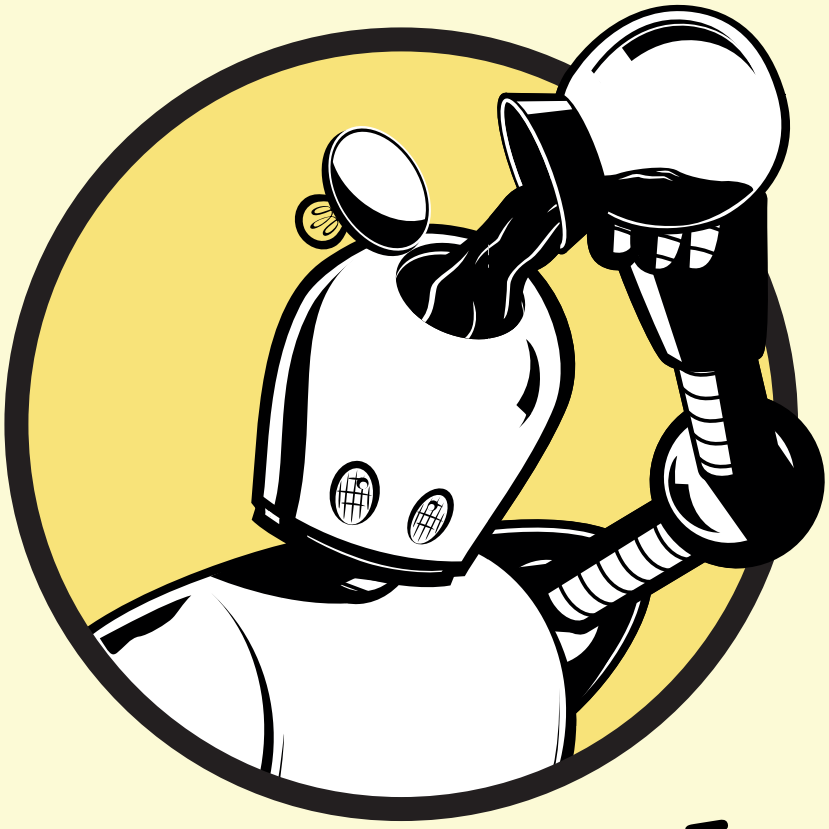


# EARLY ACCESS

no starch press®

# NO STARCH PRESS EARLY ACCESS PROGRAM: FEEDBACK WELCOME!

The Early Access program lets you read significant portions of an upcoming book while it's still in the editing and production phases, so you may come across errors or other issues you want to comment on. But while we sincerely appreciate your feedback during a book's EA phase, please use your best discretion when deciding what to report.

At the EA stage, we're most interested in feedback related to content—general comments to the writer, technical errors, versioning concerns, or other high-level issues and observations. As these titles are still in draft form, we already know there may be typos, grammatical mistakes, missing images or captions, layout issues, and instances of placeholder text. No need to report these—they will all be corrected later, during the copyediting, proofreading, and typesetting processes. Please note that any online resources may not be available until the book is complete.

If you encounter any errors ("errata") you'd like to report, please fill out **this Google form** so we can review your comments.

# INTRODUCTION TO SYSTEM PROGRAMMING IN LINUX

STEWART N. WEISS

Early Access edition, 01/23/24



ISBN 13: 978-1-7185-0356-4 (print)

Published by No Starch Press®, Inc.
245 8th Street, San Francisco, CA 94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Sabrina Plomitallo-González

For customer service inquiries, please contact info@nostarch.com. For information on distribution, bulk sales, corporate sales, or translations: sales@nostarch.com. For permission to translate this work: rights@nostarch.com. To report counterfeit copies or piracy: counterfeit@nostarch.com.

# CONTENTS



The chapters in red are included in this Early Access PDF.

# PREFACE


The story of this book can trace its origins to 1983. In that year I was a graduate student in the Courant Institute of Mathematical Science at New York University. The Courant, as it was called, had acquired Sun Microsystem workstations, which were running Berkeley Software Distribution 4.1, an early version of Unix. After I learned the rudiments of Unix, it didn't take long for me to become a Unix convert.

When I was hired to teach in a full-time position at Hunter College in 1987, I set up a small Unix network with my research computers. I didn't know how I could be efficient in my work without Unix to help me.

After a while I decided that my students should also learn how to work in a Unix environment. If it was a good thing for me, then it would be a good thing for them too. I didn't give much thought to it back then, but I had become what these days people would call a Unix evangelist. In 1996, my dear friend Matthew Smosna was teaching an elementary course in the use of Unix tools and scripting at NYU and was kind enough to share his notes and slides with me. Using his notes as a starting point, I created a sim-

ilar course at Hunter College in 1997. Matthew passed away unexpectedly that year, but my gratitude to him has endured.

The course began as a tutorial on a rather large collection of Unix tools, ranging from text filtering tools such as `sed`, `awk`, and `grep`, to shell scripting and a smattering of Perl, to an overview of the Unix operating system and its structure. Many of the students whom I taught returned to me after graduation, thanking me for giving them the chance to learn Unix because they landed their first jobs because of it. Many students wanted to learn it in more depth, and so, in 2001, I converted the course to one in systems programming in Unix.

My goal was to teach a course in Unix system programming that was accessible to any computer science student who had had a year of programming, in C++ at the time, and who had taken a class in operating systems. I started putting all of my lecture notes on my website and began to receive thank you notes from people around the world for making them available. It was nice to get this kind of feedback, but at the same time it made me realize that because the notes were highly visible, I needed to perfect them.

Through my years of teaching I've learned that most, if not all, students learn programming only if they get a chance to write lots of programs and see many examples of the concept that I'm currently teaching. Therefore, I've always created and shared many *demo programs* with them and told them they should copy them and modify them and see how the changes affect its behavior. There is nothing like hands-on work to drive ideas home. This book is modeled on this principle.

# INTRODUCTION


There are several advantages to you to learn how to write system programs in a Unix/Linux environment, and I will assume that if you are reading this, you either know that already or have heard from someone else that perhaps it's worth your while to learn more about it. You can find resources online for learning how to develop programs in Unix, but many are designed for people who are knowledgeable in Unix and are expert programmers. This book does not require you to have any prior programming experience with Unix or Linux and does not assume that you're an expert programmer already. It has several distinct but related objectives:

- to teach you how to write programs on and for the Linux operating system;

- to show you how you can work efficiently within a Unix/Linux environment;
- to teach you how the Unix operating system is designed and structured, so that you have a deeper understanding of what happens "under the hood", so to speak, and
- to give you an appreciation of the marvel of Unix, so that you will want to learn as much as you can.

These are pretty hefty objectives, and they may seem to be too much to attain in one book. To achieve them, the book is neither thorough nor comprehensive. It isn't a reference book on everything Unix. It doesn't cover every aspect of programming in the Unix environment. It is tutorial and hands-on, and covers what I hope is enough to get you started and to show you how you can learn more on your own, but it is also conceptual, showing you not just *what* various features do and how to use them, but *how* they work, so that you understand *why* you need to do what you do and why something is not working the way you expected it to when things go wrong. The book's approach is similar to the message in the well-known proverb often attributed to the Chinese philosopher, Lao-Tzu, paraphrased as follows:

> Give a person a fish and they will eat for a day. Teach a person to fish and they will eat for a lifetime.

When I present material about a particular topic, I will often do so from three different perspectives.

- One perspective is that of a system programmer in that it presents the information needed to write system programs in a Unix/Linux system.

- A second is that of a non-programming user, because it presents the command level view of Unix.
- The third perspective is that of a computer scientist, because it examines the internal structure of the GNU/Linux operating system.

These are not mutually exclusive categories; you will at times be a system programmer, at other times an ordinary user, and perhaps you are always a computer scientist at heart. My hope is that each of these perspectives will be valuable to you.

Although the book includes tutorial information in case you have little or no experience with Unix/Linux as a user, this material can be skipped if you are an experienced user. There is some very basic background that you are expected to have:

- You should be able to login to a Unix system, open up a terminal window, and accomplish elementary tasks such as changing your password.
- You should be able to read C programs and understand them, mostly.
- You should know how to use elementary macros that define symbols and include header files.

## UNIX, Unix, Linux and More

Unix has a history dating back to 1969 and since that time many different variants have been developed, of which Linux is one. In 1969 and for many years after that, Unix was always written as *UNIX* because its name was a pun based on an earlier system named *MULTICS* on which its original developers worked. In fact for a very short period of time it was called UNICS. In 1993, *UNIX* became a registered trademark of *The Open Group*, a consortium of companies. The term *Unix* is not trademarked and does not refer to any one operating system. In general it refers to any operating system that is what people often call *Unix-like*. In the interest of clarity, when the term *UNIX* appears in the text, it refers very narrowly to any operating system that

has been certified by The Open Group as conforming to its branding of the term. I mostly use the word *Unix*, which in some contexts has precise meaning and in others, does not.

One important consequence of the fact that there are so many different varieties of Unix is that a program that works on one Unix system may not work on another. This problem led over time to the standardization of Unix. Chapter 1 contains a brief history of the various applicable standards. The general problem of writing programs that work across a variety of operating systems is called *portability*. Chapter 1 also describes steps that you can take to make your code portable to Unix systems other than the one on which you wrote it, provided that they conform to one standard or another.

The term *Linux* poses a slightly different problem. Technically speaking, Linux is not an entire operating system with all of its utilities and programs that come bundled together in an installation package. It is just what is commonly called the *kernel*, which is defined in Chapter 1. The rest of the operating system is mostly programs and libraries developed by *GNU* as part of the *GNU Project* [1]. (*GNU* is a recursively defined acronym for ***G**NU's **N**ot **U**nix*.) For this reason there are many people who believe it should be called *GNU/Linux*. I am one of those who believe that its name should reflect the major contribution to it by the GNU Project; therefore, when I want to refer specifically to the entire operating system I will sometimes call it GNU/Linux as a reminder, but when I refer specifically to its kernel, I will call it Linux. Chapter 1 has a more detailed discussion of the history of UNIX, the GNU Project, and Linux.

## The Importance of Open Source Software

The fact that Unix has been an open source operating system is fundamental to the teaching paradigm on which this book is based. Each time that a new topic is introduced, such as file systems or multi-threading, my strategy is to explain the underlying concepts, perhaps look at the design and structure of the interfaces, and then pick a command to implement. We'l go through a few iterations to get it right, fixing problems, learning about what did and did not work, until we are satisfied with the solution. Then, after it is all tidied up, we'll look at selected files from the actual Linux or GNU library implementations. None of this would be possible with proprietary code.

## About C and C++ in This Book

The programming examples that I use here are written in C, and this implies that you should be able to read and understand simple C programs. If you know C++ you are sufficiently prepared. Many people who know C++ often think that they don't know C and they get discouraged. This is not quite true. The C++ language is more or less an extension to C and embeds it. If you know C++, you know a great deal of C. There are minor differences that

1. https://www.gnu.org/gnu/gnu.html

arise in the syntax of declarations (such as struct and function declarations), but the real problem is that most C++ programmers never learn how to use the C libraries. Most use C++ stream I/O and never learn what they think are the archaic functions of the C standard I/O library. These functions are at times much more useful and powerful than any found in C++.

If you have not programmed in C or C++, but have in a language such as Python or Java you will have more to learn, and I suggest obtaining a companion C language reference or tutorial to have in hand when reading this book. I include a list of C resources at the end of this Introduction.

## Notation Used in the Book

In the description of a command or function, square brackets `[ ]` enclose optional arguments. The brackets are not part of the command. Italic text denotes placeholders, not actual text that you type. An ellipsis `...` means more than one copy of the preceding token. For example, the description

```
ls [option] ... [directory_name] ...
```

indicates that both the option specifiers and the argument to the `ls` command are optional, but that any options should precede any directory names, and that both option specifiers and directory names can occur multiple times, as in

```
ls -l -t mydir yourdir
```

The words *option* and *directory_name* are placeholders for one or more options to `ls`.

A vertical bar | is a logical-or, indicating one choice of multiple alternatives. For example,

```
bash [options] [command_string | file]
```

indicates that after the various options, you can supply either a command string or the name of a file but not both.

Throughout the book we will use the `$` character as the prompt string displayed inside a terminal window. The text that you would enter after it is shown in **boldface**. For example, there is a command named `echo` that just prints out whatever you enter after it. We would demonstrate how you use it as follows:

```
$ echo 'Is this really how echo works?'
Is this really how echo works?
$
```

Notice that the prompt character is displayed again. This is how we indicate that you are seeing *all* of the output of the command. If we don't show all of the output, it would be displayed as follows:

```
$ echo 'Is this really how echo works?'
```

```
Is this really how echo works?

--snip--
```

In the Unix system that you use for following along with this book, the prompt character that you see might be something other than `$`. Many systems might have a default prompt that includes more information, such as your login name or the name of the computer. In fact you are usually able to customize your prompt.

# 1

# BASIC CONCEPTS OF UNIX AND SYSTEM PROGRAMMING

This first chapter presents the big picture of system programming and basic background information on Unix. We first explore what system programs are and how they are different from other kinds of programs. Next, we'll introduce many of the fundamental concepts that underlie the Unix family of operating systems, and we'll conclude with a brief discussion of the history and standardization of Unix and the C programming language.

When we examine the various concepts that make Unix what it is, we start with the kernel, because the kernel is essentially the core of a Unix system. From there, we move on to Unix shells, which are the programmable, interchangeable user interfaces in Unix systems, separate and distinct from the Unix kernel. After that, we'll cover the concepts of users and groups. We then explain how the distinction between privileged and unprivileged instructions in Unix enables the kernel to manage resources securely and efficiently. Next, we'll introduce the concept of a user *process*, a representation of a running user program managed by the kernel, and *threads*, which are particular types of processes in Unix. We'll also explain the idea of an *en-*

*vironment list*, which is a set of variables and values passed to new processes, and we'll describe the Unix directory hierarchy and present an overview of files, directories, and permissions.

One important part of any Unix system is its online documentation, which plays a critical role in how you'll learn system programming in this book. We'll cover its organization and use in this chapter as well.

## What Is System Programming?

The first programs people learn to write are simple ones, but despite their simplicity, we can use them to explain what system programming is. The simplest possible program that actually does something has no input and just prints a message onto the screen. One such program is the ubiquitous "hello, world" program that you most likely wrote at the beginning of your development as a programmer. The first C version of this program appeared in Kernighan and Ritchie's *The C Programming Language* (Prentice-Hall, 1978) and has since become the *de facto* first program that students learn to write when they are learning a new programming language. Here's the original version of that program:

*hello_world.c*

```
#include <stdio.h>
void main()  {
    printf("hello, world.\n");
 }
```

*Listing 1-1: Kernighan and Ritchie's original "hello,world" program*

The first line is an *include directive*, which starts with keyword `#include` and is followed by the specification of a file. It tells the C preprocessor to read the contents of the file, in this case the C header *stdio.h*, at that point in the program. We need that action to take place because the main program makes a call to the C `printf()` function, whose declaration is in *stdio.h*. Without it, the compiler could not tell whether `printf()` was being called properly. The C preprocessor has to find the header file before it can read it, and header files can be in many possible places. The angle brackets `<...>` around the name of the file tell the preprocessor that it's in one of the standard places that it searches.

### The Magic of Input and Output

The `printf()` function is one way in C to print information on the screen. I'm using the word *screen* as a synonym for the more technical term, *terminal window*. In this simple example, the `printf()` has a single string "`hello, world.\n`" as its argument. The `\n` is the *newline* character, which causes the next character following it to appear at the beginning of the next line on the screen.

Let's assume that this code is stored in a file named *hello_world.c* and that we've already compiled it into an executable program named *hello_world*. Because we don't want to be sidetracked by the details of how to compile code on a Unix system, we omit that compilation procedure here.

Running this program in a terminal window causes the string `hello, world.` to be displayed, and then the prompt reappears. To run the program, we enter its name, preceded by `./`, and press ENTER:

```
$ ./hello_world
hello, world.
$
```

For someone who has never written a program before, this seems like magic. All you have to do is include the *stdio.h* header file in the code and give the `printf()` function the string that you want to print, and voilá, when you run the program the string appears.

It clearly isn't magic though, and a lot must be going on behind the scenes to make the characters appear on the screen. C has given us a very powerful tool, `printf()`, so that we can write programs that print to the screen without needing to learn a lot about terminals and other technology.

