes, to discard everything in the input queue also. (We want to do this if we're changing the input attributes or changing between canonical and noncanonical modes, so that previously entered characters aren't interpreted in the wrong mode.)

Most Unix systems implement all the canonical processing in a module called the *terminal line discipline*. We can think of this as a box that sits between the kernel's generic read and write functions and the actual device driver. We show this in Figure 11.2.

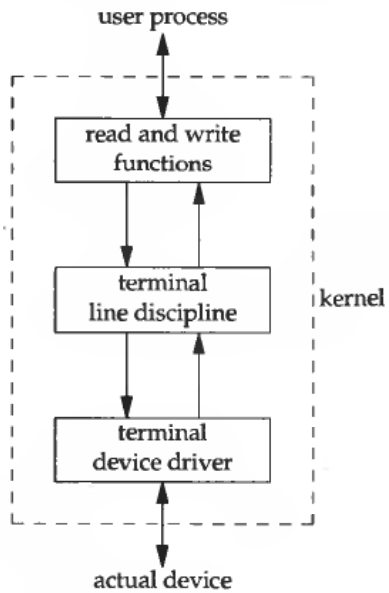


Figure 11.2 Terminal line discipline.

We'll return to this picture in Section 12.4 when we discuss the stream I/O system, and in Chapter 19 when we discuss pseudo terminals.

All the characteristics of a terminal device that we can examine and change are contained in a `termios` structure. This is defined in the header `<termios.h>`, which we use throughout this chapter.

```
struct termios {
  tcflag_t  c_iflag;    /* input flags */
  tcflag_t  c_oflag;    /* output flags */
  tcflag_t  c_cflag;    /* control flags */
  tcflag_t  c_lflag;    /* local flags */
  cc_t      c_cc[NCCS]; /* control characters */
};
```

Roughly speaking, the input flags control the input of characters by the terminal device driver (strip eighth bit on input, enable input parity checking, etc.), the output flags control the driver output (perform output processing, map newline to CR/LF, etc.), the control flags affect the RS-232 serial lines (ignore modem status lines, one or two stop bits per character, etc.), and the local flags affect the interface between the driver and the user (echo on or off, visually erase characters, enable terminal-generated signals, job control stop signal for background output, etc.).

The type `tcflag_t` is big enough to hold each of the flag values. It is often defined as an `unsigned long`. The `c_cc` array contains all the special characters that we can change. `NCCS` is the number of elements in this array and is typically between 11 and 18 (since most Unix implementations support more than the 11 POSIX-defined special characters). The `cc_t` type is large enough to hold each special character and is typically an `unsigned char`.

> Earlier versions of System V had a header named `<termio.h>` and a structure named `termio`. POSIX.1 added an `s` to the names, to differentiate them from their predecessors.

Figure 11.3 lists all the terminal flags that we can change to affect the characteristics of a terminal device. Note that even though POSIX.1 defines a common subset that both SVR4 and 4.3+BSD start from, both of these implementations have their own additions. These additions come from the historical differences between the two systems. We'll discuss each of these flag values in detail in Section 11.5.

Given all the options presented in Figure 11.3, how do we examine and change these characteristics of a terminal device? Figure 11.4 summarizes the various functions defined by POSIX.1 that operate on terminal devices. (We described `tcgetpgrp` and `tcsetpgrp` in Section 9.7.)

Note that POSIX.1 doesn't use the classic `ioctl` on terminal devices. Instead, it uses the 12 functions shown in Figure 11.4. The reason is that the `ioctl` function for terminal devices uses a different data type for its final argument, which depends on the action being performed. This makes type checking of the arguments impossible.

Although only 12 functions operate on terminal devices, realize that the first two functions in Figure 11.4, `tcgetattr` and `tcsetattr`, manipulate about 50 different flags (Figure 11.3). The large number of options available for terminal devices and trying to determine which options are required for a particular device (be it a terminal, modem, laser printer, or whatever) complicates the handling of terminal devices.

Field	Flag	Description	POSIX.1	SVR4	4.3+BSD extension
c_iflag	BRKINT	generate SIGINT on BREAK	•		
	ICRNL	map CR to NL on input	•		
	IGNBRK	ignore BREAK condition	•		
	IGNCR	ignore CR	•		
	IGNPAR	ignore characters with parity errors	•		
	IMAXBEL	ring bell on input queue full		•	•
	INLCR	map NL to CR on input	•		
	INPCK	enable input parity checking	•		
	ISTRIP	strip eighth bit off input characters	•		
	IUCLC	map uppercase to lowercase on input		•	
	IXANY	enable any characters to restart output		•	•
	IXOFF	enable start/stop input flow control	•		
	IXON	enable start/stop output flow control	•		
	PARMRK	mark parity errors	•		
c_oflag	BSDLY	backspace delay mask		•	
	CRDLY	CR delay mask		•	
	FFDLY	form feed delay mask		•	
	NLDLY	NL delay mask		•	
	OCRNL	map CR to NL on output		•	
	OFDEL	fill is DEL, else NUL		•	
	OFILL	use fill character for delay		•	
	OLCUC	map lowercase to uppercase on output		•	
	ONLCR	map NL to CR-NL (ala CRMOD)		•	•
	ONLRET	NL performs CR function		•	
	ONOCR	no CR output at column 0		•	
	ONOEOT	discard EOTs (^D) on output			•
	OPOST	perform output processing	•		
	OXTABS	expand tabs to spaces			•
	TABDLY	horizontal tab delay mask		•	
	VTDLY	vertical tab delay mask		•	
c_cflag	CCTS_OFLOW	CTS flow control of output			•
	CIGNORE	ignore control flags			•
	CLOCAL	ignore modem status lines	•		
	CREAD	enable receiver	•		
	CRTS_IFLOW	RTS flow control of input			•
	CSIZE	character size mask	•		
	CSTOPB	send two stop bits, else one	•		
	HUPCL	hangup on last close	•		
	MDMBUF	flow control output via Carrier			•
	PARENB	parity enable	•		
	PARODD	odd parity, else even	•		
c_lflag	ALTWERASE	use alternate WERASE algorithm			•
	ECHO	enable echo	•		
	ECHOCTL	echo control chars as ^(Char)		•	•
	ECHOE	visually erase chars	•		
	ECHOK	echo kill	•		
	ECHOKE	visual erase for kill		•	•
	ECHONL	echo NL	•		
	ECHOPRT	visual erase mode for hardcopy		•	•
	FLUSHO	output being flushed		•	•
	ICANON	canonical input	•		
	IEXTEN	enable extended input char processing	•		
	ISIG	enable terminal-generated signals	•		
	NOFLSH	disable flush after interrupt or quit	•		
	NOKERNINFO	no kernel output from STATUS			•
	PENDIN	retype pending input		•	•
	TOSTOP	send SIGTTOU for background output	•		
	XCASE	canonical upper/lower presentation		•	

Figure 11.3 Terminal flags.

Function	Description
tcgetattr	fetch attributes (termios structure)
tcsetattr	set attributes (termios structure)
cfgetispeed	get input speed
cfgetospeed	get output speed
cfsetispeed	set input speed
cfsetospeed	set output speed
tcdrain	wait for all output to be transmitted
tcflow	suspend transmit or receive
tcflush	flush pending input and/or output
tcsendbreak	send BREAK character
tcgetpgrp	get foreground process group ID
tcsetpgrp	set foreground process group ID

Figure 11.4 Summary of POSIX.1 terminal I/O functions.

The relationships among the 12 functions shown in Figure 11.4 are shown in Figure 11.5.