Let's take this one step further. The preceding program outputs text but has no input. The next program performs both input and output:

*hello.c*

```
#include <stdio.h>
void main() {
 ❶ char    username[256];
   printf("Enter your name: ");
 ❷ scanf("%255s", username);
   printf("hello, %s\n", username);
}
```

*Listing 1-2: A program that performs both input and output*

This program begins with a declaration of a `char` array named `username` of length 256 ❶. This array can store up to 255 characters plus a terminating *null byte*. A null byte is the non-printing character whose code is zero. As a character, it's written as `\0`. In many programming languages, a null byte is required at the end of a string. The program then prints a prompt message to the screen asking the user to enter their name. Then the C `scanf()` function ❷ reads characters from the keyboard until either 255 characters are entered or it finds a white space character (blank, tab, or newline), and it stores this data into the `username` array. The program prints `hello` followed by the contents of that array.

### THE SCANF() FUNCTION

The `scanf()` function is the C Library's formatted input function. It reads input, by default from the keyboard, following a format that you give it. In general, its first parameter is a string enclosed in double quotes followed by one or more pointers. The double quoted string is called the *format specification*. In this example, it is `"%255s"`, which specifies that the input data should be stored as a string (`s` for string) with a maximum width of 255 characters. The argument following the format specification must be a pointer to the start of a character

array large enough to hold 255 characters plus the null byte. Since in C, array names can be used wherever a constant pointer is expected, the array `username` is a valid argument.

Let's think about how this input and output actually take place. The program makes calls to the `scanf()` and `printf()` functions, but where is their code and how is it executed? Many beginning programmers mistakenly believe that the header files included by their programs contain the function implementations, because all they have to do is put appropriate include directives into their programs for them to work. However, those implementations are not in the header files.

### The Role of the C Library in I/O

In general, when functions are not defined in the same file as the code calling them, they need to be *linked* into that code. When a compiler is processing a file and finds a function call not defined in the same file, it marks the call as an *unresolved symbol* because it can't assign an address for that function. The same is true if it finds unknown type names, variable names, and so on. Linking is the act of assigning addresses to these unresolved symbols. The compiler does not link. The *linker* is the component of the compiler collection that resolves undefined symbols. A *compiler collection* is a set of programs that build programs from source code.

The C language provides no built-in facilities for performing operations such as input/output, memory management, and string manipulation. Instead, they're defined in a standard *library*, which you compile and link into your program. In the case of `scanf()` and `printf()`, their definitions are part of the *C Standard Library*. The linker automatically searches through this library without your needing to do anything special, thereby finding the function definitions. In "Object Libraries" on page 86 we explain exactly what a library is and how you can see what it contains.

Inside that library, the `scanf()` function makes calls to a lower level function named `read()`. The code that implements `read()` is not part of the library; it's part of the Unix operating system itself. You'll get a better picture of this in the following section, "System Resources" and a more detailed explanation in Chapter 3, "System Calls." The `read()` code performs the actual transfer of bytes from the input device to the program's memory. Similarly, the `printf()` function makes calls to a lower level function named `write()`, which is also implemented within the operating system. The `write()` code handles all of the details of writing to the terminal. In short, the operating system performs all transfers of data to and from the terminal. Figure 1-1 illustrates how this happens.

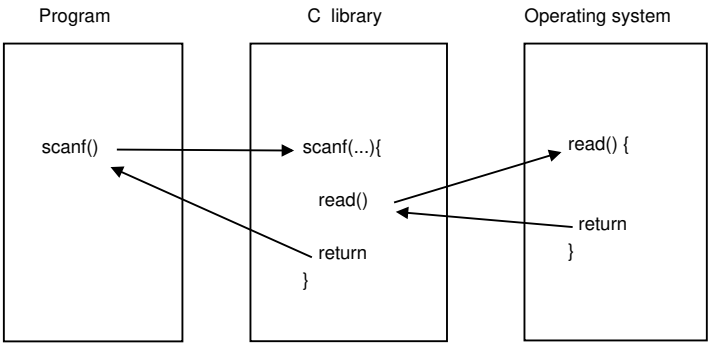



*Figure 1-1: Execution flow of input operation*

When we run a program like *hello.c* in Listing 1-2, we have the illusion that the program is connected directly to the keyboard and the display device via C library functions. If you run the program on your own personal computing device, this illusion may not be far from reality. However, we can also run it on a multi-user system in a terminal window, and the results will be exactly the same. This fact complicates the picture even further. In a Unix system, and in almost all modern operating systems, many people can work on the system at the same time, and programs belonging to different people can run at the same time, each receiving input from a different keyboard and sending output to a different display. Each person will see the same output as if they had run the program on a single-user machine. The operating system is what makes this possible. It has to ensure that each user's programs do not interfere with each other.

## System Resources

We can frame this problem in terms of resources. *Resources* are objects that software uses and/or modifies. For example, a program's input and output data are resources, as are the values that it stores in its internal data structures. A program has the privilege to access or modify any of its own resources.

In Unix systems, some resources are protected from access by ordinary programs and are accessible only by the operating system. These protected resources are called *system resources*. System resources include hardware, such as the CPU, physical memory, screen displays, storage devices, and network connections. They also include objects that aren't hardware, such as system data structures and files. These are sometimes called *soft resources*. Figure 1-2 illustrates the way an operating system is layered in order to control access to system resources.

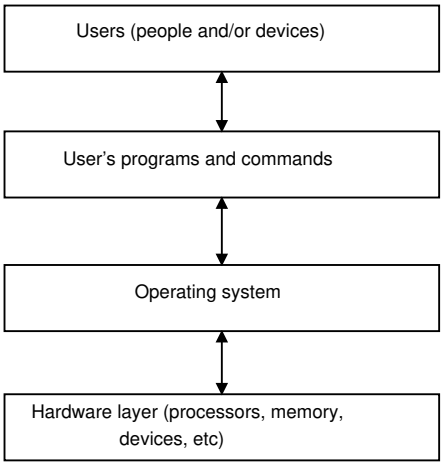



*Figure 1-2: An operating system has layers to protect resources.*

A modern operating system such as Unix provides an interface that programs can use for requesting access to system resources. This interface is called its *application programming interface (API)*. In computer science jargon, the term *application* is often used to refer to programs intended to be run by ordinary users. An API typically consists of a collection of function, type, and constant definitions, and sometimes variable definitions as well. The API that an operating system provides in effect defines the means by which an application can request services that the operating system provides. The functions in the API are called *system calls*.

**NOTE** *If you're familiar with object-oriented programming, you may notice a resemblance between the operating system's API and a class interface. Both provide a set of methods for accessing protected data only through a well-defined set of access points.*

## System Programs Explained

We've now set the stage to make a distinction between ordinary programs and system programs. A program that's not a system program is designed as if it has exclusive access to all of the resources it uses. It doesn't deal with the complexity of connecting to monitors and keyboards and isn't cognizant of the fact that the operating system must manage these resources.

In contrast, a program that makes direct requests for the services exposed in an operating system's API is called a *system program*, and when we write this kind of program we are *system programming*. System programs make requests for resources and services directly from the operating system. Sometimes people write them to extend the functionality of the operating system itself or provide functions that higher level applications can use. For example, we could write a program that gets the current time from the operating system's internal clock and displays it in various formats for any user. This would be a system program.

The term *system program* also applies to any program that can run independently of the operating system and extend its functionality, even if

it doesn't make any direct calls to the API. Tools such as compilers, assemblers, linkers, terminal emulators, and so on are considered to be system programs, and they play a fundamental role in a computer system. As Richard Stallman [28] wrote, "The kernel is an essential part of an operating system, but useless by itself; it can only function in the context of a complete operating system." In this view, system programs are like an extension to the operating system, even though their definition is a bit fuzzy. The primary purpose of this book is to show you how you can write programs of this nature, namely those that interact directly with the operating system and, in effect, act like a part of that system.

## Fundamental Concepts of Unix

This section introduces the core concepts that underlie the design of the Unix operating system. From its beginning, Unix was designed around a small set of clever ideas, as its authors, Dennis Ritchie and Ken Thompson[24], put it: "The success of UNIX lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small yet powerful operating system." Those "fertile ideas" included, in the order in which we discuss them, the concepts of a programmable shell, users and groups, privileged and unprivileged instructions, environments, files and the directory hierarchy, device-independent input and output, and most important, processes. I describe these concepts in this section, not in great detail, preceded by a brief overview of that "small yet powerful operating system" itself, now known as the Unix *kernel*. Along the way, I introduce some Unix commands to demonstrate the concepts.

Before we dive into all of this material, we need to address one sticky point having to do with standards. Unix has many different varieties, which many people call *flavors*. This is a consequence of its history, and in "Unix History and Standards" later I summarize this problem and how it's been resolved, in particular making reference to the most important family of standards known as the *POSIX* standards, an acronym for *Portable Operating System Interface*. Between here and that section, I'll sometimes make reference to the fact that something does or does not conform with POSIX.1-2017, which is the most recent version as of this writing. We cannot overstate the importance of conformance; commands and functions that are either not specified by or not conforming to the POSIX.1 standard are not portable and not guaranteed to work on all Unix systems that conform to the standard. I'll point this out when it is relevant.

### The Unix Kernel

It is perhaps unfortunate that the term *operating system* has no single, universally agreed upon definition. If you look at almost any textbook on operating systems [27, 30], you'll find two different views of what constitutes an operating system:

- One view is that the operating system is the collection of all software that provides services to applications and users and manages and protects all hardware resources. In this view, tools like user interfaces and browsers are part of the operating system.
- A more narrow view is that the operating system is only the program that is loaded into memory on startup and remains in memory, controlling all computer resources, until the computer is powered off.

Regardless of which definition you decide to adopt, the term *kernel* is unambiguously used as another name for the second definition. It's an appropriate name, since it's the core of the Unix system. McKusick *et al*, in their seminal book on the design of the 4.4BSD operating system [15], define a kernel as "a small nucleus of software that provides only the minimal facilities necessary for implementing additional operating system services." In this book, I use the narrow definition of an operating system, namely that it is the kernel and nothing more.

The kernel is a program, or a collection of interacting programs, depending on the particular implementation of Unix, with many *entry points*. An entry point is an instruction in a program at which execution can begin. Each of these entry points provides a service that the kernel performs. If you are used to thinking of programs as always starting at their first line, this may be disconcerting.

Most likely, in the programs that you have written so far, there has been a single entry point, namely the `main()` function. However, it's possible to create code that can have several entry points. Software libraries are code modules with multiple entry points. You can think of entry points as functions that can be called by other programs. They perform services such as opening, reading, and writing files, creating new processes, allocating memory, and so on. Each of these functions expects a certain number of arguments of certain types, and produces well-defined results. The collection of kernel entry points makes up a large part of its API. In fact, you can think of the kernel as containing a collection of separate functions, bundled together into a large package, and its API as the collection of signatures or prototypes of these functions.

### Kernel Roles and Responsibilities

What are the kernel's responsibilities, and what does it do? The goal for now is to paint a picture of *what* the kernel does and not to describe *how* it does this.

When a Unix system boots, a combination of firmware and software loads the kernel into the portion of memory called *system space* or *kernel space*, where it stays until the machine is shut down. User programs are not allowed to access system space. If they try, the kernel terminates them.

The kernel has full access to all of the hardware attached to the computer. The kernel maintains various system resources in order to perform services for user programs. These system resources include many different

data structures that keep track of input/output (*I/O*), memory, and device usage for example.

The Unix kernel manages and protects all of these resources and provides an operating environment that allows all users to work efficiently, safely, and happily. It prevents users and the programs that they run from accessing any hardware resources directly. In other words, if a user's running program wants to read from or write to a disk, it must ask the kernel to do that on its behalf, rather than doing it on its own. The kernel will perform the task and transfer any data to or from a portion of memory that the user's program can access.

To understand why this is necessary, consider what would happen if users' programs could access the hard disk directly. A user could run a program that could try to acquire all disk space, or even worse, try to erase the disk, subverting the kernel's ability to protect its resources.

The Unix kernel also protects users from each other and protects itself from users, while simultaneously giving users the impression that they each have the computer entirely to themselves. This is precisely the illusion described in the section "What Is System Programming?" on page 2. Somehow everyone is able to run programs that seem as if they have the computer all to themselves, as if no one else were using the machine. Users have their own disk space, their own private portion of memory, their fair share of time on the CPU, and so on.

In order to achieve these objectives, the inventors of Unix incorporated several key principles into its design:

- The system designates two levels of privilege—user privilege and kernel privilege—such that certain instructions can be executed only with kernel privilege.
- Each user has a unique identity. A privileged user can create groups of users, and those groups have unique identities as well. These user and group identifiers are assigned privileges and protections for all user resources such as disk storage, running programs, and so on.
- The system of files supports creation, modification, retrieval, and removal of persisted data and programs, as well as privacy, protection, and the ability to share software and data.
- Physical memory is divided into two regions: one called *user space* where ordinary user programs are loaded, and one called *system space*, which is where the operating system itself is stored.
- The kernel has exclusive control of the use of the processor, and it decides at any given time, what runs next.
- The kernel has the exclusive ability to load programs into memory, run them, and terminate them. A running program cannot even terminate itself; the best it can do is to ask the kernel to terminate it!
- The kernel has complete and exclusive control of all computer hardware.

We'll describe each of these principles in more depth in the remaining sections of this chapter.

### Kernel Services

I've mentioned reading and writing files and terminal I/O as some of the types of services that the kernel provides, but to give you an even better sense of the scope of its services, the following list shows the types of services it performs:

- Process scheduling and management
- I/O handling
- Physical and virtual memory management
- Device management
- Filesystem management
- Signaling and inter-process communication
- Multi-threading
- Protection and security
- Networking services

Figure 1-3 depicts how users and their programs access system resources and services through the kernel's application programming interface.

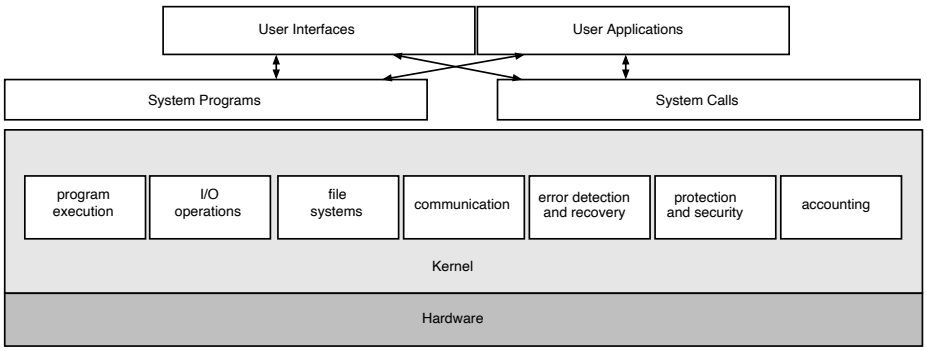



*Figure 1-3: A schematic view of the role of the kernel*

In the figure, each of the boxes inside the kernel region represents a different service category. The box labeled *System Calls* represents the part of the API that programs use to request and obtain these services, whereas the box labeled *System Programs* is the set of standalone programs that users can run to obtain these services.

## Shells and Commands

The kernel provides services to running programs, but not directly to users; instead, users interact with Unix by entering commands through a command line interpreter running in a terminal window, or by interacting with

a graphical user interface (GUI), which I do not discuss in this book. A *command line interpreter* is a program that reads commands and carries them out.

### Commands

A *command* is an instruction that you enter by inputting text, usually but not always, using a keyboard. Commands may have options and arguments following the command name. *Options* modify the behavior of the command, whereas *arguments* are the command's inputs. For example:

```
$ gcc -g -o myprog myprog.c
```

The following list explains each part of that command line:

**`gcc`** The command name (the GNU Compiler Collection).

**`-g`** An option to `gcc` that tells it to include debugging information in the generated executable.

**`-o myprog`** An option with an *option argument*, `myprog`. The `-o` option tells `gcc` to put the output into the file named immediately after it, in this case, *myprog*, which is its argument.

**`myprog.c`** The command's only argument, which is the name of its input file.

The command line is everything that you type up to but not including the newline character produced when you press ENTER. In this example, the command is the entire command line, but sometimes a single line can have multiple commands separated by command-separator characters such as the semi-colon, as in:

```
$ gcc -g -o myprog myprog.c ; gcc -g -o hello hello.c
```

Technically, a *simple command* is a single command, not a sequence of commands. When we use the term *command*, we usually mean a simple command.