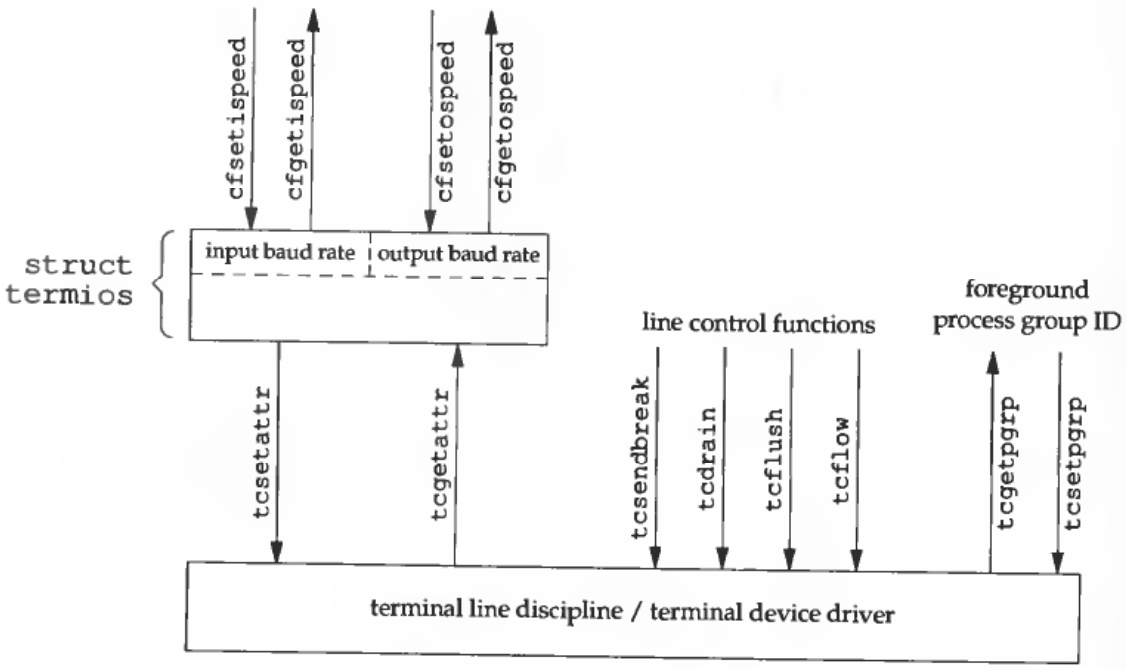


Figure 11.5 Relationship between the terminal-related functions.

POSIX.1 doesn't specify where in the termios structure the baud rate information is stored, that is an implementation feature. Many older systems stored this information in the c_cflag field. 4.3+BSD has two separate fields in the structure—one for the input speed and one for the output speed.

11.3 Special Input Characters

POSIX.1 defines 11 different characters that are handled specially on input. SVR4 adds another 6 special characters and 4.3+BSD adds 7. Figure 11.6 summarizes these special characters.

Character	Description	c_cc subscript	Enabled by field	Enabled by flag	Typical value	POSIX.1	SVR4	4.3+BSD
							extension	
CR	carriage return	(can't change)	c_lflag	ICANON	\r	•		
DISCARD	discard output	VDISCARD	c_lflag	IEXTEN	^O		•	•
DSUSP	delayed suspend (SIGTSTP)	VDSUSP	c_lflag	ISIG	^Y		•	•
EOF	end-of-file	VEOF	c_lflag	ICANON	^D	•		
EOL	end-of-line	VEOL	c_lflag	ICANON		•		
EOL2	alternate end-of-line	VEOL2	c_lflag	ICANON			•	•
ERASE	backspace one character	VERASE	c_lflag	ICANON	^H	•		
INTR	interrupt signal (SIGINT)	VINTR	c_lflag	ISIG	^?, ^C	•		
KILL	erase line	VKILL	c_lflag	ICANON	^U	•		
LNEXT	literal next	VLNEXT	c_lflag	IEXTEN	^V		•	•
NL	linefeed	(can't change)	c_lflag	ICANON	\n	•		
QUIT	quit signal (SIGQUIT)	VQUIT	c_lflag	ISIG	^\	•		
REPRINT	reprint all input	VREPRINT	c_lflag	ICANON	^R		•	•
START	resume output	VSTART	c_iflag	IXON/IXOFF	^S	•		
STATUS	status request	VSTATUS	c_lflag	ICANON	^T			•
STOP	stop output	VSTOP	c_iflag	IXON/IXOFF	^Q	•		
SUSP	suspend signal (SIGTSTP)	VSUSP	c_lflag	ISIG	^Z	•		
WERASE	backspace one word	VWERASE	c_lflag	ICANON	^W		•	•

Figure 11.6 Summary of special terminal input characters.

Of the 11 POSIX.1 special characters, we can change nine of them to almost any value that we like. The exceptions are the newline and carriage-return characters, (\n and \r respectively), and perhaps the STOP and START characters (depends on the implementation). To do this we modify the appropriate entry in the c_cc array of the termios structure. The elements in this array are referred to by name, with each name beginning with a V (the third column in Figure 11.6).

POSIX.1 optionally allows us to disable these characters. If _POSIX_VDISABLE is in effect, then the value of _POSIX_VDISABLE can be stored in the appropriate entry in the c_cc array to disable that special character. This feature can be queried with the pathconf and fpathconf functions (Section 2.5.4).

FIPS 151-1 requires support for _POSIX_VDISABLE.

SVR4 and 4.3+BSD also support this feature. SVR4 defines _POSIX_VDISABLE as 0, while 4.3+BSD defines it as octal 377.

Some earlier Unix systems disabled a feature if the corresponding special input character was 0.

Example

Before describing all the special characters in detail, let's look at a small program that changes them. Program 11.1 disables the interrupt character and sets the end-of-file character to Control-B.

```
#include    <termios.h>
#include    "ourhdr.h"

int
main(void)
{
    struct termios  term;
    long            vdisable;

    if (isatty(STDIN_FILENO) == 0)
        err_quit("standard input is not a terminal device");

    if ( (vdisable = fpathconf(STDIN_FILENO, _PC_VDISABLE)) < 0)
        err_quit("fpathconf error or _POSIX_VDISABLE not in effect");

    if (tcgetattr(STDIN_FILENO, &term) < 0) /* fetch tty state */
        err_sys("tcgetattr error");

    term.c_cc[VINTR] = vdisable;    /* disable INTR character */
    term.c_cc[VEOF]  = 2;           /* EOF is Control-B */

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &term) < 0)
        err_sys("tcsetattr error");

    exit(0);
}
```

Program 11.1 Disable interrupt character and change end-of-file character.

There are a few things to note in this program.

1. We modify the terminal characters only if standard input is a terminal device. We call `isatty` (Section 11.9) to check this.
2. We fetch the `_POSIX_VDISABLE` value using `fpathconf`.
3. The function `tcgetattr` (Section 11.4) fetches a `termios` structure from the kernel. After we've modified this structure we call `tcsetattr` to set the attributes. The only attributes that change are the ones we specifically modified.
4. Disabling the interrupt key is different from ignoring the interrupt signal. All Program 11.1 does is disable the special character that causes the terminal driver to generate `SIGINT`. We can still use the `kill` function to send the process the signal. □

We now describe each of the special characters in more detail. We call these the special input characters, but two of the characters, STOP and START (Control-S and Control-Q), are also handled specially when output. Note that most of these special characters, when they are recognized by the terminal driver and processed specially, are then discarded: they are not returned to the process in a read operation. The exceptions to this are the newline characters (NL, EOL, EOL2) and the carriage return (CR).

CR The POSIX.1 carriage return character. We cannot change this character. This character is recognized on input in canonical mode. When both ICANON (canonical mode) and ICRNL (map CR to NL) are set and IGNCR (ignore CR) is not set, the CR character is translated to NL and has the same effect as an NL character.

This character is returned to the reading process (perhaps after being translated to an NL).

DISCARD The SVR4 and 4.3+BSD discard character. This character is recognized on input in extended mode (IEXTEN). It causes subsequent output to be discarded, until another DISCARD character is entered or the discard condition is cleared (see the FLUSHO option). This character is discarded when processed (i.e., it is not passed to the process).

DSUSP The SVR4 and 4.3+BSD delayed-suspend job-control character. This character is recognized on input in extended mode (IEXTEN) if job control is supported and if the ISIG flag is set. Like the SUSP character, this delayed-suspend character generates the SIGTSTP signal that is sent to all processes in the foreground process group (refer to Figure 9.7). But the delayed-suspend character is sent to the process group only when a process reads from the controlling terminal, not when the character is typed. This character is discarded when processed (i.e., it is not passed to the process).

EOF The POSIX.1 end-of-file character. This character is recognized on input in canonical mode (ICANON). When we type this character, all bytes waiting to be read are immediately passed to the reading process. If there are no bytes waiting to be read, a count of 0 is returned. Entering an EOF character at the beginning of the line is the normal way to indicate an end-of-file to a program. This character is discarded when processed in canonical mode (i.e., it is not passed to the process).

EOL The POSIX.1 additional line delimiter character, like NL. This character is recognized on input in canonical mode (ICANON).

This character is not normally used. This character is returned to the reading process.

EOL2 The SVR4 and 4.3+BSD additional line delimiter character, like NL. This character is recognized on input in canonical mode (ICANON).

This character is not normally used. This character is returned to the reading process.

ERASE The POSIX.1 erase character (backspace). This character is recognized on input in canonical mode (ICANON). It erases the previous character in the line, not erasing beyond the beginning of the line. This character is discarded when processed in canonical mode (i.e., it is not passed to the process).

INTR The POSIX.1 interrupt character. This character is recognized on input if the ISIG flag is set. It generates the SIGINT signal that is sent to all processes in the foreground process group (refer to Figure 9.7). This character is discarded when processed (i.e., it is not passed to the process).

KILL The POSIX.1 kill character. (The name "kill" is once again a misnomer. This character should be called the line erase character.) It is recognized on input in canonical mode (ICANON). It erases the entire line. It is discarded when processed (i.e., it is not passed to the process).

LNEXT The SVR4 and 4.3+BSD literal-next character. This character is recognized on input in extended mode (IEXTEN). It causes any special meaning of the next character to be ignored. This works for all the special characters mentioned in this section. Using this we can type any character to a program. The LNEXT character is discarded when processed, but the next character entered is passed to the process.

NL The POSIX.1 newline character, which is also called the line delimiter. We cannot change this character. This character is recognized on input in canonical mode (ICANON).

This character is returned to the reading process.

QUIT The POSIX.1 quit character. This character is recognized on input if the ISIG flag is set. It generates the SIGQUIT signal that is sent to all processes in the foreground process group (refer to Figure 9.7). This character is discarded when processed (i.e., it is not passed to the process).

Recall from Figure 10.1 that the difference between INTR and QUIT is that the QUIT character not only terminates the process by default, but it also generates a core file.

REPRINT The SVR4 and 4.3+BSD reprint character. This character is recognized on input in extended, canonical mode (both IEXTEN and ICANON flags set). It causes all unread input to be output (reechoed). This character is discarded when processed (i.e., it is not passed to the process).

START The POSIX.1 start character. This character is recognized on input if the IXON flag is set, and it is automatically generated as output if the IXOFF flag is set. A received START character with IXON set causes stopped output (from a previously entered STOP character) to restart. In this case the START character is discarded when processed (i.e., it is not passed to the process).

When IXOFF is set, the terminal driver automatically generates a START character to resume input that it had previously stopped, when the new input will not overflow the input buffer.

STATUS The 4.3+BSD status-request character. This character is recognized on input in extended, canonical mode (both IEXTEN and ICANON flags set). It generates the SIGINFO signal that is sent to all processes in the foreground

process group (refer to Figure 9.7). Additionally, if the NOKERNINFO flag is not set, status information on the foreground process group is also displayed on the terminal. This character is discarded when processed (i.e., it is not passed to the process).

STOP The POSIX.1 stop character. This character is recognized on input if the IXON flag is set, and it is automatically generated as output if the IXOFF flag is set. A received STOP character with IXON set stops the output. In this case the STOP character is discarded when processed (i.e., it is not passed to the process). The stopped output is restarted when a START character is entered.

When IXOFF is set, the terminal driver automatically generates a STOP character to prevent the input buffer from overflowing.

SUSP The POSIX.1 suspend job-control character. This character is recognized on input if job control is supported and if the ISIG flag is set. It generates the SIGTSTP signal that is sent to all processes in the foreground process group (refer to Figure 9.7). This character is discarded when processed (i.e., it is not passed to the process).

WERASE The SVR4 and 4.3+BSD word erase character. This character is recognized on input in extended, canonical mode (both IEXTEN and ICANON flags set). It causes the previous word to be erased. First it skips backward over any whitespace (spaces or tabs), then backward over the previous token, leaving the cursor positioned where the first character of the previous token was located. Normally the previous token ends when a whitespace character is encountered. We can change this, however, by setting the ALTWERASE flag. This flag causes the previous token to end when the first nonalphanumeric character is encountered. The word erase character is discarded when processed (i.e., it is not passed to the process).

Another "character" that we need to define for terminal devices is the BREAK character. BREAK is not really a character, but a condition that occurs during asynchronous serial data transmission. A BREAK condition is signaled to the device driver in various ways, depending on the serial interface. Most terminals have a key labeled BREAK that generates the BREAK condition, which is why most people think of BREAK as a character. For asynchronous serial data transmission, a BREAK is a sequence of zero-valued bits that continues for longer than the time required to send one byte. The entire sequence of zero-valued bits is considered a single BREAK. In Section 11.8 we'll see how to send a BREAK.

11.4 Getting and Setting Terminal Attributes

We call two functions to get and set a termios structure: tcgetattr and tcsetattr. This is how we examine and modify the various option flags and special characters to make the terminal operate the way we want it to.

```
#include <termios.h>

int tcgetattr(int filedes, struct termios *termptr);

int tcsetattr(int filedes, int opt, const struct termios *termptr);
```

Both return: 0 if OK, –1 on error

Both functions take a pointer to a `termios` structure and either return the current terminal attributes or set the terminal's attributes. Since these two functions operate only on terminal devices, if *filedes* does not refer to a terminal device, an error is returned and `errno` is set to `ENOTTY`.

The argument *opt* for `tcsetattr` lets us specify when we want the new terminal attributes to take effect. *opt* is specified as one of the following constants:

`TCSANOW`	The change occurs immediately.
`TCSADRAIN`	The change occurs after all output has been transmitted. This option should be used if we are changing the output parameters.
`TCSAFLUSH`	The change occurs after all output has been transmitted. Furthermore, when the change takes place, all input data that has not been read is discarded (flushed).

The return status of `tcsetattr` confuses the programming. This function returns OK if it was able to perform *any* of the requested actions, even if it couldn't perform all the requested actions. If the function returns OK it is our responsibility to see if all the requested actions were performed. This means that after we call `tcsetattr` to set the desired attributes, we need to call `tcgetattr` and compare the actual terminal's attributes with the desired attributes to detect any differences.

11.5 Terminal Option Flags

In this section we list all the various terminal option flags, expanding the descriptions of all the options from Figure 11.3. This list is alphabetical, and indicates in which of the four terminal flag fields the option appears. (The field a given option is controlled by is usually not apparent from just the option name.) We also list whether each option is POSIX defined or supported by either SVR4 or 4.3+BSD.

All the flags listed specify one or more bits that we turn on or clear, unless we call the flag a *mask*. A mask defines multiple bits. We have a defined name for the mask, and a name for each value. For example, to set the character size, we first zero the bits using the character-size mask `CSIZE`, and then set one of the values `CS5`, `CS6`, `CS7`, or `CS8`.

The six delay values supported by SVR4 are also masks: `BSDLY`, `CRDLY`, `FFDLY`, `NLDLY`, `TABDLY`, and `VTDLY`. Refer to the `termio`(7) manual page in AT&T [1991] for the length of each delay value. In all cases a delay mask of 0 means no delay. If a delay

is specified, the OFILL and OFDEL flags determine if the driver does an actual delay or if fill characters are transmitted instead.

Example

Program 11.2 demonstrates the use of these masks to extract a value and to set a value.

```
#include    <termios.h>
#include    "ourhdr.h"

int
main(void)
{
    struct termios  term;
    int             size;

    if (tcgetattr(STDIN_FILENO, &term) < 0)
        err_sys("tcgetattr error");

    size = term.c_cflag & CSIZE;
    if      (size == CS5)   printf("5 bits/byte\n");
    else if (size == CS6)   printf("6 bits/byte\n");
    else if (size == CS7)   printf("7 bits/byte\n");
    else if (size == CS8)   printf("8 bits/byte\n");
    else                    printf("unknown bits/byte\n");

    term.c_cflag &= ~CSIZE;      /* zero out the bits */
    term.c_cflag |= CS8;         /* set 8 bits/byte */

    if (tcsetattr(STDIN_FILENO, TCSANOW, &term) < 0)
        err_sys("tcsetattr error");

    exit(0);
}
```

Program 11.2 Example of `tcgetattr`.

□

We now describe each of the flags.

`ALTWERASE` (`c_lflag`, 4.3+BSD) When set, an alternate word erase algorithm is used when the WERASE character is entered. Instead of moving backward until the previous whitespace character, this flag causes the WERASE character to move backward until the first nonalphanumeric character is encountered.

`BRKINT` (`c_iflag`, POSIX.1) If this flag is set and `IGNBRK` is not set, when a BREAK is received the input and output queues are flushed and a `SIGINT` signal is generated. This signal is generated for the foreground process group if the terminal device is a controlling terminal.

If neither IGNBRK nor BRKINT is set, then a BREAK is read as a single character \0, unless PARMRK is set, in which case the BREAK is read as the three byte sequence \377, \0, \0.

BSDLY (c_oflag, SVR4) Backspace delay mask. The values for the mask are BS0 or BS1.

CCTS_OFLOW (c_cflag, 4.3+BSD) CTS flow control of output. (See Exercise 11.4.)

CIGNORE (c_cflag, 4.3+BSD) Ignore control flags.

CLOCAL (c_cflag, POSIX.1) If set, the modem status lines are ignored. This usually means that the device is locally attached. When this flag is not set, an open of a terminal device usually blocks until the modem is answered, for example.

CRDLY (c_oflag, SVR4) Carriage return delay mask. The values for the mask are CR0, CR1, CR2, or CR3.

CREAD (c_cflag, POSIX.1) If set, the receiver is enabled and characters can be received.