In GNU/Linux and some other Unix systems, some commands have two kinds of command options, *short* and *long*:

- Short options begin with a single dash -, and are a single character, as in `-a` and `-H`.
- Long options start with a double-dash -- and can be words, such as `--date` and `--file-type`.

POSIX.1-2017 does not require conforming systems to provide long options, but GNU/Linux has them.

Both types of options can have option arguments. For example, in

```
$ gcc -g -o myprog myprog.c
```

the `-o` option has the `myprog` argument.

In Unix systems that conform to the POSIX.1-2017 standard, if an option has an argument, the argument is required; you cannot omit it. On the other hand, GNU/Linux permits a command to have options with non-required arguments. For example, you can enter the *firefox* web browser's name to start it from the command line in GNU/Linux:

```
$ firefox
```

If you give it the `-P myprofile` option, it starts up with the user profile named `myprofile`. If you enter just

```
$ firefox -P
```

it displays a dialog box asking you to pick a profile from a list. The profile name is a non-required argument to `-P`.

The rules for giving option arguments are:

- The argument to a short option follows it immediately, possibly with intervening space or TAB characters, as in `-ohello` or `-o hello`. The one exception is that non-required arguments can't have space before them.
- The argument to a long option follows the `=` operator *without intervening space*, as in `--date='Jan 01,1970'`.

The typical command consists of the command name followed by options and then arguments, but some commands allow the options and arguments to be intermixed. For example:

```
$ gcc -g myprog.c
$ gcc myprog.c  -g
```

These command lines are equivalent.

### Shells

The word *shell* is the Unix term for a particular type of command line interpreter. Command line interpreters have always been provided with operating systems since their inception. Early mainframes and personal computer operating systems required people to interact with them exclusively through a command line interpreter. DOS, for example, provided a command line interpreter, which became the basis for the Microsoft Command window, which was simply a DOS emulator.

A command line interpreter presents a prompt of some kind, indicating that it's waiting for you to enter a command. At the prompt, you type a command and press ENTER, causing the command to be executed, after which the prompt reappears:

```
$ hostname
harpo
$
```

If you enter the `hostname` command, it shows the name of the computer on which you're working. Here it printed `harpo`, the name of my computer, and re-displayed the prompt. The shell continues to run until you give it a command to terminate itself, such as `exit`.

In Unix, a shell is not just a command line interpreter. It's also a programming language interpreter; you can use it to define variables, evaluate expressions, perform I/O, use conditional control-of-flow statements such as loops and branching statements, define and call functions, and much more. In short, it has most of the features of a high-level programming language such as C. You can save a sequence of shell commands into a file to be executed at another time. Such a file is called a *shell script*. You can arrange for the shell to execute these shell scripts in a few different ways.

Most shells also implement various frequently used commands as functions inside the shell itself, which are called *shell builtins* or just *builtins*. Building a command directly into the shell speeds up its execution because calling a function takes much less time than starting a separate program, which requires kernel intervention.

In a typical Unix system, you can choose which shell you'd like to use from among several different shells, depending on your preferences.

The oldest of the most commonly distributed shells, which was part of Seventh Edition UNIX (released in 1979 by Bell Labs), is known as the *Bourne shell*, so named because it was written by Stephen Bourne [1]. The name of the shell program was `sh`, which is what you had to enter to run it. It was the first extension to the original UNIX shell, written by Ken Thompson. The Bourne shell is important because it is always a part of any Unix distribution and many administrative scripts are written in it, requiring that it's installed. Some commands will fail if it isn't found on the system.

Other common shells that have been around a long time include the *C-shell* (`csh`) and the *Korn shell* (`ksh`).

However, the most commonly used shell in GNU/Linux systems is the *Bourne Again SHell*, whose program name is `bash`, and that is the shell we'll use in this book. The GNU Project created `bash` by extending the Bourne shell with features from the Korn shell and the C shell (*https://www.gnu.org/software/bash/*).

### Users and Groups

Historically, *users* in Unix were people who were given access to the system and could run programs and own files. Part of the security of Unix rests on the principle that every user of the system must be authenticated. *Authentication* is a form of security clearance, like showing an ID card before entering a building or passing through a scanner at an airport.

The Unix method of authentication gives every user a unique *username* and an associated unique, non-negative integer *user ID*, or *UID* for short. The username is the name a person enters to *log in* to the system. Each user also has an associated password. Unix uses the username/password pair to authenticate a user attempting to log in. If the username does not exist or

the password doesn't match it, the system rejects the user. System files store passwords in an encrypted form.

**LOGGING IN**

To *log in* to a system is to *log* into it. One of the dictionary meanings of the verb *to log* that existed long before computers did, is to record something in a logbook, as a sea captain or airplane pilot does. The term *login* conveys the idea that the action is being recorded in a logbook. In Unix, logins are recorded in a file that acts like a logbook. The system maintains a list of names of users who are allowed to log in. We take this term for granted. We use the noun *login* as a single word only because it has become a single word on millions of login screens around the world. To log in, as a verb, really means, to log *into* something; it requires an indirect object.

To be precise, in modern Unix systems, a user is any entity that can run programs and own files. This entity need not be an actual person. For various reasons, the definition of a user was generalized to allow abstract entities as well as programs to be users as well. For example, `root`, `syslog`, and `lp` are each non-person users.

A *group* is a set of users. Just as each user has a username and user ID, each group has a unique *group name* and an associated unique, non-negative integer *group ID*, or *GID* for short. Unix uses groups to provide a means of resource sharing. For example, a file can be associated with a group, and all users in that group would have the same access rights to that file. Since a program is just an executable file, the same is true of programs; an executable program can be associated with a group so that all members of that group will have the same right to run that program.

Every user belongs to at least one group, called the user's *primary group*. You can use the `id` command to print your username and user ID, and the group name and group ID of all groups to which you belong:

```
$ id
uid=500(stewart) gid=500(stewart) groups=500(stewart),4(adm),24(cdrom),27(sudo)
```

In fact, you can supply `id` with any username, and it will list their information:

```
$ id syslog
uid=102(syslog) gid=106(syslog) groups=106(syslog),4(adm),5(tty)
```

Alternatively, you can use the `groups` command to print a list of groups to which you (or another user) belongs:

```
$ groups
stewart adm cdrom sudo
$ groups syslog
syslog : syslog adm tty
```

In Unix, the *superuser* is a distinguished user whose username is (usually) *root* and whose UID is 0. The superuser can perform actions that ordinary users cannot, such as changing a person's username or modifying the operating system's configuration. Anyone who can log in as *root* in Unix has absolute power over that system. For this reason, most Unix systems record every attempt to log in as *root*, so that a system administrator can monitor and catch break-in attempts.

### Privileged and Non-Privileged Instructions

In order to prevent ordinary users and their programs from accessing hardware and performing other operations that may corrupt the state of the computer system, Unix requires that the processor support two modes of operation, known as *privileged* and *unprivileged* mode. These modes are also known as *supervisor mode* and *user mode*, respectively. *Privileged instructions* are instructions that can alter system resources, directly or indirectly. Examples of privileged instructions include:

- Acquiring more memory
- Changing the system time
- Raising the priority of the running process
- Reading from or writing to the disk
- Entering privileged mode

*Only the kernel is allowed to execute privileged instructions.* Programs run by ordinary users can execute only unprivileged instructions. The security, reliability, and integrity of the operating system depend upon this separation of powers.

### Environments

When a program is run in Unix, one of the steps that the kernel takes prior to running the program is to make available to it an array of name-value pairs called the *environment list*, or simply the *environment*. Each name-value pair in this list is a string of the form *`name=value`* where *`value`* is a `NULL`-terminated C string, and there are no spaces around the `=` character. The *`name`* is called an *environment variable* and *`name=value`* is called an *environment string*. For example

```
LOGNAME=stewart
```

is an environment string that specifies that the variable named `LOGNAME` has the value `stewart`. Variable names are not allowed to contain the `=` character, but otherwise they have no restrictions. However, for portability of any programs that use these variables, and by convention, they should contain only uppercase letters, digits, and underscores and should not begin with a digit (see *The Open Group Base Specifications*, Issue 7, 2018, Chapter 8 [17]).

A more interesting example is the `COLUMNS` environment variable, which stores the number of columns in the currently open terminal window. If you assign a value to `COLUMNS` in a startup `bash` script, terminal windows will have that number of columns, overriding the rules that are used to size a terminal for the data to be displayed. (Generally you should not do this.) By setting `COLUMNS` to the null string, as in

```
COLUMNS=
```

the operating system determines the size of terminal windows. Even though it's a number, the value of `COLUMNS` is stored as a string.

Environment variables can influence the behavior of many programs, including the shell itself. When you log in to a Unix system, the operating system creates the environment for you, using configuration information from various files in the system. From that point forward, whenever you run a program, it inherits a copy of the current values of the environment. That program can use the environment variables to customize its behavior, and it can also modify its own copy of the environment. In Chapter 2, I explain how the environment is passed to a program, how it affects the behavior of the shell, and how you can customize it. In Chapter 11, I explain in detail how the environment is represented and where it is stored in memory when a program is running.

You can see the values of environment variables from the command line in various ways. The `printenv` command displays the values of all environment variables, as does the `env` command. Both may produce more lines than one screen can display. Soon you'll see how to *page* output one screenful at a time. If you want to see the values of selected environment variables, give their names as arguments to the `printenv` command:

```
$ printenv LINES COLUMNS SHELL
23
80
/bin/bash
```

A program can call the `getenv()` function to retrieve a particular environment string. To demonstrate, the following small program, named *getenv_demo.c*, prints out the number of columns in the terminal window in which it runs:

*getenv_demo.c*

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    char* num_columns = getenv("COLUMNS");
    printf("This window has %s columns.\n", num_columns);
}
```

The program needs to include the *stdio.h* header file because it calls the `printf()` function, and the *stdlib.h* header file because it calls `getenv()`, which is declared in that header. We compile it and run it as follows:

```
$ gcc getenv_demo.c  -o print_numcols
$ ./print_numcols
This window has 80 columns.
```

This is a *sneak preview* of how we compile code using the GNU `gcc` compiler. We give `gcc` the name of the source code file, *getenv_demo.c*, and the option `-o print_numcols` to store the *output* of the compiler in the executable file named *print_numcols*. Without that option, it would store the executable in a file named *a.out*. In the next chapter we'll explain thoroughly the process of building executable code.

## Files, Directories, and the Single Directory Hierarchy

In their seminal article, "The UNIX Time-Sharing System," Ritchie and Thompson stated that the single most important role of the operating system is to provide a filesystem [24]. Kernighan and Pike, in their now-famous book on programming in the Unix environment, *The UNIX Programming Environment* [11], point out that the very first aspect of Unix that Ritchie and Thompson discussed while designing the system was the structure of its system of files, because that determined how everything else was going to work, and they went so far as to state that "everything in the UNIX system is a file."

### Files

For most people who use computers, files are simply objects that store information. These objects reside on *non-volatile storage* devices, which are storage devices that retain data even when power is not applied to them, such as magnetic, optical, and electronic disks, and magnetic tapes. (In contrast, *volatile storage*, such as main memory, does not retain data when it is powered off.) These non-volatile storage devices are called *secondary storage* devices or *external storage* devices, even though they might appear to you to be "inside" the computer. The nomenclature is a historical artifact.

In many non-Unix systems, the operating system recognizes different types of files, each having its own specific structure, such as word-processor documents, image files, or spreadsheets. In fact, in those systems, files often have names or extensions that can be used to infer their structure or even cause a specific program to load them.

In Unix, however, the story is very different. From the kernel's viewpoint, an ordinary file is just an object that contains a linear sequence of bytes. It does not impose any structure on the contents of this kind of file; any structure that it might have is given to it by the user or program that creates it. These files are called *regular* or *plain* files. Some of these files are what we commonly call *text files*, because when we open them we see plain text. These files contain sequences of characters with lines demarcated by newline characters; programs that are designed to display them use the em-

bedded newline characters to create the line structure on the screen. *Binary files*, in contrast, are files that contain byte sequences that are not necessarily text characters, such as a program's executable code.

### File Types

The Unix kernel does define a small set of file types other than these regular files:

- Directories
- Device files
- Pipes
- Sockets
- Symbolic links

Directories are described in "Directories" on page 19. Device files, pipes, and sockets are collectively called *special files*. Special files are an unusual feature of the Unix system of files. They were invented to provide a method of programming I/O in a device-independent way. Chapter 18 covers device files, pipes, and device-independent I/O; Chapter 20 covers sockets. I define and discuss symbolic links in "Symbolic Links" on page 24.

### File Attributes, Permissions, and Contents

All files, regardless of their type, have *attributes*. Attributes include all of the important information about the file, such as the time the file was last modified, the time it was last accessed, the user ID of its owner, its size expressed as a number of bytes, who is allowed various types of access to the file, and so on. The attributes that describe restrictions on access to the file are called the *file mode* or the file's *permissions*. Permissions play an important role in the security of a Unix system. We'll explore them in detail in "File Permissions" on page 75.

The attributes of a file collectively are called the *file status*. The word *status* may sound misleading, but it's the word that was used by Ritchie and Thompson in the original UNIX system. Another word often used to describe a file's attributes or status is *metadata*. Unix systems make a clear distinction between the *contents* of a file and its status. Contents are a file's data; most, but not all, files have contents. Some files, such as device files and certain other special files, do not have contents—they do not store data. They are interfaces that the kernel uses to implement device-independent input and output.

The contents of a file don't contain any status information. They have no end-of-file characters to denote the end of the file, for example, or any other means of representing its length. The contents and status aren't even stored together. The status is stored in a data structure called an *inode*, whereas the contents may be spread out in multiple blocks on the same storage device as the inode.

An important fact about files is that *filenames are not part of the status of the file.* In fact, a non-directory file can have multiple names, and those

names aren't an inherent property of the file itself but of the directories that contain them..

### Directories

A directory, often called a *folder* in other operating systems, is a type of file that, from the user's perspective, appears to contain other files. We tend to visualize them as shown in Figure 1-4.

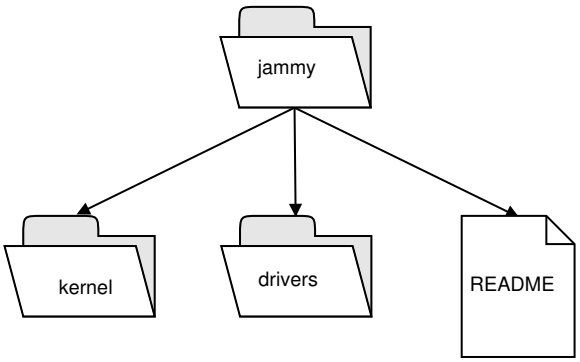



*Figure 1-4: Directory with three children*

This is only an illusion; directories don't contain files any more than the table of contents contains the chapters of the book. What then is a directory?

To be precise, a directory is a file that contains a table of *directory entries*, which are properly called *links*. A *link* is an object that associates a *filename* to an actual file. It has two components: the filename and a reference to a file's inode. The links may reference any type of file, including directories, implying that directories can be members of directories. However, a link isn't allowed to refer to a file that's on a different device from the directory itself.

Directories are never empty because every directory contains two links, named `.` (*dot*) and `..` (*dot-dot*). These entries have a predefined meaning: `.` is a link to the directory itself, and `..` is a link to the directory containing this directory, which is called the *parent* directory. Figure 1-5 shows what the actual directory table for the directory named *jammy* in Figure 1-4 looks like.

jammy directory

| Reference to file | Name |
|---|---|
| 53 | . |
| 2 | .. |
| 12 | kernel |
| 185 | drivers |
| 282 | README |

*Figure 1-5: Table for jammy directory*

The numbers in the left-hand column are just illustrative and are supposed to represent references to the inodes for the given files. For example, `drivers` is the name in this directory for the file whose inode has number `185`.

When you're working in a shell, it maintains for you a unique directory called the current working directory. The *current working directory* is the directory in which you are working. The idea of being "in a directory" deserves clarification.

We often say when speaking out loud about a computing session that we're "in a directory." Give a moment's thought to this statement. What does it actually mean? It's more intuitive when you work in a graphical user interface and the file browser displays a window whose contents are the files inside a single directory. In this case, the directory whose files are in that window is the directory in which you're currently working. The same thing is true in a command line interface; you have a unique working directory.

Two directory-related commands will demonstrate these ideas. The `ls` command can display the contents of directories. Entering `ls` without arguments displays the contents of the current working directory:

```
$ ls
chapters/    fonts/     images/    main.tex      main.bib
```

Alternatively, we can give `ls` one or more directory names as its arguments to see their contents:

```
$ ls chapters  images
chapters:
appendix_a.tex   chapter_02.tex     chapter_05.tex       preface.tex
back_matter.tex  chapter_03.tex     front_matter.tex
chapter_01.tex   chapter_04.tex     intro.tex

images:
chapter_01/  chapter_2/  chapter_3/  chapter_4/     chapter_5/
```

Notice that each directory's name appears first, followed by the files that are in that directory. The number of columns that `ls` uses is based on how many names the directory has and their lengths.

We can change the current working directory with the `cd` command:

```
$ cd chapters
$ ls
appendix_a.tex   chapter_02.tex     chapter_05.tex       preface.tex
back_matter.tex  chapter_03.tex     front_matter.tex
chapter_01.tex   chapter_04.tex     intro.tex
```

Notice that now the `ls` command displays the contents of the new working directory, which is *chapters*. We can return to the previous directory via the .. link:

```
$ cd ..
$ ls
chapters/    fonts/     images/    main.tex      main.bib
```

and you can see from the output of `ls` that the working directory is once again the parent of *chapters*, since the list of filenames is the same as it was before we changed directory to *chapters*.

### Filenames

Files and filenames, as noted earlier, are different things. A *filename* is a string that names a file. It is part of the link contained inside a directory. A single non-directory file may have names in different directories (on the same logical device) and can therefore appear to be a member of many directories. However, files exist independently of the directories in which they appear. If the same file has names in different directories, the references associated to those names in the links all point to the exact same inode, namely the unique inode for that file. It's like a person traveling with several passports. The passports might have different names for the person and be used in different countries, but they each represent the same person. Figure 1-6 illustrates this idea.

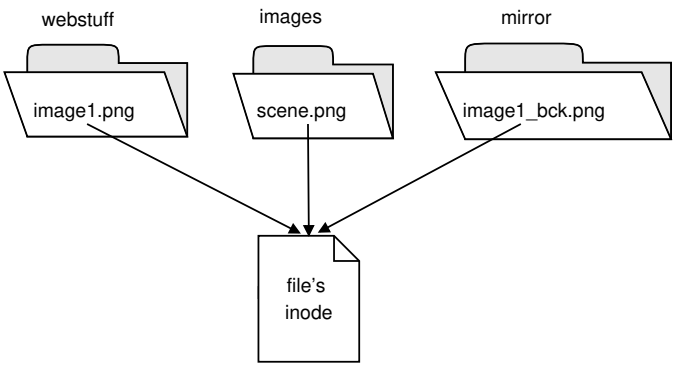



*Figure 1-6: A file with three names*

It shows one file that's known by three different names, each a link in a different directory.

Filenames are allowed to be quite long. The maximum number of characters in a filename is defined by a system-dependent constant `NAME_MAX`, which is usually 255 characters. They can contain almost any character except `/` and the null character `\0`, but you shouldn't use certain characters in filenames even if they're allowed. For example, a filename can have spaces and new lines, but if it does, you'll usually need to put quotes around the name to use it as an argument to commands. Certain characters, such as `$`, `&`, `*`, and others, have a distinct meaning to various programs and must be *escaped* by preceding them with a backslash if they're used in those contexts, so it's best to avoid them. The convention is to use only alphanumeric characters, the underscore, and the hyphen in filenames. Unix is *case-sensitive*, so that *source* and *Source* would be treated as two different filenames.

Unlike most other operating systems, Unix doesn't use filename extensions for any purpose, although application-level software such as compilers and word processors might use them as guides. Desktop environments such as GNOME and KDE can create associations based on filename extensions in much the same way that Windows and macOS do, but Unix itself doesn't have a notion of file type based on content, and it provides the same set of operations for all files, regardless of their type. In Unix, we use the word *suffix* for the part of a filename after a period, such as the *c* in *myprog.c*.

### The Directory Hierarchy

Unix organizes files into a tree-like hierarchy that most people erroneously call the *filesystem*. It's more accurately called the *directory hierarchy*, because the term *filesystem* refers to a set of data structures written onto an unstructured disk device to enable the creation and management of files and directories.

Each node in this tree-like hierarchy is either a non-directory file or a directory. Each edge is a *directed edge* from a non-empty directory to each file that is contained in that directory, including files that are directories, and we call the contained files the *children* or *child nodes* of that directory. The directory is called the *parent* of those child nodes. This hierarchy's base is a single *root* directory whose name is the `/` character. Even though the base is named `/`, when people refer to this directory, they usually call it the *root directory*, since saying *forward slash* is not very descriptive and is also a mouthful.

Because a single file can have names in different directories, a file may have more than one parent node, as shown in Figure 1-6. This is why the hierarchy is *tree-like* but not a tree, since in a tree every node has a unique parent.

In a typical, modern Unix system, the directory hierarchy is a *directed acyclic graph*, which is a directed graph that contains no cycles. It has no cycles because a directory, unlike a non-directory file, can't have more than one name, which implies that it's an entry in exactly one parent directory. This implies that no edge can be pointing to it from any descendant node, and hence the graph has no cycles. Some Unix implementations do allow

the superuser to give directories more than one name, in which case, it is possible for the hierarchy to have cycles.

This idea of a single directory hierarchy is a defining characteristic of Unix; other operating systems, such as *Microsoft Windows* have separate directory hierarchies for each distinct device. In Unix, even though the files in this single tree might be on different devices, the directory hierarchy on any device can be attached to the single tree by a procedure called *mounting*. After that hierarchy is mounted on the tree, its files can be accessed in the same way as all other files.

The typical Unix directory hierarchy has several directories just under the root. These directories are called the *top-level directories*. Figure 1-7 illustrates a portion of the top of a typical directory hierarchy.

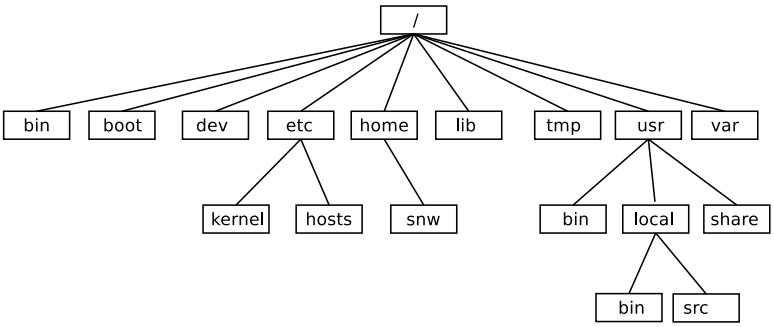



*Figure 1-7: A portion of the top of a typical UNIX directory hierarchy.*

The following list describes the top-level directories present in most Unix systems. The only directories actually required by POSIX.1-2017 are */dev* and */tmp*.

***bin*** All essential binary executables including those shell commands that must be available when the computer is running in single-user mode (something like safe mode in Windows).

***boot*** Static files of the bootloader.

***dev*** Essential device files (covered in Chapter 14).

***etc*** Almost all host configuration files, roughly like the registry file of Windows.

***home*** If present, all users' home directories.

***lib*** Essential shared libraries and kernel modules.

***media*** Mount point for removable media.

***mnt*** Mount point for mounting a filesystem temporarily.

***opt*** Add-on application software packages.

***sbin*** Essential system binaries.

***srv*** Data for services provided by this system.

***tmp*** Temporary files created by applications.

***usr*** Originally, this was the top of the hierarchy of user data files, but now it's the top of a hierarchy containing non-essential binaries, libraries, and sources. Typical subdirectories are */usr/bin* and */usr/sbin*, which contain binaries; */usr/lib*, containing library files; and */usr/local*, the top of a third level of local programs and data.

***var*** *Variable* files, meaning files whose contents can change.

All files, including directories, can be characterized by two independent binary properties: their shareability and their variability. *Shareable* files can be stored on one host and used on others. *Unshareable* files aren't shareable. For example, the files in user home directories are shareable because they don't depend on where they are stored, whereas bootloader files are specific to a given machine and aren't shareable.

*Variable* files are files whose data can change. *Static* files are files whose contents don't change. They include, for example, executable binaries, libraries, documentation files, and other files that don't normally change in the day-to-day operation of the computer. In modern Unix systems, the shareability and variability of files is a factor in deciding which ones are in which parts of the hierarchy. Files that differ in either of these attributes are placed into different directories, which makes it easy to store files with different usage characteristics on different filesystems and also makes backing up easier. For example, the */etc* directory is unshareable—it contains files specific to the particular computer—and it's static, because its contents are configuration files that are modified only when we apply updates, install new software, or the superuser decides to change configurations. The */var* directory is so named because it is variable. It contains many different types of logfiles that the kernel and applications update on a regular basis. Some of its subdirectories, such as */var/mail*, may be shareable, whereas others such as */var/log* may be unshareable. The */usr* directory is shareable and static. It contains application binaries, libraries, and static data.

### Symbolic Links

An ordinary link is a directory entry that points to the inode for a file, but a *symbolic link* is a file whose contents are just the name of another file. The file to which the link points is called the *target* of the link. The inode for a symbolic link identifies that file as a symbolic link. It's similar to a *shortcut* in the Windows operating systems. Symbolic links are often called *soft links* in contrast to ordinary links, which are called *hard links*.

Usually, commands, programs, and the kernel itself, when they are given a symbolic link when a filename is expected, will operate on the target of the link, not the link itself. They can easily see that the file is a symbolic link because the inode indicates it. We say that a link is *dereferenced* or is *followed* when the link is opened to access its target.

Symbolic links pose hazards for the operating system and applications because of the possibility of circular references and infinite loops. The danger is that a symbolic link can point to a directory, which means that if a program follows symbolic links, it might return to a directory that it already vis-

ited and end up in a cycle. Chapters 7 and 8 address issues related to symbolic links in more detail.

#### Pathnames

A *pathname* is a character string that identifies a file. There are two types of pathnames: absolute and relative. An *absolute* pathname starts at the root of the directory hierarchy and starts with a leading forward slash `/`. Zero or more filenames separated by slashes follow that leading slash, such as */data/jammy/kernel/sched/sched.h*. All filenames except the last must be directory names or symbolic links whose targets are directory names. Each of the names in the example pathname except *sched.h* is a directory. The last name in the path may be any type of file. Other examples of absolute pathnames are */usr/bin/*, */usr/local/share/man*, and */home/stewart/unixbook/figures/figure01.png*

Terminating a pathname with a slash is acceptable if the last filename in it is a directory, as in the pathname */usr/bin/*.

If you accidentally insert more than one slash between the names in the path, it will be ignored. The two absolute pathnames */usr/local/share/man* and */usr/local///share/man* are the same.

If a pathname doesn't start with a leading slash, it's called a *relative pathname*. A relative pathname starts in the current working directory, which we can now accurately define. The *current working directory* (also called the *present working directory*) is the directory that any running program uses to resolve pathnames that do not begin with a `/`. For example, if the current working directory is */home/stewart/unix_book*, the pathname *chapters/chapter_01* refers to a file whose absolute pathname is */home/stewart/unix_book/chapters/chapter_01*.

The environment variable `PWD` contains the absolute pathname of the current working directory. The `pwd` command prints the value of `PWD`:

```
$ pwd
/home/stewart/unix_book
$ printenv PWD
/home/stewart/unix_book
```

Pathnames can become very long if they contain symbolic links, and Unix systems limit their length, expressed in bytes. POSIX.1-2017 specifies that the constant `PATH_MAX` is the maximum number of bytes allowed in a pathname, including the terminating null byte. On many Linux systems, it is 4096 bytes.

### Processes

People (and sometimes programs) write programs. Programs are sequences of instructions to the computer, written in a programming language. The language might be a high-level one, such as C or C++, or it might be a low-level one, such as an assembly language. In general, programs can't be executed in the form in which they're written; they must be translated into an

executable form. The exceptions to this are programs written in scripting languages, such as *JavaScript*, *PHP*, and *Basic*. These aren't translated into an executable; an interpreter program reads the source code directly and executes their instructions one after another.

We call the first form of a program the *source code*, and the second form, the *executable code*, or simply, the *executable*. For example, the source file *hello_world.c* from Listing 1-1, is a human-readable text file. You can use the GNU C compiler to build an executable from it named `hello_world` with the command

```
$ gcc  hello_world.c  -o hello_world
```

The file *hello_world* will be an executable file residing, by default, in the same directory as *hello_world.c*. You can't use ordinary text editors to see or modify the contents of this file because it's not plaintext; it's a binary file.

Perhaps surprisingly, even running a program is a complex procedure (we'll cover the details in Chapter 12). The executable form of most programs isn't something we can actually run. We can't just load it into memory and tell the machine to start running that file from its first byte. That file is usually a conglomeration of executable code, various tables, and instructions to a linker and loader. When you enter the command

```
$ ./hello_world
```

a sequence of actions takes place that causes a *loader* to use the information in that *hello_world* executable to load the file, as well as any shared objects that it needs, into memory, and to transfer control to a *dynamic linker*, which runs that program.

Many users can run a single program at the same time on a given machine, or a single user can run one multiple times in different terminal windows. Either way, it means that one executable can have many running instances, which is what leads us to distinguish between programs and processes. A *process* is a running instance of a program. Each separate running instance is a different process, although each and every one of them is executing the exact same executable file.

This formal definition of a process doesn't really tell you what a process is in concrete terms, even though it's the one you'll likely see in an operating systems textbook. It's like defining a baseball game as an instance of the implementation of the set of rules created by Alexander Cartwright in 1845 by which two teams compete against each other on a playing field. Neither definition gives you a mental picture of what's being defined. Let's make it more concrete.

When a program is run on a computer, it uses various resources such as primary memory, secondary storage space, kernel memory for mappings and tables of various kinds, such as a table of which parts of primary memory it uses, privileges such as the right to read or write certain files or devices, and much, much more. As a result, at any instant of time, a process is associated with the collection of all resources allocated to that instance of the program it executes, as well as any other properties and settings that

characterize that instance, such as the values of the processor's registers. Thus, although the idea of a process sounds like an abstract idea, it is, in fact, a very concrete thing, and an operating system must manage it.

Unix systems assign to each process a unique non-negative integer called its *process identifier* or *PID* for short. We can learn a bit about processes using the `ps` command, which can display a list of running processes, as well as selected information about each of them. It has various options to control which processes it displays and what information it outputs. In its simplest form, with no options, we can use it to see the PIDs of our own running processes:

```
$ ps
    PID TTY          TIME CMD
  10278 pts/0    00:00:00 bash
  11087 pts/0    00:00:00 ps
```

This lists two processes, one running `bash` and the other running the `ps` command itself. They use so little time that it shows up as zeros, and their respective PIDs are 10278 and 11087. They're both running in a terminal whose device name is `pts/0`.

At the programming language level, we can call the `getpid()` function to obtain the PID of the process that invokes it. We demonstrate this in the *getpid_demo.c* program:

*getpid_demo.c*

```
#include <stdio.h>
❶ #include <unistd.h>
void main()
{
 ❷ printf("I am the process with process-id %d\n", getpid());
}
```

All this program does is print its own PID, but it illustrates how to use `getpid()`. The program includes the header file `<unistd.h>` ❶ because the `getpid()` function, called inside the argument list of `printf()` ❷, is a system call, and almost all system call declarations are in `<unistd.h>`. This is our first program to make a system call.

The return value of `getpid()` is the PID of the process that calls it. Because PIDs are integers, in the format string of `printf()`, we use the `%d` format specification to print the return value as a fixed decimal numeral. Assuming that *getpid_demo.c* is in our working directory, we can compile and run it with these commands:

```
$  gcc getpid_demo.c -o getpid_demo
$ ./getpid_demo
I am the process with process-id 18805
```

If we were to run this same program again, it would print a different PID, proving a new process is created whenever it is run.

### Threads

The programs that we've described so far in this chapter are assumed to have a single thread of control. A *thread of control* is a single sequence of instructions that's executed one instruction at a time, one after the other, during the execution of a program. Originally, all programs had a single thread of control. As the cost of computer processors became smaller and smaller, hardware vendors started building computers containing multiple processors, and computer scientists sought ways to take advantage of this new technology. They designed and created programming languages and libraries that would allow a program to contain more than one thread of control, each of which could run on the separate processors simultaneously. These threads of control were named *threads* for simplicity.

POSIX.1-2017 formally defines a *thread* as a single flow of control through a process together with the required system resources to support a flow of control [17].

The traditional Unix process is a single thread, but in modern operating systems, processes in general can have multiple threads. When a process has multiple threads, it's called a *multithreaded process*. A multithreaded process has two types of resources: those that are shared among all of its threads, which are generally called *global* or *shared*, and those that are unique to each thread, commonly called either *thread local*, *private*, or *per-thread*. In Chapter 13 we detail exactly which process resources are shared and which are thread local.

Unix systems in general support multithreading, and Linux in particular supports several different types of threads. Linux handles threads in an interesting way; it treats all threads as standard processes. It doesn't provide any special scheduling or data structures for threads. To the Linux kernel, processes and threads are both called *tasks* and are both represented internally by the same data structure, called a `task_struct` (see [2]). In Linux, a *task* is an entity that's assigned system resources and can be scheduled on a processor. The difference between threads and ordinary processes in Linux is that threads can share resources, such as their address space, whereas processes don't share any resources.

In many Unix implementations, a thread has a *thread identifier (TID)* that is unique in the operating system, but POSIX.1-2017 doesn't require this. It only requires that within a single process, each thread's TID is unique. Linux handles TIDs with a two-pronged approach: in a single-threaded process, the TID is equal to the process ID, whereas in a multithreaded process, all threads have the same PID, but each one has a unique TID. In Linux, a thread can call the `gettid()` function to obtain its thread ID. The *gettid_demo.c* program demonstrates this idea:

*gettid_demo.c*

```
#define _GNU_SOURCE ❶
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
```

```
void main()
{
    printf("I am a thread with thread  ID %d\n", gettid());
}
```

The program uses the C preprocessor `#define` directive to define the symbol `_GNU_SOURCE` ❶. Unless this symbol is defined, the compiler won't see the various declarations in the header files that are needed for the program to call `gettid()`. This is an example of a *feature test macro*, which is explained in "Portability" in Chapter 3. The `#define` directive must appear before all include directives. We can compile and run it as shown in the following sample session:

```
$ gcc gettid_demo.c  -o gettid_demo
$ gettid_demo
I am a thread with thread ID 1810
```

If we run this program again, it too will display a different TID each time for the same reasons as before—a new process runs and its TID is the same as its PID when it has one thread.

## Online Documentation

Unix systems provide several different types of *online documentation*. In this context, *online* means on the computer that you are using, not on the World Wide Web.

### The Man Pages

In 1971, shortly after the release of First Edition UNIX, Dennis Ritchie and Ken Thompson, with help from Joseph Ossanna and Robert Morris, wrote the first *UNIX Programmer's Manual*, which is still available online (*https://www.bell-labs.com/usr/dmr/www/manintro.html*). This manual was initially a single volume, but in short order it grew into a set of seven volumes, organized by topic. It was available in both printed form and as formatted files suitable for display on an ordinary character display device. Over time it grew in size. Every Unix distribution now comes with this set of manual pages, called *man pages* for short. The manual has eight numbered sections in a typical Unix system as of this writing, as shown in Table 1-1.

**Table 1-1:** Manual Sections

| Number | Common name | Description |
|---|---|---|
| 1 | User commands | Executable programs and shell commands |
| 2 | System calls | Functions provided by the kernel |
| 3 | Library calls | Functions within program libraries |
| 4 | Special files | Files usually found in */dev* |
| 5 | File formats and conventions | Formats of system files |
| 6 | Games | Various games and humorous programs |
| 7 | Miscellaneous | Macro packages and conventions |
| 8 | System administration commands | Usually only for root |

The man pages are an important part of Unix documentation. They act as an online reference when you want to learn about any part of the Unix system, such as a command, a function from one of the libraries, a system call, a device interface, a system file, or various file formats, and much more. Although the documentation is very thorough and detailed, it's usually not tutorial in nature, and it can be overwhelming sometimes, but many pages have code examples that you can compile, modify, and run.

Over the years in which I taught Unix system programming, students would sometimes say that they didn't need to learn how to use the man pages because all that information is online and they just had to "Google" it. It's true that you can find copies of the man pages on many websites and read posts on discussion boards, but the reasons for reading the man pages on your own Unix installation go beyond this:

- The versions of the man pages on your system were installed at the time that the software they document was installed, and they are updated whenever you update the software itself, and the software has updates to apply to them.
- Man pages are written by the people who wrote and maintain the software and are trustworthy and accurate.
- The man pages on your system are self-contained in the sense that any cross references they make are also on your system.
- You can read them even if your internet connection isn't available.

To view the man page for a given topic, enter **`man`** followed by the topic in which you're interested, meaning the command name, function name, and so on. For example, enter **`man man`** to read the man page for the `man` command itself:

```
$ man man
MAN(1)                        Manual pager utils                        MAN(1)

NAME
       man - an interface to the system reference manuals
```

```
--snip--
```

The output is just the first few lines of that page. The first line shows that the `man` command is in Section 1 of the man pages because the title contains `MAN(1)`. The text `Manual pager utils` is not the name of Section 1; we'll call it the *man page header* or the *header* when the meaning is clear. Different man pages in Section 1 may have different headers. After the word `NAME` is the name of the command followed by a very brief description of what the command does. This is the very first man page you should read, and we'll revisit it shortly.

All POSIX-conforming Unix systems are required to contain man pages for all of the header files that might be included by a function in the kernel's API. To put it more precisely, each function in the System Interfaces volume of POSIX.1-2017 specifies the headers that an application must include to use that function, and a POSIX-conforming system must have a man page for each of those headers. They may not be installed on the system you're using, but they're available. They're installed only if the system administrator installed the application development files.

The man page for the `scanf()` function starts with the following lines:

```
SCANF(3)                  Linux Programmer's Manual                 SCANF(3)
NAME
       scanf,  fscanf, sscanf, vscanf, vsscanf, vfscanf
       - input format conversion

SYNOPSIS
       #include <stdio.h>
--snip--
```

It tells us that we need the header file *stdio.h* to use `scanf()`. We can enter **`man stdio.h`** to read about that header file, which outputs the following:

```
stdio.h(7POSIX)           POSIX Programmer's Manual          stdio.h(7POSIX)

PROLOG
       This  manual  page is part of the POSIX Programmer's Manual.  The Linux
       implementation of this interface may differ (consult the  corresponding
       Linux  manual page for details of Linux behavior), or the interface may
       not be implemented on Linux.

NAME
       stdio.h — standard buffered input/output
--snip--
```

One challenge with using the man pages is that you need to know the name of the command or function in which you're interested for them to be of help. The man pages do have a relatively simple search mechanism, but they are really intended as a reference manual for people who already have a

sense of what it is they need to look up, so if you know what you want to do but don't know the command name, the challenge is how to find it.

The man pages play a key role in helping you solve problems on your own. My method of teaching how to write system programs is based on using the man pages to guide the learning process. They're inextricably linked to learning system programming in this book, so I've included a separate section, "Using the Manual Pages," that explains their structure and how to use them in greater depth, including the syntax they use for specifying options and arguments.

### The Info Documentation System

Because of some deficiencies in the man pages, the GNU project developed an alternative documentation system named *Info*, which it based on the *Texinfo* documentation system. Texinfo (pronounced like "Tekinfo"), is a documentation system that uses a single source file to produce both online and printed output. It's based on a help system that Richard M. Stallman created for the Emacs text editor in 1975 and 1976 (*https://www.gnu.org/software/texinfo/manual/texinfo/html_node/History.html*).

The Info pages for various commands and utility programs sometimes contain much more information than their man page counterparts. In some cases, the man page for a command refers the reader to the Info page. To read an Info page, enter the `info` command (lowercase). For example, to learn about the `ls` command, enter **`info ls`**:

```
$ info ls
Next: dir invocation,  Up: Directory listing

10.1 'ls': List directory contents
==================================

The 'ls' program lists information about files (of any type, including
directories). Options and file arguments can be intermixed arbitrarily,
as usual.
--snip--
```

When there isn't a page for a particular topic in the Info system, the Info reader opens up the man page for that topic instead.

The Info pages use a method of navigation similar to the one in Emacs, which people often find hard to use. There's a method of reading an Info document and bypassing the navigation in it by *piping* its output into a *pager* program such as `more` or `less`, as shown here:

```
$ info ls | more
File: coreutils.info, Node: ls invocation,  Next: dir invocation, Up: Directory
listing

10.1 'ls': List directory contents
==================================
```

```
The 'ls' program lists information about files (of any type, including
directories). Options and file arguments can be intermixed arbitrarily,
as usual.

--snip--
```

The same information is displayed, but it also mentions the file in which it's contained, *coreutils.info*. We'll explain how this works and what pagers are in "The Pager" on page 34.

### Application Provided Documentation

Sometimes you can also find information about a particular application or program in one of the directories in */usr/share/doc*. Many applications and higher level program installers place their documentation there. This documentation sometimes includes extensive usage examples, development notes, and hints on where to find further information.

Some commands have a means of displaying their own help, usually by providing an option such as `--help`:

```
$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

--snip--
```

The rest of the output is primarily a description of the various options of `ls` and their arguments.

### Shell Help

Certain shells have a help feature for commands that are built into the shell. In particular, `bash` has a `help` command, which when entered without arguments prints a two-column list of all `bash` built-ins with options and arguments listed:

```
$ help
GNU bash, version 5.1.16(1)-release (x86_64-pc-linux-gnu)
These shell commands are defined internally. Type `help' to see this list.
Type `help name' to find out more about the function `name'.
Use `info bash' to find out more about the shell in general.
Use `man -k' or `info' to find out more about commands not in this list.

A star (*) next to a name means that the command is disabled.

 job_spec [&]                            history [-c] [-d offset] [n] or hist>
 (( expression ))                        if COMMANDS; then COMMANDS; [ elif C>
 . filename [arguments]                  jobs [-lnprs] [jobspec ...] or jobs >
```

```
:                                       kill [-s sigspec | -n signum | -sigs>
[ arg... ]                              let arg [arg ...]
[[ expression ]]                        local [option] name[=value] ...
alias [-p] [name[=value] ... ]          logout [n]
bg [job_spec ...]                       mapfile [-d delim] [-n count] [-O or>
bind [-lpsvPSVX] [-m keymap] [-f file>  popd [-n] [+N | -N]
break [n]                               printf [-v var] format [arguments]
builtin [shell-builtin [arg ...]]       pushd [-n] [+N | -N | dir]
--snip--
```

When given the name of a particular `bash` built-in, it prints a short summary of how to use that command:

```
$ help pwd
pwd: pwd [-LP]
    Print the name of the current working directory.

    Options:
      -L print the value of $PWD if it names the current working
      directory
      -P print the physical directory, without any symbolic links

    By default, `pwd' behaves as if `-L' were specified.

    Exit Status:
    Returns 0 unless an invalid option is given or the current directory
    cannot be read.
$
```

The `help` command uses the same syntax as the man pages.

#### Other Sources of Documentation

You can download many manuals from the organizations that wrote and maintain the code. The single most important manual to have on hand is the GNU C Library Reference Manual available at *https://www.gnu.org/software/libc/manual/*.

## Using the Manual Pages

To make the most of the man pages, you need to learn how to use the pager that displays the pages and to read the man page for the `man` command itself, so that you can understand man page structure and what options the `man` command has.

### The Pager

A *pager* is a program that displays its input one screen at a time. The man pages are stored in a compressed format in the directory hierarchy. The `man`

command decompresses and formats them and then displays them with its pager. The default pager is actually named `pager`, but it's usually a symbolic link to the `less` command. Therefore, when you view a page, you'll most likely be using `less`. The : at the bottom of the screen is followed by your cursor because the : is the `less` command's prompt for you to type something on the keyboard. You can change the viewer that `man` uses by changing the value of the `PAGER` environment variable. The following list describes some of the basic navigation controls when you use the default pager. To see the list of all possible navigation operators, read the man page for the pager.

- To see the next screen press SPACE or enter **`f`** (for *forward*).
- To go back one screen, enter **`b`** (for *backward*)
- To stop reading, enter **`q`** for *quit*.
- To go to line *N*, enter ***`NG`***. If you just enter **`G`**, you'll go to the bottom of the page.
- To search forward for *keyword* enter ***`/keyword`***. Enter **`n`** to find the next occurrence downward, **`N`** to search upward.
- To search backward for *keyword* enter **`?<keyword>`**. Enter **`n`** to find the next occurrence upward, **`N`** to search downward.

Both of the search operators accept patterns with wildcards, which you can read about in the man page for the `pager` command.

### The Structure of Man Pages

Entering **`man`** followed by the name of any command or topic that has a man page, displays that man page. We saw earlier that the `man` command has a page for itself as well. We're about to study that page, but before we do, let's take a look at a couple of other, simpler pages.

Since we've already seen the `echo` command in the Introduction, let's start with that. If you want to learn more about how to use `echo`, you'd enter **`man echo`** and you'd see several screens of output, beginning with:

```
echo(1)                  User Commands                  echo(1)

NAME
       echo - display a line of text

SYNOPSIS
       echo [SHORT-OPTION]... [STRING]...
       echo LONG-OPTION

DESCRIPTION
       Echo the STRING(s) to standard output.

       -n     do not output the trailing newline
       -e     enable interpretation of backslash escapes
```

```
       -E     disable interpretation of backslash escapes (default)
--snip--
```

The top of the page often has everything you need to know, such as what options are available, and whether there are multiple forms of the command.

Sometimes the name of the man page `man` displays is different from the name of the command that you entered as an argument. For example, entering **`man view`** produces the output:

```
VIM(1)                      General Commands Manual                     VIM(1)

NAME
       vim - Vi IMproved, a programmer's text editor
       SYNOPSIS
       vim [options] [file ..]
       vim [options] -
--snip--
       ex
       view
       gvim gview evim eview
       rvim rview rgvim rgview

--snip--
```

which is the page for `vim`, but the `view` command is listed on that page. Sometimes a single man page provides information about related commands. Notice too that instead of the title `User Commands`, this page's title is `General Commands Manual`. People who write man pages follow a standard, but that standard allows some variation, such as in the title of the page.

The sections of a man page are somewhat standardized. A few sections are required, but most sections are optional. The following list shows some common section names and describes their contents.

**`NAME`** The name of this manual page

**`SYNOPSIS`** A brief summary of the command or function's interface

**`DESCRIPTION`** An explanation of what the program, function, or format does

**`OPTIONS`** For commands only, a description of the command line options accepted by a program and how they change its behavior

**`USAGE`** For commands, a more thorough description of the use of the command

**`ENVIRONMENT VARIABLES`** A list of all environment variables that affect the command or function and how they affect it

**`EXIT STATUS`** For commands, a list of exit values returned by the command

**`RETURN VALUE`** For functions, a list of the values the function will return to the caller and the conditions that cause these values to be returned

**ERRORS** For functions, a list of the values that may be placed in the static variable `errno` in the event of an error, along with information about the cause of the errors

**FILES** A list of the files used by the command or function, and files that might be modified

**ATTRIBUTES** Architectures on which it runs, availability, code independence, and so on

**VERSIONS** A brief summary of the kernel or library versions where a function appeared, or changed significantly in its operation

**CONFORMING TO** The standards to which the implementation conforms

**BUGS** A list of limitations, known defects or inconveniences, and other questionable activities

**EXAMPLES** If present, example of how to use the command or function

**AUTHORS** A list of authors of the documentation or program

**SEE ALSO** A list of commands related to this command

**NOTES** General comments that do not fit elsewhere

**NOTE** *It's unfortunate nomenclature that the word* section *is used in two different ways. Do not confuse the sections of a man page with the sections of the manual.*

The most important sections to study when reading a man page for the first time are `NAME`, `SYNOPSIS`, `DESCRIPTION`, and `SEE ALSO`, and if you're reading about a command, then check the `OPTIONS` section also. The `SYNOPSIS` section contains a brief summary of the command or function's interface. If there's an `EXAMPLES` section, I often look at it at right after reading the `SYNOPSIS`, which is usually my first stop on the page. The examples typically include programs you can copy and run or commands that you can try out.

The `SYNOPSIS` section for commands shows the command's syntax, including all arguments and options. Square brackets `[ ]` surround optional elements, a vertical bar `|` (sometimes called an *alternation* operator) separates choices among elements, angle brackets `< >` surround placeholders, and an ellipsis `...` represents elements that can be repeated. When multiple option letters are enclosed in square brackets, such as in `[-aHvW]`, all of them can be given together. If it were written as `[ -a | -H | -v | -W]`, only one of the choices would be allowed. To illustrate, the `git` command, which is a version control program, has the following complex synopsis:

```
git [--version] [--help] [-C <path>] [❶-c <name>=<value>]
    [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
 ❷ [-p|--paginate|-P|--no-pager] [--no-replace-objects] [--bare]
    [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
 ❸ [--super-prefix=<path>] [--config-env=<name>=<envvar>]
 ❹ <command> [<args>]
```

From this synopsis we can conclude several rules:

- The placeholder `<command>` ❹ is the only required element after the command name.
- The element `[-c <name>=<value>]` ❶ is an option to `git`, but if the `-c` is present, it must be followed by the name-value assignment.
- The vertical bar `|` ❷ is used to indicate that at most one of `-p`, `--paginate`, `-P` or `--no-pager` can be used.
- `--super-prefix` ❸ is a *long option* that has a required argument.

For functions, the `SYNOPSIS` shows any required data declarations or `#include` directives, followed by the function declaration. If there are *feature test macro* requirements, which we cover in "Feature Test Macros" on page 103, these are described as well. When you read about a function, you must read the `ERRORS` and `RETURN VALUE` sections; they tell you what possible errors the function reports, what values it can return, and how you need to handle them.

For learning how to use commands and functions, the man page by itself is usually sufficient. To understand how a command interacts with the operating system or how it might be implemented, we'll need to do more research. In "Learning System Programming," we'll go through an exercise that shows how to use the man pages in more detail.

### Searching Through the Man Pages

The `man` command has a number of options for performing searches. Let's look at the beginning of the man page for `man`:

```
MAN(1)                          Manual pager utils                          MAN(1)

NAME
       man - an interface to the system reference manuals

SYNOPSIS
       man [man options] [[section] page ...] ...
       man -k [apropos options] regexp ...
       man -K [man options] [section] term ...
       man -f [whatis options] page ...
       man -l [man options] file ...
       man -w|-W [man options] page ...