CRTS_IFLOW (c_cflag, 4.3+BSD) RTS flow control of input. (See Exercise 11.4.)

CSIZE (c_cflag, POSIX.1) This field is a mask that specifies the number of bits per byte for both transmission and reception. This size does not include the parity bit, if any. The values for the field defined by this mask are CS5, CS6, CS7, and CS8, for 5, 6, 7, and 8 bits per byte, respectively.

CSTOPB (c_cflag, POSIX.1) If set, two stop bits are used, otherwise one stop bit is used.

ECHO (c_lflag, POSIX.1) If set, input characters are echoed back to the terminal device. Input characters can be echoed in either canonical or noncanonical mode.

ECHOCTL (c_lflag, SVR4 and 4.3+BSD) If set and if ECHO is set, ASCII control characters (those character in the range 0 through octal 37, inclusive) other than the ASCII TAB, the ASCII NL, and the START and STOP characters, are echoed as ^X, where *X* is the character formed by adding octal 100 to the control character. This means the ASCII Control-A character (octal 1) is echoed as ^A. Also, the ASCII DELETE character (octal 177) is echoed as ^?. If this flag is not set, the ASCII control characters are echoed as themselves. As with the ECHO flag, this flag affects the echoing of control characters in both canonical and noncanonical modes.

Be aware that some systems echo the EOF character differently, since its typical value is Control-D. (Control-D is the ASCII EOT character, which can cause some terminals to hangup.) Check your manual.

ECHOE (c_lflag, POSIX.1) If set and if ICANON is set, the ERASE character erases the last character in the current line from the display. This is

usually done in the terminal driver by writing the three-character sequence: backspace, space, backspace.

If the WERASE character is supported, ECHOE causes the previous word to be erased using one or more of the same three-character sequence.

If the ECHOPRT flag is supported, the actions described here for ECHOE assume that the ECHOPRT flag is not set.

ECHOK (c_lflag, POSIX.1) If set and if ICANON is set, the KILL character erases the current line from the display or outputs the NL character (to emphasize that the entire line was erased).

If the ECHOKE flag is supported, this description of ECHOK assumes that ECHOKE is not set.

ECHOKE (c_lflag, SVR4 and 4.3+BSD) If set and if ICANON is set, the KILL character is echoed by erasing each character on the line. The way in which each character is erased is selected by the ECHOE and ECHOPRT flags.

ECHONL (c_lflag, POSIX.1) If set and if ICANON is set, the NL character is echoed, even if ECHO is not set.

ECHOPRT (c_lflag, SVR4 and 4.3+BSD) If set and if both ICANON and IECHO are set, then the ERASE character (and WERASE character, if supported) cause all the characters being erased to be printed as they are erased. This is often useful on a hardcopy terminal to see exactly which characters are being deleted.

FFDLY (c_oflag, SVR4) Form feed delay mask. The values for the mask are FF0 or FF1.

FLUSHO (c_lflag, SVR4 and 4.3+BSD) If set, output is being flushed. This flag is set when we type the DISCARD character, and it is cleared when we type another DISCARD character. We can also set or clear this condition by setting or clearing this terminal flag.

HUPCL (c_cflag, POSIX.1) If set, when the last process closes the device, the modem control lines are lowered (i.e., the modem connection is broken).

ICANON (c_lflag, POSIX.1) If set, canonical mode is in effect (Section 11.10). This enables the following characters: EOF, EOL, EOL2, ERASE, KILL, REPRINT, STATUS, and WERASE. The input characters are assembled into lines.

If canonical mode is not enabled, read requests are satisfied directly from the input queue. A read does not return until at least MIN bytes have been received or the time-out value TIME has expired between bytes. Refer to Section 11.11 for additional details.

ICRNL (c_iflag, POSIX.1) If set and if IGNCR is not set, a received CR character is translated into an NL character.

IEXTEN (c_lflag, POSIX.1) If set, the extended, implementation-defined special characters are recognized and processed.

IGNBRK (c_iflag, POSIX.1) When set, a BREAK condition on input is ignored. See BRKINT for a way to have a BREAK condition either generate a SIGINT signal, or be read as data.

IGNCR (c_iflag, POSIX.1) If set, a received CR character is ignored. If this flag is not set, it is possible to translate the received CR into an NL character if the ICRNL flag is set.

IGNPAR (c_iflag, POSIX.1) When set, an input byte with a framing error (other than a BREAK) or an input byte with a parity error is ignored.

IMAXBEL (c_iflag, SVR4 and 4.3+BSD) Ring bell when input queue is full.

INLCR (c_iflag, POSIX.1) If set, a received NL character is translated into a CR character.

INPCK (c_iflag, POSIX.1) When set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled.

Parity "generation and detection" and "input parity checking" are two different things. The generation and detection of parity bits is controlled by the PARENB flag. Setting this flag usually causes the device driver for the serial interface to generate parity for outgoing characters and to verify the parity of incoming characters. The flag PARODD determines if the parity should be odd or even. If an input character arrives with the wrong parity, then the state of the INPCK flag is checked. If this flag is set, then the IGNPAR flag is checked (to see if the input byte with the parity error should be ignored); and if the byte should not be ignored, then the PARMRK flag is checked to see what characters should be passed to the reading process.

ISIG (c_lflag, POSIX.1) If set, the input characters are compared against the special characters that cause the terminal-generated signals to be generated (INTR, QUIT, SUSP, and DSUSP), and if equal, the corresponding signal is generated.

ISTRIP (c_iflag, POSIX.1) When set, valid input bytes are stripped to seven bits. When this flag is not set, all eight bits are processed.

IUCLC (c_iflag, SVR4) Map uppercase to lowercase on input.

IXANY (c_iflag, SVR4 and 4.3+BSD) Enable any characters to restart output.

IXOFF (c_iflag, POSIX.1) If set, start–stop input control is enabled. When the terminal driver notices that the input queue is getting full, it outputs a STOP character. This character should be recognized by the device that is sending the data and cause the device to stop. Later, when the characters on the input queue have been processed, the terminal driver will output a START character. This should cause the device to resume sending data.

IXON	(c_iflag, POSIX.1) If set, start–stop output control is enabled. When the terminal driver receives a STOP character, output stops. While the output is stopped, the next START character resumes the output. If this flag is not set, the START and STOP characters are read by the process as normal characters.
MDMBUF	(c_cflag, 4.3+BSD) Flow control the output according to the modem carrier flag.
NLDLY	(c_oflag, SVR4) Newline delay mask. The values for the mask are NL0 or NL1.
NOFLSH	(c_lflag, POSIX.1) By default, when the terminal driver generates the SIGINT and SIGQUIT signals, both the input and output queues are flushed. Also, when it generates the SIGSUSP signal, the input queue is flushed. If the NOFLSH flag is set, this normal flushing of the queues does not occur when these signals are generated.
NOKERNINFO	(c_lflag, 4.3+BSD) When set, this flag prevents the STATUS character from printing information on the foreground process group. Regardless of this flag, however, the STATUS character still causes the SIGINFO signal to be sent to the foreground process group.
OCRNL	(c_oflag, SVR4) If set, map CR to NL on output.
OFDEL	(c_oflag, SVR4) If set, the output fill character is ASCII DEL, otherwise it's ASCII NUL. See the OFILL flag.
OFILL	(c_oflag, SVR4) If set, fill characters (either ASCII DEL or ASCII NUL, see the OFDEL flag) are transmitted for a delay, instead of using a timed delay. See the six delay masks: BSDLY, CRDLY, FFDLY, NLDLY, TABDLY, and VTDLY.
OLCUC	(c_oflag, SVR4) If set, map lowercase to uppercase on output.
ONLCR	(c_oflag, SVR4 and 4.3+BSD) If set, map NL to CR-NL on output.
ONLRET	(c_oflag, SVR4) If set, the NL character is assumed to perform the carriage-return function on output.
ONOCR	(c_oflag, SVR4) If set, a CR is not output at column 0.
ONOEOT	(c_oflag, 4.3+BSD) If set, EOT characters (^D) are discarded on output. This may be necessary on some terminals that interpret the Control-D as a hangup.
OPOST	(c_oflag, POSIX.1) If set, implementation-defined output processing takes place. Refer to Figure 11.3 for the various implementation-defined flags for the c_oflag word.
OXTABS	(c_oflag, 4.3+BSD) If set, tabs are expanded to spaces on output. This produces the same effect as setting the horizontal tab delay (TABDLY) to XTABS or TAB3.
PARENB	(c_cflag, POSIX.1) If set, parity generation is enabled for outgoing characters, and parity checking is performed on incoming characters.

The parity is odd if PARODD is set, otherwise it is even parity. See also the discussion of the INPCK, IGNPAR, and PARMRK flags.

PARMRK (c_iflag, POSIX.1) When set and if IGNPAR is not set, a byte with a framing error (other than a BREAK) or a byte with a parity error, is read by the process as the three character sequence \377, \0, X, where X is the byte received in error. If ISTRIP is not set, a valid \377 is passed to the process as \377, \377. If neither IGNPAR nor PARMRK is set, a byte with a framing error (other than a BREAK) or a byte with a parity error is read as a single character \0.

PARODD (c_cflag, POSIX.1) If set, the parity for outgoing and incoming characters is odd parity. Otherwise, the parity is even parity. Note that the PARENB flag controls the generation and detection of parity.

PENDIN (c_lflag, SVR4 and 4.3+BSD) If set, any input that has not been read is reprinted by the system when the next character is input. This action is similar to what happens when we type the REPRINT character.

TABDLY (c_oflag, SVR4) Horizontal tab delay mask. The values for the mask are TAB0, TAB1, TAB2, or TAB3.

The value XTABS is equal to TAB3. This value causes the system to expand tabs into spaces. The system assumes a tab stop every eight spaces, and we can't change this assumption.

TOSTOP (c_lflag, POSIX.1) If set and if the implementation supports job control, the SIGTTOU signal is sent to the process group of a background process that tries to write to its controlling terminal. By default, this signal stops all the processes in the process group. This signal is not generated by the terminal driver if the background process that is writing to the controlling terminal is either ignoring or blocking the signal.

VTDLY (c_oflag, SVR4) Vertical tab delay mask. The values for the mask are VT0 or VT1.

XCASE (c_lflag, SVR4) If set and if ICANON is also set, the terminal is assumed to be uppercase only, and all input is converted to lowercase. To input an uppercase character, precede it with a backslash. Similarly, an uppercase character is output by the system by being preceded by a backslash. (This option flag is obsolete today, since most, if not all, uppercase only terminals have disappeared.)

11.6 stty Command

All the options described in the previous section can be examined and changed from within a program, with the tcgetattr and tcsetattr functions (Section 11.4), or from the command line (or a shell script), with the stty(1) command. This command is just an interface to the first six functions that we listed in Figure 11.4. If we execute this command with its -a option, it displays all the terminal options.

```
$ stty -a
speed 9600 baud; 34 rows; 80 columns;
lflags: icanon isig iexten echo echoe echok echoke -echonl echoctl
        -echoprt -altwerase -noflsh -tostop -mdmbuf -flusho -pendin
        -nokerninfo -extproc
iflags: istrip icrnl -inlcr -igncr ixon -ixoff ixany imaxbel -ignbrk
        brkint -inpck -ignpar -parmrk
oflags: opost onlcr -oxtabs
cflags: cread cs7 parenb -parodd hupcl -clocal -cstopb -crtscts
cchars: discard = ^O; dsusp = ^Y; eof = ^D; eol = <undef>;
        eol2 = <undef>; erase = ^H; intr = ^?; kill = ^U; lnext = ^V;
        quit = ^\; reprint = ^R; start = ^Q; status = ^T; stop = ^S;
        susp = ^Z; werase = ^W;
```

Option names preceded by a hyphen are disabled. The last four lines display the current settings for each of the terminal special characters (Section 11.3). The first line displays the number of rows and columns for the current terminal window—we discuss this in Section 11.12.

Since the `stty` command is a user command and not an operating system function, it is specified by POSIX.2, not POSIX.1.

> Historically, System V versions of `stty` operate on the standard input and write any output to standard output. Version 7 and BSD systems operate on the standard output and write any output to standard error. The current draft of POSIX.2 follows the System V convention, as does the 4.3+BSD version.
>
> The Version 7 manual page for `stty` required a single page, while the SVR4 version requires over six pages. Terminal drivers tend to acquire more and more options over time.

11.7 Baud Rate Functions

The term baud rate is a historical term that should be referred to today as "bits per second." Although most terminal devices use the same baud rate for both input and output, the capability exists to set the two to different values, if the hardware allows this.

```
#include <termios.h>

speed_t cfgetispeed(const struct termios *termptr);

speed_t cfgetospeed(const struct termios *termptr);
```

Both return: baud rate value

```
int cfsetispeed(struct termios *termptr, speed_t speed);

int cfsetospeed(struct termios *termptr, speed_t speed);
```

Both return: 0 if OK, –1 on error

The return value from the two `cfget` functions and the *speed* argument to the two `cfset` functions are one of the following constants: `B50`, `B75`, `B110`, `B134`, `B150`, `B200`, `B300`, `B600`, `B1200`, `B1800`, `B2400`, `B4800`, `B9600`, `B19200`, or `B38400`. The constant `B0` means "hangup." When `B0` is specified as the output baud rate when `tcsetattr` is called, the modem control lines are no longer asserted.

To use these functions we must realize that the input and output baud rates are stored in the device's `termios` structure, as shown in Figure 11.5. Before calling either of the `cfget` functions we first have to obtain the device's `termios` structure using `tcgetattr`. Similarly, after calling either of the two `cfset` functions, all we've done is set the baud rate in a `termios` structure. For this change to affect the device we have to call `tcsetattr`.

If there is an error in either of the baud rates that we set, we may not find out about the error until we call `tcsetattr`.

11.8 Line Control Functions

The following four functions provide line control capability for terminal devices. All four require that *filedes* refer to a terminal device, otherwise an error is returned with `errno` set to `ENOTTY`.

```
#include <termios.h>

int tcdrain(int filedes);

int tcflow(int filedes, int action);

int tcflush(int filedes, int queue);

int tcsendbreak(int filedes, int duration);
```

All four return: 0 if OK, –1 on error

The `tcdrain` function waits for all output to be transmitted. `tcflow` gives us control over both input and output flow control. The *action* argument must be one of the following four values:

`TCOOFF`	Output is suspended.
`TCOON`	Output that was previously suspended is restarted.
`TCIOFF`	The system transmits a STOP character. This should cause the terminal device to stop sending data.
`TCION`	The system transmits a START character. This should cause the terminal device to resume sending data.

The `tcflush` function lets us flush (throw away) either the input buffer (data that has been received by the terminal driver, which we have not read) or the output buffer (data that we have written, which has not yet been transmitted). The *queue* argument must be one of the following three constants:

`TCIFLUSH`	The input queue is flushed.
`TCOFLUSH`	The output queue is flushed.
`TCIOFLUSH`	Both the input and output queues are flushed.

The `tcsendbreak` function transmits a continuous stream of zero bits for a specified duration. If the *duration* argument is 0, the transmission lasts between 0.25 and 0.5 seconds. POSIX.1 specifies that if *duration* is nonzero, the transmission time is implementation dependent.

> The SVR4 SVID states that if *duration* is nonzero, no zero bits are transmitted. The SVR4 manual page, however, states that if *duration* is nonzero, then `tcsendbreak` behaves like `tcdrain`. Yet another system's manual page states that if *duration* is nonzero, the time that the zero bits are transmitted is *duration*×*N*, where *N* is between 0.25 and 0.5 seconds. Clearly there is little agreement on how to handle this condition.

11.9 Terminal Identification

Historically, the name of the controlling terminal in most versions of Unix has been `/dev/tty`. POSIX.1 provides a run-time function that we can call to determine the name of the controlling terminal.

```
#include <stdio.h>

char *ctermid(char *ptr);
```

Returns: (see following)

If *ptr* is nonnull, it is assumed to point to an array of at least `L_ctermid` bytes and the name of the controlling terminal of the process is stored in the array. The constant `L_ctermid` is defined in `<stdio.h>`. If *ptr* is a null pointer, the function allocates room for the array (usually as a static variable). Again, the name of the controlling terminal of the process is stored in the array.

In both cases, the starting address of the array is returned as the value of the function. Since most Unix systems use `/dev/tty` as the name of the controlling terminal, this function is intended to aid portability to other operating systems.

Example—`ctermid` Function

Program 11.3 is an implementation of the POSIX.1 `ctermid` function. □

```
#include    <stdio.h>
#include    <string.h>

static char ctermid_name[L_ctermid];

char *
ctermid(char *str)
{
    if (str == NULL)
        str = ctermid_name;
    return(strcpy(str, "/dev/tty"));    /* strcpy() returns str */
}
```

Program 11.3 Implementation of POSIX.1 `termid` function.

Two functions that are more interesting for a Unix system are `isatty`, which returns true if a file descriptor refers to a terminal device, and `ttyname`, which returns the pathname of the terminal device that is open on a file descriptor.

```
#include <unistd.h>

int isatty(int filedes);

                    Returns: 1 (true) if terminal device, 0 (false) otherwise

char *ttyname(int filedes);

                    Returns: pointer to pathname of terminal, NULL on error
```

Example—`isatty` Function

The `isatty` function is trivial to implement as we show in Program 11.4. We just try one of the terminal-specific functions (that doesn't change anything if it succeeds) and look at the return value.

```
#include    <termios.h>

int
isatty(int fd)
{
    struct termios  term;

    return(tcgetattr(fd, &term) != -1); /* true if no error (is a tty) */
}
```

Program 11.4 Implementation of POSIX.1 `isatty` function.

```
#include    "ourhdr.h"

int
main(void)
{
    printf("fd 0: %s\n", isatty(0) ? "tty" : "not a tty");
    printf("fd 1: %s\n", isatty(1) ? "tty" : "not a tty");
    printf("fd 2: %s\n", isatty(2) ? "tty" : "not a tty");
    exit(0);
}
```

Program 11.5 Test the `isatty` function.

We test our `isatty` function with Program 11.5, giving us

```
$ a.out
fd 0: tty
fd 1: tty
fd 2: tty
$ a.out </etc/passwd 2>/dev/null
fd 0: not a tty
fd 1: tty
fd 2: not a tty
```

□

Example—`ttyname` Function

The `ttyname` function (Program 11.6) is longer, as we have to search all the device entries, looking for a match. The technique is to read the `/dev` directory, looking for an entry with the same device number and i-node number. Recall from Section 4.23 that each filesystem has a unique device number (the `st_dev` field in the `stat` structure, from Section 4.2), and each directory entry in that filesystem has a unique i-node number (the `st_ino` field in the `stat` structure). We assume in this function that when we hit a matching device number and matching i-node number, we've located the desired directory entry. We could also verify that the two entries have matching `st_rdev` fields (the major and minor device numbers for the terminal device) and that the directory entry is also a character special file. But since we've already verified that the file descriptor argument is both a terminal device and a character special file, and since a matching device number and i-node number is unique on a Unix system, there is no need for the additional comparisons.

We can test this implementation with Program 11.7. Running Program 11.7 gives us

```
$ a.out < /dev/console 2> /dev/null
fd 0: /dev/console
fd 1: /dev/ttyp3
fd 2: not a tty
```

□

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <dirent.h>
#include    <limits.h>
#include    <string.h>
#include    <termios.h>
#include    <unistd.h>

#define DEV     "/dev/"         /* device directory */
#define DEVLEN  sizeof(DEV)-1   /* sizeof includes null at end */

char *
ttyname(int fd)
{
    struct stat     fdstat, devstat;
    DIR             *dp;
    struct dirent   *dirp;
    static char     pathname[_POSIX_PATH_MAX + 1];
    char            *rval;

    if (isatty(fd) == 0)
        return(NULL);
    if (fstat(fd, &fdstat) < 0)
        return(NULL);
    if (S_ISCHR(fdstat.st_mode) == 0)
        return(NULL);

    strcpy(pathname, DEV);
    if ( (dp = opendir(DEV)) == NULL)
        return(NULL);
    rval = NULL;
    while ( (dirp = readdir(dp)) != NULL) {
        if (dirp->d_ino != fdstat.st_ino)
            continue;       /* fast test to skip most entries */

        strncpy(pathname + DEVLEN, dirp->d_name, _POSIX_PATH_MAX - DEVLEN);
        if (stat(pathname, &devstat) < 0)
            continue;
        if (devstat.st_ino == fdstat.st_ino &&
            devstat.st_dev == fdstat.st_dev) {  /* found a match */
                rval = pathname;
                break;
        }
    }
    closedir(dp);
    return(rval);
}
```

Program 11.6 Implementation of POSIX.1 `ttyname` function.

```
#include    "ourhdr.h"

int
main(void)
{
    printf("fd 0: %s\n", isatty(0) ? ttyname(0) : "not a tty");
    printf("fd 1: %s\n", isatty(1) ? ttyname(1) : "not a tty");
    printf("fd 2: %s\n", isatty(2) ? ttyname(2) : "not a tty");
    exit(0);
}
```

Program 11.7 Test the `ttyname` function.

11.10 Canonical Mode

Canonical mode is simple—we issue a read and the terminal driver returns when a line has been entered. Several conditions cause the read to return:

- The read returns when the requested number of bytes has been read. We don't have to read a complete line. If we read a partial line, no information is lost—the next read starts where the previous read stopped.
- The read returns when a line delimiter is encountered. Recall from Section 11.3 that the following characters are interpreted as "end-of-line" in canonical mode: NL, EOL, EOL2, and EOF. Also, from Section 11.5 recall that if `ICRNL` is set, and if `IGNCR` is not set, then the CR character also terminates a line since it acts just like the NL character.

  Realize that of these five line delimiters, one (EOF) is discarded by the terminal driver when it's processed. The other four are returned to the caller as the last character of the line.
- The read also returns if a signal is caught and if the function is not automatically restarted (Section 10.5).

Example—`getpass` Function

We now show the function `getpass` that reads a password of some type from the user at a terminal. This function is called by the Unix `login`(1) and `crypt`(1) programs. To read the password it must turn off echoing, but it can leave the terminal in canonical mode, as whatever we type as the password forms a complete line. Program 11.8 shows a typical Unix implementation.

```
#include    <signal.h>
#include    <stdio.h>
#include    <termios.h>

#define MAX_PASS_LEN    8       /* max #chars for user to enter */

char *
getpass(const char *prompt)
{
    static char     buf[MAX_PASS_LEN + 1];  /* null byte at end */
    char            *ptr;
    sigset_t        sig, sigsave;
    struct termios  term, termsave;
    FILE            *fp;
    int             c;

    if ( (fp = fopen(ctermid(NULL), "r+")) == NULL)
        return(NULL);
    setbuf(fp, NULL);

    sigemptyset(&sig);  /* block SIGINT & SIGTSTP, save signal mask */
    sigaddset(&sig, SIGINT);
    sigaddset(&sig, SIGTSTP);
    sigprocmask(SIG_BLOCK, &sig, &sigsave);

    tcgetattr(fileno(fp), &termsave);   /* save tty state */
    term = termsave;                    /* structure copy */
    term.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);
    tcsetattr(fileno(fp), TCSAFLUSH, &term);

    fputs(prompt, fp);

    ptr = buf;
    while ( (c = getc(fp)) != EOF && c != '\n') {
        if (ptr < &buf[MAX_PASS_LEN])
            *ptr++ = c;
    }
    *ptr = 0;           /* null terminate */
    putc('\n', fp);     /* we echo a newline */

                        /* restore tty state */
    tcsetattr(fileno(fp), TCSAFLUSH, &termsave);

                        /* restore signal mask */
    sigprocmask(SIG_SETMASK, &sigsave, NULL);
    fclose(fp);         /* done with /dev/tty */

    return(buf);
}
```

Program 11.8 Implementation of getpass function.

There are several points to consider in this example.

- We call the function `ctermid` to open the controlling terminal, instead of hard-wiring `/dev/tty` into the program.
- We read and write only to the controlling terminal and return an error if we can't open this device for reading and writing. There are other conventions to use. The 4.3+BSD version of `getpass` reads from standard input and writes to standard error if the controlling terminal can't be opened for reading and writing. The SVR4 version always writes to standard error but only reads from the controlling terminal.
- We block the two signals `SIGINT` and `SIGTSTP`. If we didn't do this, entering the INTR character would abort the program and leave the terminal with echoing disabled. Similarly, entering the SUSP character would stop the program and return to the shell with echoing disabled. We choose to block the signals while we have echoing disabled. If they are generated while we're reading the password, they are held until we return. There are other ways to handle these signals. Some versions just ignore `SIGINT` (saving its previous action) while in `getpass`, resetting the action for this signal to its previous value before returning. This means that any occurrence of the signal while it's ignored is lost. Other versions catch `SIGINT` (saving its previous action) and if the signal is caught, then after resetting the terminal state and signal action, just send themselves the signal with the `kill` function. None of the versions of `getpass` catch, ignore, or block `SIGQUIT`, so entering the QUIT character aborts the program and probably leaves the terminal with echoing disabled.
- Be aware that some shells, notably the KornShell, turn echoing back on whenever they read interactive input. These shells are the ones that provide command-line editing and therefore manipulate the state of the terminal every time we enter an interactive command. So, if we invoke this program under one of these shells and abort it with the QUIT character, it may reenable echoing for us. Other shells that don't provide this form of command-line editing, such as the Bourne shell and C shell, will abort the program and leave the terminal in a noecho mode. If we do this to our terminal, the `stty` command can reenable echoing.
- We use standard I/O to read and write the controlling terminal. We specifically set the stream to be unbuffered, otherwise there might be some interactions between the writing and reading of the stream (we would need some calls to `fflush`). We could have also used unbuffered I/O (Chapter 3), but we would have to simulate the `getc` function using `read`.
- We store only up to eight characters as the password. Any additional characters that are entered are just ignored.

Program 11.9 calls `getpass` and prints what we entered. This is just to let us verify that the ERASE and KILL characters work (as they should in canonical mode).

```
#include    "ourhdr.h"

char    *getpass(const char *);

int
main(void)
{
    char    *ptr;

    if ( (ptr = getpass("Enter password:")) == NULL)
        err_sys("getpass error");
    printf("password: %s\n", ptr);

    /* now use password (probably encrypt it) ... */

    while (*ptr != 0)
        *ptr++ = 0;     /* zero it out when we're done with it */

    exit(0);
}
```

Program 11.9 Call the `getpass` function.

Whenever a program that calls `getpass` is done with the cleartext password, it should zero it out in memory, just to be safe. If the program were to generate a `core` file that others could read (recall from Section 10.2 that the default permissions on a `core` file allow everyone to read it), or if some other process were somehow able to read our memory, they might be able to read the cleartext password. (By "cleartext" we mean the password that we type at the prompt that is printed by `getpass`. Most Unix programs then modify this cleartext password into an "encrypted" password. The field `pw_passwd` in the password file, for example, contains the encrypted password, not the cleartext password.) □

11.11 Noncanonical Mode

Noncanonical mode is specified by turning off the `ICANON` flag in the `c_lflag` field of the `termios` structure. In noncanonical mode the input data is not assembled into lines. The following special characters (Section 11.3) are not processed: ERASE, KILL, EOF, NL, EOL, EOL2, CR, REPRINT, STATUS, and WERASE.

As we said, canonical mode is easy—the system returns up to one line at a time. But with noncanonical mode, how does the system know when to return data to us? If it returned one byte at a time, there would be excessive overhead. (Recall Figure 3.1 where we saw how much overhead there was in reading one byte at a time. Each time we doubled the amount of data returned we halved the system call overhead.) The system can't always return multiple bytes at a time, since sometimes we don't know how much data to read, until we start reading it.

The solution is to tell the system to return when either a specified amount of data has been read or after a given amount of time has passed. This technique uses two

variables in the `c_cc` array in the `termios` structure: MIN and TIME. These two elements of the array are indexed by the names `VMIN` and `VTIME`.

MIN specifies the minimum number of bytes before a `read` returns. TIME specifies the number of tenths-of-a-second to wait for data to arrive. There are four cases.

Case A: MIN > 0, TIME > 0

TIME specifies an interbyte timer that is started only when the first byte is received. If MIN bytes are received before the timer expires, `read` returns MIN bytes. If the timer expires before MIN bytes are received, `read` returns the bytes received. (At least one byte is returned if the timer expires, because the timer is not started until the first byte is received.) In this case the caller blocks until the first byte is received. If data is already available when `read` is called, it is as if the data had been received immediately after the `read`.

Case B: MIN > 0, TIME == 0

The `read` does not return until MIN bytes have been received. This can cause a `read` to block indefinitely.

Case C: MIN == 0, TIME > 0

TIME specifies a read timer that is started when `read` is called. (Compare this to case A, where a nonzero TIME represented an interbyte timer that was not started until the first byte was received.) `read` returns when a single byte is received or when the timer expires. If the timer expires, `read` returns 0.

Case D: MIN == 0, TIME == 0

If some data is available, `read` returns up to the number of bytes requested. If no data is available, `read` returns 0 immediately.

Realize in all these cases that MIN is only a minimum. If the program requests more than MIN bytes of data, it's possible to receive up to the requested amount. This also applies to cases C and D where MIN is 0.

Figure 11.7 summarizes the four different cases for noncanonical input. In this figure *nbytes* is the third argument to `read` (the maximum number of bytes to return).

	MIN > 0	MIN == 0
TIME > 0	**A:** `read` returns [MIN, *nbytes*] before timer expires; `read` returns [1, MIN) if timer expires. (TIME = interbyte timer. Caller can block indefinitely.)	**C:** `read` returns [1, *nbytes*] before timer expires; `read` returns 0 if timer expires. (TIME = `read` timer.)
TIME == 0	**B:** `read` returns [MIN, *nbytes*] when available. (Caller can block indefinitely.)	**D:** `read` returns [0, *nbytes*] immediately.

Figure 11.7 Four cases for noncanonical input.

Be aware that POSIX.1 allows the subscripts VMIN and VTIME to have the same values as VEOF and VEOL, respectively. Indeed, SVR4 does this. This provides backward compatibility for older versions of System V. The problem is that in going from noncanonical to canonical mode, we must now restore VEOF and VEOL also. If we don't do this, and VMIN equals VEOF, and we set VMIN to its typical value of 1, the end-of-file character becomes Control-A. The easiest way around this problem is to save the entire termios structure when going into noncanonical mode and restore it when going back to canonical mode.

Example

Program 11.10 defines the functions tty_cbreak and tty_raw that set the terminal in a *cbreak mode* and a *raw mode*. (The terms cbreak and raw come from the Version 7 terminal driver.) We can reset the terminal to its prior state by calling the function tty_reset. Two additional functions are also provided: tty_atexit can be established as an exit handler to assure that the terminal mode is reset by exit, and tty_termios returns a pointer to the original canonical mode termios structure. We use all these functions in the modem dialer in Chapter 18.

```
#include    <termios.h>
#include    <unistd.h>

static struct termios   save_termios;
static int              ttysavefd = -1;
static enum { RESET, RAW, CBREAK }  ttystate = RESET;

int
tty_cbreak(int fd)  /* put terminal into a cbreak mode */
{
    struct termios  buf;

    if (tcgetattr(fd, &save_termios) < 0)
        return(-1);

    buf = save_termios; /* structure copy */

    buf.c_lflag &= ~(ECHO | ICANON);
                    /* echo off, canonical mode off */

    buf.c_cc[VMIN] = 1; /* Case B: 1 byte at a time, no timer */
    buf.c_cc[VTIME] = 0;

    if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
        return(-1);
    ttystate = CBREAK;
    ttysavefd = fd;
    return(0);
}

int
tty_raw(int fd)     /* put terminal into a raw mode */
{
    struct termios  buf;

    if (tcgetattr(fd, &save_termios) < 0)
        return(-1);
```

```
    buf = save_termios; /* structure copy */
    buf.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);
                    /* echo off, canonical mode off, extended input
                       processing off, signal chars off */
    buf.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);
                    /* no SIGINT on BREAK, CR-to-NL off, input parity
                       check off, don't strip 8th bit on input,
                       output flow control off */
    buf.c_cflag &= ~(CSIZE | PARENB);
                    /* clear size bits, parity checking off */
    buf.c_cflag |= CS8;
                    /* set 8 bits/char */
    buf.c_oflag &= ~(OPOST);
                    /* output processing off */
    buf.c_cc[VMIN] = 1; /* Case B: 1 byte at a time, no timer */
    buf.c_cc[VTIME] = 0;
    if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
        return(-1);
    ttystate = RAW;
    ttysavefd = fd;
    return(0);
}
int
tty_reset(int fd)       /* restore terminal's mode */
{
    if (ttystate != CBREAK && ttystate != RAW)
        return(0);
    if (tcsetattr(fd, TCSAFLUSH, &save_termios) < 0)
        return(-1);
    ttystate = RESET;
    return(0);
}
void
tty_atexit(void)        /* can be set up by atexit(tty_atexit) */
{
    if (ttysavefd >= 0)
        tty_reset(ttysavefd);
}
struct termios *
tty_termios(void)       /* let caller see original tty state */
{
    return(&save_termios);
}
```

Program 11.10 Set terminal mode to raw or cbreak.

Our definition of cbreak mode is the following:

- Noncanonical mode. As we mentioned at the beginning of this section, this mode turns off some input character processing. It does not turn off signal handling, so the user can always type one of the terminal-generated signals. Be aware, that the caller should catch these signals, or there's a chance that the signal will terminate the program, and the terminal will be left in cbreak mode.

  As a general rule, whenever we write a program that changes the terminal mode, we should catch most signals. This allows us to reset the terminal mode before terminating.
- Echo off.
- One byte at a time input. To do this we set MIN to 1 and TIME to 0. This is Case B from Figure 11.7. A `read` won't return until at least one byte is available.

We define raw mode as follows:

- Noncanonical mode. Additionally we turn off processing of the signal-generating characters (`ISIG`) and the extended input character processing (`IEXTEN`). We also disable a BREAK character from generating a signal by turning off `BRKINT`.
- Echo off.
- We disable the CR-to-NL mapping on input (`ICRNL`), input parity detection (`INPCK`), the stripping of the eighth bit on input (`ISTRIP`), and output flow control (`IXON`).
- Eight bit characters (`CS8`), and parity checking is disabled (`PARENB`).
- All output processing is disabled (`OPOST`).
- One byte at a time input (MIN = 1, TIME = 0).

Program 11.11 tests the raw and cbreak modes. Running Program 11.11 we can see what happens with these two terminal modes.

```
$ a.out
Enter raw mode characters, terminate with DELETE
                                                    4
                                                    33
                                                      133
                                                         62
                                                           63
                                                             60
                                                               172
                                  type DELETE
Enter cbreak mode characters, terminate with SIGINT
1                                 type Control-A
10                                type backspace
signal caught                     type interrupt key
```

```
#include    <signal.h>
#include    "ourhdr.h"

static void sig_catch(int);

int
main(void)
{
    int     i;
    char    c;

    if (signal(SIGINT, sig_catch) == SIG_ERR)   /* catch signals */
        err_sys("signal(SIGINT) error");
    if (signal(SIGQUIT, sig_catch) == SIG_ERR)
        err_sys("signal(SIGQUIT) error");
    if (signal(SIGTERM, sig_catch) == SIG_ERR)
        err_sys("signal(SIGTERM) error");

    if (tty_raw(STDIN_FILENO) < 0)
        err_sys("tty_raw error");
    printf("Enter raw mode characters, terminate with DELETE\n");
    while ( (i = read(STDIN_FILENO, &c, 1)) == 1) {
        if ((c &= 255) == 0177)     /* 0177 = ASCII DELETE */
            break;
        printf("%o\n", c);
    }
    if (tty_reset(STDIN_FILENO) < 0)
        err_sys("tty_reset error");
    if (i <= 0)
        err_sys("read error");

    if (tty_cbreak(STDIN_FILENO) < 0)
        err_sys("tty_raw error");
    printf("\nEnter cbreak mode characters, terminate with SIGINT\n");
    while ( (i = read(STDIN_FILENO, &c, 1)) == 1) {
        c &= 255;
        printf("%o\n", c);
    }
    tty_reset(STDIN_FILENO);
    if (i <= 0)
        err_sys("read error");
    exit(0);
}

static void
sig_catch(int signo)
{
    printf("signal caught\n");
    tty_reset(STDIN_FILENO);
    exit(0);
}
```

Program 11.11 Test the raw and cbreak modes.

In raw mode the characters entered were Control-D (04) and the special function key F7. On the terminal being used, this function key generated six characters: *ESC* (033), *[* (0133), *2* (062), *3* (063), *0* (060), and *z* (0172). Notice with the output processing turned off in raw mode (`-OPOST`) we do not get a carriage-return output after each character. Also notice that special character processing is disabled in cbreak mode (so Control-D, the end-of-file character, and backspace aren't handled specially), while the terminal-generated signals are still processed. □

11.12 Terminal Window Size

SVR4 and Berkeley systems provide a way to keep track of the current terminal window size and to have the kernel notify the foreground process group when the size changes. The kernel maintains a `winsize` structure for every terminal and pseudo terminal.

```
struct winsize {
  unsigned short  ws_row;     /* rows, in characters */
  unsigned short  ws_col;     /* columns, in characters */
  unsigned short  ws_xpixel;  /* horizontal size, pixels (not used) */
  unsigned short  ws_ypixel;  /* vertical size, pixels (not used) */
};
```

The rules for this structure are as follows:

1. We can fetch the current value of this structure using an `ioctl` (Section 3.14) of `TIOCGWINSZ`.
2. We can store a new value of this structure in the kernel using an `ioctl` of `TIOCSWINSZ`. If this new value differs from the current value stored in the kernel, a `SIGWINCH` signal is sent to the foreground process group. (Note from Figure 10.1 that the default action for this signal is to be ignored.)
3. Other than storing the current value of the structure and generating a signal when the value changes, the kernel does nothing else with the values in this structure. Interpreting the values in the structure is entirely up to the application.

The reason for providing this feature is to notify applications (such as the `vi` editor) when the window size changes. When the application receives the signal it can fetch the new size and redraw the screen.

Example

Program 11.12 prints the current window size and goes to sleep. Each time the window size changes, `SIGWINCH` is caught and the new size is printed. We have to terminate this program with a signal.

```
#include    <signal.h>
#include    <termios.h>
#ifndef TIOCGWINSZ
#include    <sys/ioctl.h>    /* 4.3+BSD requires this too */
#endif
#include    "ourhdr.h"

static void pr_winsize(int), sig_winch(int);

int
main(void)
{
    if (isatty(STDIN_FILENO) == 0)
        exit(1);

    if (signal(SIGWINCH, sig_winch) == SIG_ERR)
        err_sys("signal error");

    pr_winsize(STDIN_FILENO);    /* print initial size */
    for ( ; ; )                  /* and sleep forever */
        pause();
}

static void
pr_winsize(int fd)
{
    struct winsize  size;

    if (ioctl(fd, TIOCGWINSZ, (char *) &size) < 0)
        err_sys("TIOCGWINSZ error");
    printf("%d rows, %d columns\n", size.ws_row, size.ws_col);
}

static void
sig_winch(int signo)
{
    printf("SIGWINCH received\n");
    pr_winsize(STDIN_FILENO);
    return;
}
```

Program 11.12 Print window size.

Running Program 11.12 on a windowed terminal gives us

```
$ a.out
35 rows, 80 columns          initial size
SIGWINCH received            change window size: signal is caught
40 rows, 123 columns
SIGWINCH received            and again
42 rows, 33 columns
^? $                         type the interrupt key to terminate
```

□

11.13 termcap, terminfo, and curses

termcap stands for "terminal capability," and it refers to the text file /etc/termcap and a set of routines to read this file. The termcap scheme was developed at Berkeley to support the vi editor. The termcap file contains descriptions of various terminals: what features the terminal supports (how many lines and rows, does the terminal support backspace, etc.) and how to make the terminal perform certain operations (clear the screen, move the cursor to a given location, etc.). By taking this information out of the compiled program and placing it into a text file that can easily be edited, it allows the vi editor to run on many different terminals.

The routines that support the termcap file were then extracted from the vi editor and placed into a separate curses library. Lots of features were added to make this library usable for any program that wanted to manipulate the screen.

The termcap scheme was not perfect. As more and more terminals were added to the data file, it took longer to scan the file looking for a specific terminal. The data file also used two-character names to identify the different terminal attributes. These deficiencies led to development of the terminfo scheme and its associated curses library. The terminal descriptions in terminfo are basically compiled versions of a textual description and can be located faster at run time. terminfo appeared with SVR2 and has been in all System V releases since then.

SVR4 uses terminfo, while 4.3+BSD uses termcap.

A description of terminfo and the curses library is provided by Goodheart [1991]. Strang, Mui, and O'Reilly [1991] provide a description of termcap and terminfo.

Neither termcap nor terminfo, by itself, addresses the problems we've been looking at in this chapter—changing the terminal's mode, changing one of the terminal special characters, handling the window size, and so on. What they do provide is a way to perform typical operations (clear the screen, move the cursor) on a wide variety of terminals. On the other hand, curses does help with some of the details that we've addressed in this chapter. Functions are provided by curses to set raw mode, set cbreak mode, turn echo on and off, and the like. But curses is designed for character-based dumb terminals, while the trend today is toward pixel-based graphics terminals.

11.14 Summary

Terminals have many features and options, most of which we're able to change to suit our needs. In this chapter we've described numerous functions that change a terminal's operation—special input characters and the option flags. We've looked at all the terminal special characters and the many options that can be set or reset for a terminal device.

There are two modes of terminal input—canonical (line at a time) and noncanonical. We showed examples of both modes and provided functions that map between the POSIX.1 terminal options and the older BSD cbreak and raw modes. We also described how to fetch and change the window size of a terminal. Chapters 17 and 18 show additional examples of terminal I/O.

Exercises

11.1 Write a program that calls `tty_raw` and terminates (without resetting the terminal mode). If your system provides the `reset`(1) command (both SVR4 and 4.3+BSD provide it) use it to restore the terminal mode.

11.2 The `PARODD` flag in the `c_cflag` field allows us to specify even or odd parity. The BSD `tip` program, however, also allows the parity bit to be 0 or 1. How does it do this?

11.3 If your system's `stty`(1) command outputs the MIN and TIME values, do the following exercise. Log in to the system twice and start the `vi` editor from one login. Use the `stty` command from your other login to determine what values `vi` sets MIN and TIME to (since it sets the terminal to noncanonical mode).

11.4 As the terminal interface moves to faster line speeds (19,200 and 38,400 are becoming common nowadays) the need for hardware flow control becomes important. This involves the RS-232 RTS (request to send) and CTS (clear to send) signals, instead of the XON and XOFF characters. Hardware flow control is not specified by POSIX.1. Under SVR4 and 4.3+BSD how can a process enable or disable hardware flow control?