DESCRIPTION
       man  is  the system's manual pager. Each page argument given to man is
       normally the name of a program, utility or function. The  manual  page
       associated with each of these arguments is then found and displayed. A
       section, if provided, will direct man to look only in that  section  of
       the  manual. The  default action is to search in all of the available
       sections following a pre-defined order (see DEFAULTS), and to show only
       the first page found, even if page exists in several sections.
--snip--
```

You may not see all of the options that appear here. The POSIX.1-2017 standard (*https://pubs.opengroup.org/onlinepubs/9699919799/utilities/man.html*) requires only the `-k` option, but most implementations provide more. The output shown in this example is from the man page on Ubuntu Linux 22.04. The Linux Man-Pages Project (*https://www.kernel.org/doc/man-pages/*) provides and standardizes man pages, separately from the POSIX.1-2017 standard. A number of Linux distributions, including Debian, Fedora, Gentoo, openSUSE, and Ubuntu, as well as macOS and a few proprietary Unix systems, conform to this latter standard. (See *https://man-db.gitlab.io/man-db/* for an alternative set of man pages that can be installed on other systems.)

The most important options for us are `-k` and `-K`, which allow us to search through the man pages for keywords. If you read further in the man page, you'll see the following example:

```
man -k printf
     Search the short descriptions and manual page names for the
     keyword printf as regular expression. Print out any matches.
     Equivalent to apropos printf.
```

Further down the page, you'll see a description of what this and the `-K` do:

```
    -k, --apropos
           Equivalent to apropos. Search the short manual page descrip
           tions for keywords and display any matches. See apropos(1) for
           details.
    -K, --global-apropos
           Search for text in all manual pages. This is a brute-force
           search, and is likely to take some time ❶; if you can, you should
           specify a section to reduce the number of pages that need to be
           searched. Search terms may be simple strings (the default), or
           regular expressions if the --regex option is used.
```

The `-k` option allows us to search through all man pages to find those short descriptions that match the word we give it. The *short description* is the `NAME` section and its one-line description. The `-K` option searches the entire page, not just the short description, for a match. We are warned that this is slow ❶, but we may occasionally find use for it.

The page also suggests that we should read about the `apropos` command. If we look at its man page, we find exactly what we need:

```
$ man apropos
APROPOS(1)                  Manual pager utils                  APROPOS(1)

NAME
       apropos - search the manual page names and descriptions

SYNOPSIS
       apropos  [-dalv?V] [-e|-w|-r] [-s list] [-m system[,...]] [-M path] [-L
       locale] [-C file] keyword ...
```

```
DESCRIPTION
       Each manual page has a short description available within it. apropos
       searches the descriptions for instances of keyword.

       keyword is usually a regular expression, as if (-r) was used, or may
       contain wildcards (-w), or match the exact keyword (-e). Using these
       options, it may be  necessary to quote the keyword or escape (\) the
       special characters to stop the shell from interpreting them.
--snip--
```

We can use `apropos` for searching. If we give it the `-r` option, we can supply a regular expression, which is a particular type of pattern, or we can give it `-w` and use a different kind of pattern called *wildcards*, which are patterns used for matching filenames. If we give it the `-e` option, it will match the keyword exactly. If we read more in this page, we'll see that by default matching is case-insensitive. Equally important, the `-s` option lets us limit searches to specific sections (volumes), and the `-a` option forces the match to return only those pages that match all of the search terms rather than any of the search terms. A few examples will demonstrate:

```
$ apropos  case
$ apropos  Case
```

Both of these match any line containing the word *case*, case insensitively. Matches can include lines that contain words that have *case* as a substring, such as *lowercase*, *case-insensitive* and so on, and the search will check all sections. Here are two examples that clarify this:

```
$ apropos -s2,3 file
$ apropos  -e file
```

The first command limits the search to sections 2 and 3 and matches descriptions with any words containing *file*, such as *filename*, *FileProducer*, and so on. The second matches only lines that have the exact word *file*, so it excludes *filename*, and *FileProducer* and so on. Consider this one:

```
$ apropos -a convert case
```

This command matches all pages whose short descriptions contain the two words *convert* and *case*, not necessarily next to each other, such as *convert lowercase*. The following command matches just those lines containing the exact word *case* or a word in which *case* is part of a hyphenated word such as *case-sensitive*:

```
$ apropos   ' case' 'case-'
alsaucm (1)          - ALSA Use Case Manager
strcasecmp (3)       - compare two strings ignoring case
strcasecmp (3posix)  - case-insensitive string comparisons
strncasecmp (3)      - compare two strings ignoring case
```

```
strncasecmp (3posix) - case-insensitive string comparisons
wcscasecmp (3)       - compare two wide-character strings, ignoring case
wcscasecmp (3posix)  - case-insensitive wide-character string comparison
wcsncasecmp (3)      - compare two fixed-size wide-character strings, ignorin...
wcsncasecmp (3posix) - case-insensitive wide-character string comparison
```

In summary, `apropos` is a valuable tool for searching for help. We'll be using it extensively in the rest of the book when we need to do background research to implement various system programs.

## Unix History and Standards

> "Finally, the number of UNIX installations has grown to 10, with more expected." – Ken Thompson and Dennis Ritchie, in the UNIX Programmer's Manual, 2nd Edition, June 1972.

Why should you learn anything about the history of Unix if all that you care about is how to write system programs? The most compelling answer is that Unix's complex, haphazard history is the cause of its lack of a single standard and the consequent need to read documentation very carefully to decide whether your code will be portable or even be able to run on your own system. By knowing something about its history you'll see that certain features originated in different Unix distributions and are sometimes incompatible, and that some are fusions of ideas from different branches of the Unix family tree.

Unix has a colorful history filled with many stories [26]. Many articles, websites, and books describe that history in great detail, and at the end of this section I include references to several of them. Here, I describe the major milestones on the path from its birth as an experimental platform for Ken Thompson's "Space Travel" game through the present.

### *The Birth of UNIX*

Ken Thompson wrote the first version of UNIX in assembly language in 1969 while he was working for AT&T Bell Labs. He also revised an old programming language named B so that the system would have a compiler to build programs to run on this new system. By 1970, Dennis Ritchie began working with Thompson on his system. From 1971 through 1973, he worked on a new language, C (based on B) to facilitate the development of Unix, and in 1973, almost all of the UNIX kernel was rewritten in C. This made it possible to port UNIX to any machine with a C compiler. This was the first time an operating system was made portable, and it's also why so much of modern Unix is based on C. In 1974, they presented their ideas in a seminal paper at the ACM Symposium on Operating Systems at IBM Yorktown Heights, the result of which was that awareness of this new operating system grew rapidly.

Meanwhile, work on UNIX continued in Bell Labs, and its popularity within the labs spread as well. In those early years, the UNIX systems were

called *Research systems* by Bell Labs, and each new release was called an *edition*, with their numbers corresponding to the numbers of the *Unix Programmer's Manual* released at the same time. These editions were given names V1, V2, and so on. In 1974, AT&T began licensing UNIX to universities. Because of government restrictions, it wasn't allowed to sell it.

### Early Branches

The University of California at Berkeley (UCB) was one of the universities that obtained a copy of V4 from AT&T, and it embarked on a mission to add more features to the operating system, thereby starting a new fork in its development. When Ken Thompson spent 1975/76 visiting UCB, he and the students there added even more features to their copy of Unix. These features weren't present in the AT&T system from which it derived.

From 1974 to 1979, UCB and AT&T worked on independent copies of UNIX. By 1978, the various versions of Unix had most of the features found in it today, but not all in one system. In the late 1970s, legal actions began under U.S. anti-trust legislation to break up AT&T, the result of which was that by 1982, when the break-up was complete, it was allowed to sell its own brand of UNIX. AT&T then staked proprietary rights to this UNIX, and sold it commercially. AT&T's first major commercial Unix was called System V, released in 1983.

The versions of UNIX developed at UCB were named *Berkeley Software Distributions* (*BSD*) and had names such as *1BSD*, *2BSD*, and so on. BSD systems were released under a much more generous license than AT&T's and didn't require a license fee or a requirement to be distributed with source code. The result was that much BSD source code was incorporated into various commercial Unix variants. By the time that 4.3BSD was written, almost none of the original AT&T source code was left in it. FreeBSD, NetBSD, and OpenBSD were all forks of 4.3BSD having none of the original AT&T source code, and no right to the UNIX trademark, but much of their code found its way into commercial Unix operating systems as well. In short, two major versions of UNIX had emerged—those based on the BSD family and those based on the AT&T version.

### The Free Software Foundation and GNU

In 1983, another event changed the face of computing. Richard Stallman, who had worked in the Artificial Intelligence Lab at the Massachusetts Institute of Technology (MIT), published a manifesto entitled *The GNU Manifesto*. His idea, radical at the time, was that software should be free, not free of cost, but free as in *freedom*. He founded the Free Software Foundation (FSF) in order to campaign and advocate for software whose source code would always be open and free, and for other freedoms associated with its use. He also started the *GNU Project* under the auspices of the FSF. *GNU* was a recursive acronym, for *GNU's Not Unix*.

The objective of the GNU Project was to build a free alternative Unix system, starting from scratch. The project also developed a vast collection

of free software tools and libraries, including compilers, text editors, debuggers, and so on. Although the kernel of the operating system, known as *Hurd* did not receive widespread use, the collection of tools and libraries that GNU created has been adopted in Unix systems worldwide.

### The Rise of Linux

In 1991, the picture was further complicated by the creation of a new kernel named Linux. The Linux kernel was developed from scratch, unlike the BSD systems, which made Linux a lot less like AT&T UNIX than BSD was. Because Linux was just a kernel, without any tools or libraries, it was bundled together with the GNU Project software to turn it into a full-fledged operating system.

Linux was started by Linus Torvalds, who at the time was a student at the University of Helsinki. Many of his ideas were based on the Minix operating system written by Andrew Tanenbaum, who was a professor in Vrije Universiteit in Amsterdam. Tanenbaum made the sources for Minix available with copies of his book on operating systems [30]. Minix ran on Intel 386 processors but wasn't efficient. Torvalds wanted to build a Unix kernel to run more efficiently on the Intel 386.

### Many Unixes

In 1993, AT&T divested itself of UNIX, selling it to Novell, which one year later sold the trademark to an industry consortium known as X/Open. There are now dozens of different Unix distributions, each with its own behavior. There are systems such as Solaris and UnixWare that are based on SVR4, the AT&T version released in 1989, and FreeBSD and OpenBSD based on the UC Berkeley distributions. Systems such as Linux are hybrids, as are AIX, IRIX, and HP-UX.

It is natural to ask what makes a system Unix. The answer is that over the course of the past 30 years or so, standards have been developed in order to define Unix. Operating systems can be branded as conforming to one standard or another. In the next section, we'll explore the various Unix standards.

You can read more about the history of various aspects of Unix in resources such as Dennis Ritchie's telling of its history [23], Salus and Reed's *The Daemon, the Gnu, and the Penguin* [26], Salus's comprehensive telling in *A Quarter Century of UNIX* [25], Brian Kernighan's memoir, *Unix: A History and a Memoir* [12], *UNIX Internals* [18], and *The Design and Implementation of the 4.4BSD Operating System* [15]. You can read transcripts of interviews with many UNIX developers in the *Oral History of UNIX* [13] and read the history of the GNU project at *https://www.gnu.org/gnu/gnu.html*. Torvalds and Diamond published an account of Linux development [33], and the appendix of [5] has an interesting exchange of ideas between Torvalds and Tanenbaum germane to the design of the Linux kernel. The bibliography also has a more extensive list of references.

### Unix and Related Standards

One widely accepted Unix standard is the *POSIX* standard, an acronym for *Portable Operating System Interface*. Technically, POSIX doesn't define Unix in particular; it's more general than that. POSIX is a family of standards known formally as *IEEE 1003*. It was also published by the *International Standards Organization* (*ISO*) with the name *ISO/IEC 9945:2003*; these are one and the same document.

**NOTE** *The most recent version of POSIX is IEEE Std 1003.1-2017, also known as POSIX.1-2017. The POSIX.1-2017 standard consolidates the major standards preceding it, including POSIX.1, and the Single UNIX Specification, Version 4 (SuSV4).*

The spirit of POSIX is to define a Unix system, as is stated in the Introduction to the specification (*http://pubs.opengroup.org/onlinepubs/9699919799/*):

> POSIX.1-2017 defines a standard operating system interface and environment, including a command interpreter (or "shell"), and common utility programs to support applications portability at the source code level. It is intended to be used by both application developers and system implementors [sic].

The Single UNIX Specification was derived from an earlier standard written in 1994 known as the *X/Open System Interface* which itself was developed around a Unix portability guide called the *Spec 1170 Initiative*, so called because it contained a description of exactly 1,170 distinct system calls, headers, commands, and utilities covered in the spec. The number of standardized elements has grown to 1,833 in the most recent version.

The Single UNIX Specification was revised many times starting in 1997 by The Open Group, which was formed in 1996 as a merger of X/Open and the Open Software Foundation (OSF), both industry consortia. The Open Group owns the UNIX trademark. It uses the Single UNIX Specification to define the interfaces an implementation must support to call itself a UNIX system. The latest edition was revised in 2018.

The specification standardizes the collection of all system calls, libraries, and those utility programs such as `grep`, `awk`, and `sed`, that make Unix feel like Unix. The collection of system calls is what defines the Unix kernel. The system calls and libraries together constitute the Unix application programming interface, whereas the utility programs constitute the Unix user interface.

There are four major parts to the standard:

**Base definitions** General terms, concepts, and interfaces common to all volumes of the standard, including utility conventions and C-language header definitions

**System interfaces** Definitions for system service functions and subroutines, language-specific system services for the C programming language, function issues, including portability, error handling, and error recovery

**Shell and utilities** Definitions for a standard source code-level interface to command interpreters and common utility programs for application programs

**Rationale** An informative section, which contains historical information concerning the contents of POSIX.1-2008 and why features were included or discarded by the standard developers

POSIX.1-2017 also defines areas as being outside of its scope:

- Graphics interfaces
- Database management system interfaces
- Record I/O considerations
- Object or binary code portability
- System configuration and resource availability

The Single UNIX Specification, Version 4, from 2018 is identical to POSIX.1-2017 with the addition of standards for the *Curses Library*, which is a terminal control library that can be used to create interactive programs that run in terminal windows, such as text editors and games.

The fact that there are standards does not imply that all Unix implementations adhere to them. Although there are systems such as AIX, Solaris, and macOS that are fully POSIX-conformant, most are mostly compliant. Systems such as FreeBSD and various versions of Linux fall into this category.

Any single Unix system may have features and interfaces that do not comply with a standard. The challenge in system programming is being able to write programs that will run on a broad range of systems in spite of this. A Unix man page generally shows to which standards the topic of the man page conforms. The `standards` man page, in Section 7, lists all of the names used for the standards referenced in the man pages. If you enter the command **`man standards`** you will see the full list. In Chapter 3, "Feature Test Macros," we'll go over how *feature test macros* are used to provide a means to compile a single program on a variety of different Unix systems.

## C Standards

The interfaces described in the POSIX standard are written in C mostly because most system programs are written in C, and because much of Unix was originally developed in C. Because of this, POSIX depends upon a standard definition of C, and it uses the ISO standard, the most recent version of which is officially known as ISO/IEC 9899:2018, and informally known as C18. C18 essentially corrected defects in C11 and adds little more. C11 incorporated the earlier ANSI C and augmented it. This version of C is known as ISO C, but people also continue to call it ANSI C, even though they are not the same. You can download the last free draft of the C11 standard as well as the most recent draft of C18 from the Resources page of my website *http://www.compsci.hunter.cuny.edu/~sweiss/resources/*.

In short, POSIX specifies not just what Unix must do, but what the various parts of the C Standard Library must do as well. It specifies, in effect, a superset of the C language, including additional functions to those introduced in standard C. Therefore, a Unix system that is POSIX conformant contains all of the library functions of the ISO C language. For example, every Unix distribution includes libraries such as the C Standard I/O Library, the C math library, and the C string library.

The C Standard Library provided for Linux as well as several other Unix distributions is the GNU C library, called GNU `libc`, or `glibc`. GNU often extends the C library, and not everything in it conforms to the ISO standard, nor to POSIX. What all of this amounts to is that the version of the C library on one system is not necessarily the same as that found on another system.

This is one reason why it's important to know the standard and know what it defines and what it doesn't define. In general, the C standard describes what's required, what's prohibited, and what's allowed within certain limits. Specifically, it describes the following:

- The representation of C programs
- The syntax and constraints of the C language
- The semantic rules for interpreting C programs
- The representation of input data to be processed by C programs
- The representation of output data produced by C programs
- The restrictions and limits imposed by a conforming implementation of C

Not all compilers and C runtime libraries comply with the standard, and this complicates programming in C. The GNU compiler has command line options that let you compile according to various standards. For example, if we wanted our *hello.c* program to be compiled against the ANSI standard, we would enter:

```
$ gcc -ansi hello.c -o hello
```

and since C90 is the same standard as ANSI, we could also enter

```
$ gcc -std=c90 hello.c -o hello
```

As another example, to compile *hello.c* against the ISO C11 standard, we could enter:

```
$ gcc -std=c11
```

Even though there are later ISO C standards, if we use the previous command, it will apply the most recent C standard anyway.

Understanding how to write programs for Unix requires knowing which features are part of C and which are there because they are part of Unix. In other words, you'll need to understand what the C libraries do and what the underlying Unix system defines. Having a good grasp of the C standard will make this easier.

## Summary

System programs are fundamentally different from the kinds of programs that most beginning students learn how to write, because they access protected resources inside the computer system. What actually happens when a program makes a relatively simple call to print onto the terminal window involves much more than what meets the eye. The sequence of steps includes the use of system calls, which are function calls into the kernel code. The kernel is the core of the operating system, the part that is memory-resident as long as the computer is powered-on, and is responsible for protecting, managing, and making available the wide range of resources in the computer system.

Unix introduced many novel ideas in the design of operating systems. Some of the most innovative ideas that made it so successful are

- A programmable, interchangeable command line interpreter, called a shell, that runs in userspace rather than as a part of the kernel
- The concept of processes and the method of process creation
- The use of two levels of privilege to provide protection of the kernel and its resources
- Device-independent I/O operations
- The representation of files as sequences of bytes without structure
- I/O redirection and pipes in particular,
- the concepts of users and groups, and file permissions,
- the single directory hierarchy, and
- the environment concept.

The growth and spread of Unix led to many different Unix varieties and distributions and a need for standardization. This in turn led to the creation of a consortium that created the POSIX standards for its interfaces and behavior.

## Exercises

1. Who are the authors of the `bash` shell? (Hint: use the man pages to find out.)
2. What is the return type of the `read()` system call?
3. Using the man pages, find the names of all of the header files that you would need to include to use the following functions in a program. There might be more than one needed for some of these.
   (a) `_exit()`
   (b) `setuid()`
   (c) `fstat()`
4. If your current working directory is */usr/share/gcc/python*, what is the shortest length relative pathname of the file */usr/lib32/libc.so.6*.

5. What command can be used to print the creation date of a file? (Hint: this information is part of a file's status.)

# 2

# WORKING IN THE COMMAND INTERFACE

This chapter is a practical introduction to working in a Unix environment with an emphasis on using bash. It begins with a brief history of Unix shells and then describes the essential features of bash and how to take advantage of many of them, including I/O redirection, background processing, shell parameters and variables, environment variables, command types, file globs, control flow, command substitution, and scripting.

The chapter is not a comprehensive tutorial on bash nor on all of its features. For example, it doesn't contain information about the *history mechanism*, which is a means to recall commands previously executed, the various modes in which bash can run, nor details about all possible compound statements and expressions. I'll focus on the most essential and useful features of the shell, including how to view and change file permissions, and how to use basic commands for file and directory manipulation. After reading the basics here, I hope you'll be interested in learning more about bash and investigating other commands on your own.

## Historical Remarks About Shells

You'll have a better understanding of `bash` and other shells if you understand their origins and how features were incorporated into them over the years.

Ken Thompson wrote the shell for the first version of Unix in 1969. It borrowed ideas from the shell used in the MULTICS project, on which he had worked. The term *shell* had been used in that project as another name for a command line interpreter [1]. The original Thompson shell lacked many features of modern day shells.

The C shell was written by Bill Joy at UC Berkeley and made its appearance in Sixth Edition Unix (1975), the first widely distributed release from UC Berkeley. It was an enhancement of the original Thompson shell with C-like syntax. Among its notable additions are the history mechanism and command line editing, and it's part of all BSD distributions.

The Bourne shell, written by Stephen Bourne, was introduced into Unix in System 7 (1979), which was the last release of Unix by AT&T Bell Labs prior to the commercialization of UNIX by AT&T. Its syntax derives in part from the Algol 68 programming language and is a part of all Unix releases.

David Korn developed the Korn shell at Bell Labs and introduced it into the SVR4 commercial release of Unix by AT&T. Its syntax is based on the Bourne shell, but it had many more features.

The TENEX C shell, or TC shell, extended the C shell with command line editing and completion, as well as other features which were found in the TENEX operating system. It was written by Ken Greer and others in 1983.

The Bourne-Again SHell (`bash`) is an extension of the Bourne shell with many of the sophisticated features of the Korn shell and the TC shell. It's the default user shell in Linux and has become popular because of this.

Many Linux distributions have incorporated a new shell known as `dash`. The `dash` shell is a POSIX-conformant implementation of the Bourne shell that is designed to be fast and efficient. It's a direct descendant of the Almquist SHell (`ash`), ported to Linux in 1997 and was renamed `dash` in 2002.

## Working with the Shell

Your view and appreciation of Unix is pretty much determined by the interface that the shell creates for you. The shell hides the inner workings of the kernel, presenting a set of high-level functions that can make the system easy to use. It's like the walnut pictured in Figure 2-1; the shell hides the kernel in the same way.

*Figure 2-1: The shell hides the kernel of Unix.*

If you've used a Unix system with a graphical user interface (GUI), be aware that this GUI is a separate and distinct application with respect to the Unix operating system. The GUI provides an alternative to a shell for interacting with Unix, but experienced users usually rely on using a shell for many tasks because it is much faster to type than it is to point and click with a mouse.

Unlike most operating systems, Unix allows you to replace the original login shell with one of your own choosing. The particular shell that will be invoked when you log in is specified in an entry for your user account in a file called the *password file*. The system administrator can change this entry, but sometimes you can use the `chsh` command to change shells, if it's available on your system. On some versions of Unix, you can enter **`passwd -s`** to change your login shell. Depending on the system, you might need someone with superuser privileges to change your login shell.

## Shell Features

In all shells, simple commands have the following form:

```
commandname command-options arg1 arg2 ... argn
```

The shell waits for you to enter a newline character (by pressing ENTER) before it attempts to interpret (or *parse*) a command. A newline signals the end of the command.

Once the shell receives the entered line, it checks to see whether *`commandname`* is a *built-in shell command*. Built-in commands are hardcoded into the shell itself. If it is, the shell executes the command. If not, it searches for a file whose pathname, either relative or absolute, is *`commandname`*. We explain how that search takes place in "Parameters and Variables" on page 62.

If the shell finds the file, it's loaded into memory, and the shell creates a new process to execute the command, passing it the arguments from the command line. When the command terminates, the created process terminates also, and the shell waits for you to enter another command. If it doesn't find the file, it displays a message, such as `Command` *`commandname`* `not found`.

In addition to command interpretation, all shells provide the following:

- Redirection of the input and output of commands: changing the source of input or the target of output, respectively

- Pipes: a method of channeling the output of one command to the input of another
- Scripting: storing a sequence of shell instructions and commands in a file that can be executed
- Filename substitution using metacharacters: a method of naming one or more files using a pattern-matching language that has *wildcards* and other operators
- Control flow constructs: loops and conditional execution

Shells written after the Bourne shell also provide the following:

- History mechanism: a method of saving and reissuing commands in whole or in part using a succinct notation
- Backgrounding and job control: a method of controlling the sequencing and timing of commands
- User-defined aliases: typically for frequently used commands

The `bash` shell has a number of features in addition to these, such as

- Interactive file and command name completion
- General interactive command line editing
- Co-processes: named processes that run concurrently with the shell
- More general redirection of input and output of commands
- Array variables within the shell
- Autoloading of function definitions from files
- Restricted shells: shells that limit what a user can do, for security reasons

In the following sections, we'll explore some of these features in more depth.

### Standard I/O and Redirection

Unix uses a clever method of handling I/O. Every program is automatically given three open *input/output streams* when it begins execution: *standard input*, *standard output*, and *standard error*. They're called *streams* because they're linear sequences of characters. By default, standard input is the keyboard, and standard output and error are the terminal window. The shell can reference these streams using the numbers 0, 1, and 2 for standard input, standard output, and standard error, respectively, as illustrated in Figure 2-2.

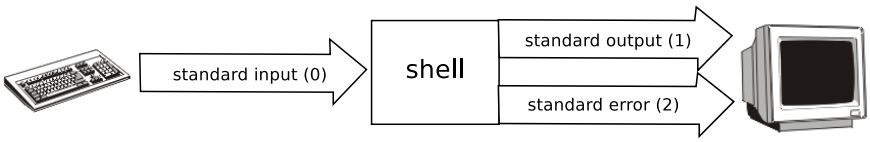



*Figure 2-2: The three streams assigned to every process*

Commands usually read from standard input, write to standard output, and send their error messages to the standard error stream, which appears in the terminal window. The shell, however, can trick a command into reading from a different source or writing to a different source. This is called *I/O redirection*. For example, the command

```
$ ls mydir
```

lists the files in the directory named *mydir* on the terminal. In contrast, the command

```
$ ls mydir > mylisting
```

creates a file called *mylisting* in the current working directory and redirects the output of the `ls` command to *mylisting*, provided that *mylisting* doesn't already exist. If it does exist, it usually displays a message such as the following:

```
bash: mylisting: cannot overwrite existing file
```

The greater-than character (>) in the preceding example is the shell *output redirection operator*. It replaces the standard output of the program to which it is applied by the file whose name immediately follows the operator, which means `> outfile` writes the standard output of whatever command to which you apply it to a file named *outfile* instead of to the terminal, provided that it's allowed to overwrite the file.

### Controlling Overwriting of Files

System administrators usually configure users' shells so that their default behavior is to prevent files from being overwritten. They do this by including either of the following two lines in the shell's startup file to enable a shell option named `noclobber`:

```
set -o noclobber
set -C          # Short form of set -o noclobber
```

In `bash`, the startup file is *.bashrc*, located in the home directory. Other shells have startup files with different names. If `noclobber` is set, overwriting the file is prevented; otherwise, the redirected output replaces the file's contents.

Enter **`echo $SHELLOPTS`** in your shell to check whether `noclobber` is set. If `noclobber` appears in the output, it's set, otherwise it isn't. The following output shows that `noclobber` is set:

```
$ echo $SHELLOPTS
braceexpand:hashall:history:ignoreeof:interactive-comments:monitor:noclobber
```

If it isn't set in your shell, in `bash`, you can set it yourself by editing your *.bashrc* to include either of the two lines shown in the previous listing.

If you're willing to overwrite a file even though `noclobber` is set, use the `>|` operator, as in the following:

```
$ ls mydir >| mylisting
```

This command redirects standard output to a file, overwriting it if it already exists, ignoring `noclobber`.

#### Redirecting Standard Input

The notation `< infile` means "read the input from the file *infile* instead of from the standard input stream, which is by default the keyboard." The `<` operator is called the *input redirection operator*. If a program named *cmd* expects input from the keyboard, the command *cmd* `<` *infile* replaces the keyboard with the contents of *infile*.

To demonstrate, suppose we've written a program named `sum` that expects the user to enter numbers, one per line, on the keyboard, ending by pressing CTRL-D, and prints their sum:

```
$ sum
1
2
3
4
CTRL-D entered here
10
```

If *numbers* is a file containing these same four numbers, one per line, we can enter the following:

```
$ sum < numbers
10
```

The input to `sum` came from *numbers* instead of the keyboard.

#### Redirecting Both Standard Input and Output

We can redirect both the input and output of a command:

```
$ command < infile > outfile
```

This causes `command` to read its input from *infile* and send its output to *outfile*.

The order of the the input and output redirection operators doesn't matter. For example, we can enter

```
$ sum < numbers > total
$
```

or

```
$ sum > total < numbers
$
```

and the sum of the numbers will be written to the file named *total* instead of to the screen.

In bash, we can write any of the following semantically equivalent lines:

```
$ command > outfile < infile
$ > outfile command < infile
$ < infile > outfile command
```

Only the first of these is POSIX-compliant syntax, and it's best to avoid putting any redirection before the command name.

### Appending Redirected Output

The *append operator* (>>) is like the output redirection operator, but instead of *overwriting* the target file, it appends the output to it. One example of its use is in maintaining logfiles. If you have a file named *mylog* that already has log data in it and you run the command

```
$ echo 'Today I learned about redirection' >> mylog
```

that line will be appended to *mylog*, and if *mylog* didn't exist, it will be created with that text as its first line.

### Redirecting with Pipes

We can carry the concept of redirection one step further to allow the output of one command to be the input of another. This is known as a *pipe* or *pipeline*, and the operator that does it is a vertical bar (|), for example:

```
$ ls mydir | sort | lpr
```

This pipeline makes the output of `ls mydir` become the input to the `sort` command, which then sorts it, and sends the sorted list to the `lpr` command, which is a command to send files to the default printer attached to the system.

We could do this using temporary files rather than pipes as follows:

```
$ ls mydir > temp1
$ sort < temp1 > temp2
$ lpr < temp2
$ rm temp1 temp2
```

It's not exactly the same, because when a pipeline is established, the commands run simultaneously. In the example using the pipe, the `ls`, `sort`, and `lpr` commands start up together. As `ls` is busy working and producing some output, `sort` is receiving that output. The shell uses a kernel mechanism called *pipes* to implement this communication. (We'll cover kernel pipes in Chapter 18.) Several other redirection operators exist, but for now we describe only two more.

### Using Redirection for Error Handling

Neither the output redirection operator nor the append operator send the standard error stream to the file. This means you'll see any error messages sent to the standard error stream on your terminal.

Some commands can produce many error messages, and sometimes you don't care about the errors. For example, if you try to list every file in the entire filesystem using `ls -R /`, you'll see many `Permission denied` errors.

Using the `find` command can also generate many error messages. You can use `find` to search through a part of the directory hierarchy for files that satisfy various criteria, and you can run commands on those files when it finds them. Sometimes you'll get lots of errors because you don't have permission to see the content of certain directories, for example:

```
$ find /var  -name "*.log"
find: '/var/spool/rsyslog': Permission denied
find: '/var/spool/cups': Permission denied
find: '/var/spool/cron/atjobs': Permission denied
find: '/var/spool/cron/crontabs': Permission denied
find: '/var/spool/cron/atspool': Permission denied
--snip--
```

Here, `find` begins a search of the directory hierarchy rooted in the top-level directory */var* for files whose names end in *.log*, and it prints their pathnames if it finds any. To specify this set of files, we used the `bash` *wildcard* operator (`*`), which matches any sequence of zero or more characters that does not start with a period. Because `*` matches any string of characters, the expression `*.log` matches filenames that begin with any non-period character and end in `.log`. (See "File Globs" on page 63 for a description of this type of pattern matching.)

We can prevent all error messages from appearing on the screen by redirecting them with the `bash` construct `2>/dev/null`. The `2>` part of this construct tells `bash` to send the standard error stream, whose number is 2, into the file */dev/null*, but */dev/null* is not a real file; it's like a black hole in that anything you write to it disappears. (The kernel discards it.) The following is part of what we see when we redirect the error stream to */dev/null* with the previous command:

```
$ find /var  -name "*.log" 2>/dev/null
/var/lib/texmf/web2c/metafont/mf.log
/var/lib/texmf/web2c/tex/tex.log
```

```
/var/lib/texmf/web2c/updmap.log
/var/lib/texmf/web2c/luatex/lualatex.log
/var/lib/texmf/web2c/luatex/lualatex-dev.log
--snip--
```

If we want to redirect both the standard output and the standard error streams to the same file, we'd use the `&>` operator, as in

```
$ find /var -name "*.log" &> myfile
```

which sends both streams to the file *myfile*. To learn more about I/O redirection, including several operators not described here, read the `bash` man page.

## Control Operators and Multitasking

You can enter multiple commands on a single line by using certain control operators that are usually called *command separators*. Command separators are characters used to terminate commands that appear on the same line.

### Sequencing

The semicolon (`;`) acts like a newline character to the shell—it terminates the preceding command:

```
$ echo 'hello world' ; hostname ; whoami
hello world
harpo
stewart
```

This example runs each of the three commands, one after the other. It also introduces the `whoami` command, which displays the username of the user running it. The semicolon `;` is used to sequentially execute the commands.

### Grouping Commands

You can use parentheses to group a sequence of commands so that they are treated like a single command. We often need to do this to bypass the operator precedence rules of the shell. To demonstrate:

```
$ echo 'hello world'; hostname; whoami > outfile
hello world
harpo
$
```

The output of `echo 'hello world'; hostname` appears on the screen and only the output of `whoami` is written to the file named *outfile*. This is because the semicolon has higher precedence than the redirection operator, which applied only to the `whoami` command. Now try this:

```
$ echo 'hello world'; ( hostname; whoami) > outfile
hello world
```

```
$
```

In this case the output of `hostname; whoami` was written to the file. Grouping parentheses comes in handy in many situations.

### Backgrounding and Job Control

The ampersand (`&`) is another useful control operator. When a command is terminated with `&`, the shell doesn't wait for the command to finish before it re-displays the prompt, which is useful when we run commands that take a long time but we want to continue working. In this case we say that the command, or *job*, is running *in the background*.

We can't see the effect of using the ampersand unless we've got a long-running command, but we can simulate a long-running command with the `sleep` command, which simply puts the calling shell to sleep for the number of seconds that we provide as an argument. In other words, `sleep` doesn't terminate until the amount of time elapses:

```
$ sleep 10
        -- 10 seconds pass here --
$
```

This command makes the shell wait for 10 seconds before the prompt returns.

The following command prints `done sleeping` after 10 seconds:

```
$ sleep 10 ; echo done sleeping
        -- 10 seconds pass here --
done sleeping
$
```

If we terminate this entire command with the ampersand, the prompt returns immediately after printing the job number and PID of the job it started in the background:

```
$ (sleep 10 ; echo done sleeping) &
[1] 15340
$ done sleeping
```

Here it prints `[1] 15340` to indicate that the backgrounded job is job number 1 and has PID 15340.

To use the job number to refer to the job, precede the number with a percent sign. For example, `%1` refers to job number 1.

When a command is backgrounded, by default, its output still goes to the terminal, which is why the `done sleeping` message appeared after the prompt. We can avoid this behavior by redirecting the output into a file:

```
$ echo 'hello world' > outfile &
$
```

This example causes the `echo` command to work in the background, writing `hello world` into the file *outfile*.

Backgrounding is a form of *multitasking*. It allows you to run more than one process under the same shell at the same time. The `&` doesn't need to be at the end of the line; we can also use it to run separate commands simultaneously by putting it between them:

```
$ echo hello world > outfile1 & whoami > outfile2 &
```

This tells the shell to run the `echo` and `whoami` commands *concurrently* and in the background, putting their outputs into the *outfile1* and *outfile2* files, respectively.

The `>` operator has higher precedence than the `&` operator, which means that the previous commands are equivalent to the following:

```
$ (echo hello world > outfile1) & (whoami > outfile2) &
```

Suppose that you put a job in the background and some time later you want to run it in the foreground. The `fg` command (for *foreground*) will bring a job to the foreground. Its general form is `fg` *`job-spec`*, where a *`job-spec`* is a percent sign (`%`) followed by a number, a string, or a question mark (`?`) followed by a string:

**`fg %n`** Foreground job $n$.

**`fg %string`** Foreground the job whose command begins with *`string`*.

**`fg %?string`** Foreground the job whose command contains *`string`*.

If there's only a single job running in the background, you can omit the `job-spec` entirely and just enter `fg` to bring that job to the foreground. In all cases, the command that you entered and put in the background is written to the terminal when it is brought to the foreground.

Let's go over some examples. Assume the backgrounded command is the following:

```
    ( sleep 30 ; echo done sleeping) &
```

If this is the only job running in the background, just enter `fg`:

```
$ ( sleep 30 ; echo done sleeping) &
[1] 15412
$ fg
( sleep 30; echo done sleeping )
    time elapses
$
```

You can also enter the `fg` command explicitly:

```
$ ( sleep 30 ; echo done sleeping) &
[1] 15373
$ fg %1
```

```
( sleep 30; echo done sleeping )
    time elapses
$
```

You can use any prefix of the command name after the percent sign to refer to the job, but if it contains special characters such as (, you need to enclose it in single quotes:

```
$ ( sleep 30 ; echo done sleeping) &
[1] 15430
$ fg %'( sle'
( sleep 30; echo done sleeping )
    time elapses
$
```

If multiple jobs start with that prefix, you'll get an error message.

In the third method, you can choose any substring contained in the command instead of a prefix of it with the `%?` operator:

```
$ ( sleep 30 ; echo done sleeping) &
[1] 15430
$ fg %?echo
( sleep 30; echo done sleeping )
    time elapses
$
```

The same rules apply about enclosing it in single quotes.

Finally, you can omit the `fg` entirely and just enter a job-spec to bring the job to the foreground. In the preceding example, we could have just entered:

```
$ ( sleep 30 ; echo done sleeping) &
[1] 15430
$ %?echo
( sleep 30; echo done sleeping )
    time elapses
$
```

If you ran a command in the foreground and decide you want to put it into the background so you can work on something else while it's running, you can do the two-step process:

1. Enter the *suspend character* by pressing CTRL-Z to stop but not terminate the running command. This is called *suspending* the process.
2. Enter the *background* command, `bg`, to put it into the background.

These operators are usually sufficient for most tasks, but if you're curious, see the `JOB CONTROL` section of the `bash` man page to learn more.

### The Shell as a Command

You can run a shell the same way that you run commands. Enter the name of a shell, for example, `bash`, `sh`, `csh`, and so on, at the command prompt within any shell to start another shell:

```
$ bash
$
```

This example starts the `bash` shell. If you do that, you'll have two instances of the `bash` shell running, but the first will be dormant, waiting for the second to exit. When one shell is created as a result of a command given in another shell, the created shell is called the *subshell* of the original shell, which is called the *parent shell*.

If you forget what shell you're currently running, two simple commands can tell you. The first one is as follows:

```
$ ps -p $$
    PID lstinlineY          TIME CMD
   9855 pts/1    00:00:00 bash
```

The `ps` command prints information about some, but not all, processes that you own. The `-p` option is followed by the process IDs of processes you want to display. The process ID of a shell, at least most shells, is in a variable named `$`. By preceding the PID with a dollar sign, you're telling the shell to evaluate the variable and replace it with that value before passing the argument to the command. In this example, `ps -p` has the argument `$$`, so the shell evaluated it and gave the command the process ID 9855.

The following shows a second method to see what shell you're running:

```
$ echo $0
bash
```

In the Introduction, I introduced the `echo` command to demonstrate some notation. That command may seem rather useless, but it's very convenient. The `echo` command evaluates its arguments and displays their values. If an argument is a variable, that variable is replaced on output by its value. If you give `echo` the name of a variable preceded by the dollar sign `$`, it evaluates the variable and displays its value. The argument `$0` is the name of the command itself, meaning whatever you typed on the command line.

As another example, you can display the value of any environment variable using `echo`; for example, to display the value of the `SHELL` environment variable, you can enter:

```
$ echo $SHELL
/bin/bash
```

This prints the name of your login shell, which may not be the current shell.

### Shell Parameters and Variables

Because bash is a full-fledged programming language, it has many features similar to higher-level compiled languages, including parameters. A *parameter* is an entity that stores a value. It's a generalization of the idea of a variable. Parameters are denoted by one of the following:

**Name** A string beginning with a letter or underscore and containing only letters, digits, and underscores, such as user_name.

**Number** Any sequence of decimal digits, such as 12, excluding 0.

**Special character** One of the following characters: @, *, ?, -, #, $, !, or 0.

When a parameter is identified by a name, it's called a *variable*. Whereas users can assign a value to a variable, only bash can give a value to a parameter that isn't a variable. We'll discuss the use of some bash parameters in "Shell Scripts" on page 70.

Some variables are built into bash; these are called *shell variables*. In general you shouldn't alter the values of most shell variables, since bash uses them in specific ways. For example, PWD is the pathname of the current working directory and you should not change it. Some shell variables, though, are customizable, such as PATH, which is the search path for commands. It's a colon-separated list of directory pathnames in which bash looks for commands, such as

```
/bin:/usr/local/bin:/usr/bin
```

When you enter a command whose name does not contain a slash character, such as echo, bash searches this list of directory paths in left-to-right order to try to find the matching command. Specifically, for each directory pathname in the list, it appends the command name to the end of that pathname, preceded by a slash, and checks whether it's an executable file that you have permission to execute. It does this until it has searched the entire list of pathnames in PATH. For example, with the PATH shown previously, if you enter the echo command, it checks whether */bin/echo* exists and is executable by you, then whether */usr/local/bin/echo* exists and is executable by you, and finally it checks */usr/bin/echo*, which is the echo command that is executed for you. You can modify the PATH variable to customize your search path. You can rearrange the order of its directories and add and remove directories from it.

You can also create your own variables with an assignment operator:

```
$ mygreeting=hello
```

This creates and initializes a variable named mygreeting. Very often, program developers create environment variables that allow the user to control the behavior of the program. In Chapter 4, you'll see examples of how various commands use variables added to the environment for their own use.

When a variable is given a value, we say that it's *set*, even if it's a null string. The only way to remove a variable is to *unset* it with the unset built-in bash command:

```
$ unset mygreeting
```

Parameters and variables are evaluated when their names are preceded by a dollar sign `$`. This is called *parameter expansion*. We can use `echo` to display the value of a variable by giving it any variable name preceded by `$`:

```
$ mygreeting='Hello, what would you like to do now?'
$ echo $mygreeting
Hello, what would you like to do now?
$
```

When the shell parsed the preceding command and found the variable `mygreeting` preceded by `$`, it replaced the variable by its value before running a process to execute the `echo` command.

You can enclose the parameter or variable name in braces:

```
$ mygreeting='Hello, what would you like to do now?'
$ echo ${mygreeting}
Hello, what would you like to do now?
$
```

The preceding command didn't need the braces; they have no effect on its behavior. However, braces are required if the parameter is more than a single digit or if the name is followed by characters that are not supposed to be part of the name. The following examples demonstrate this:

```
$ var=123
$ echo $var
123
$ echo $varabc
$
```

This produced no output because there is no variable named `varabc`. In contrast

```
$ echo ${var}abc
123abc
$ newvar=${var}_new
$ echo ${newvar}
abc_new
```

produced output because the actual variable name was enclosed in braces, so that `bash` could find it. You'll see that being able to append characters to the expansion of a variable or parameter is very useful. (See "Control Flow in the Shell," on page 66 for an example.)

### File Globs

All shells have the ability to parse patterns that represent sets of filenames. These patterns are called *file globs*, *globs*, or *wildcard expressions*. The shell will

expand a file glob into the list of filenames of existing files that match that glob. A string is called a glob if it contains one of the shell special characters: `?`, `*`, or `[`. Each of these has a specific purpose. A question mark (`?`) not enclosed in square brackets (`[ ]`) matches any single character except the forward slash (`/`). The asterisk (`*`) matches any string not containing a `/`, including an empty string. Neither of these two wildcard characters match a leading dot (.). For example, if the current directory contains the files *file_io.c*, *utils.c*, *finding.c*, *.config*, and *users.o*, then

```
$ ls f*.c
```

lists the files *file_io.c* and *finding.c* because those names match that glob.

The command

```
$ rm *.o
```

removes any file ending in *.o*, which is *users.o*, and the command

```
$ gcc -c f*.c
```

runs `gcc -c` on every file in the current working directory whose name starts with *f* and ends in *.c*.

This command

```
$ ls *
```

doesn't display *.config* because it starts with a dot.

Square brackets enclosing a list of characters represent any single character in that list, unless the first character is an exclamation mark (`!`), in which case they match any character *not* in the list:

**`[abc]`** Matches any of `a`, `b`, and `c`

**`[!abc]`** Matches any character other than `a`, `b`, and `c`

**`[[]`** Matches `[`

**`[]]`** Matches `]`

**`[abc!]`** Matches `a`, `b`, `c`, and `!`

You can use a hyphen to represent ranges of characters inside square brackets as well:

**`[a-z]`** Matches any lowercase letter

**`[!a-z]`** Matches any character other than a lowercase letter

**`[A-Za-z0-9]`** Matches all letters and digits

**`[-a]`** Matches `-` and `a`

Be very careful when using file globs, especially with dangerous commands such as `rm` that are not reversible, because they may represent files that you did not think they did. One disastrous example would be

```
$ rm -r .*
```

which a naive user might think removes the "hidden" files in the given directory and their descendants. But the pattern `.*` matches `..` because the second period is matched by `*`, implying that the command will recursively remove everything in `..`, the parent directory. I've only scratched the surface of how to use file globs. To learn more about them, enter

```
$ man 7 glob
```

to display their man page.

### Command Substitution

Most shells have a feature called *command substitution*, which allows the output of a command to replace the actual command. There are two forms. The original syntax is

```
`command`
```

and the newer syntax is

```
$(command)
```

These two methods have some subtle differences, and it's best to use the `$(...)` method. You will probably see the first method used in legacy `bash` scripts. To illustrate how they work, suppose that *filelist* contains the names of three files: *file1*, *file2*, and *file3*, one per line. Consider the command:

```
$ wc  $(cat filelist)
```

To execute this command, `bash` runs `cat filelist`, which outputs the contents of the file named *filelist*, converting the output into a space-separated list of its lines on the command line. Since *filelist* contains the lines

```
file1
file2
file3
```

`bash` replaces the text `$(cat filelist)` with the text `file1 file2 file3` on the command line, so that the actual command that it runs is

```
$ wc  file1 file2 file3
```

The result is that it runs `wc` on each of these files successively.

A second example is:

```
$ head -1 $(find repo -name "*.c" 2>/dev/null)
```

The `find` command is complex, but this use of it is not. It searches the entire directory hierarchy rooted at *repo* for every file whose name matches the file glob `*.c`, which would match all *.c* files.

Because `find` can generate many error messages, the error output is thrown away by redirecting it to */dev/null*. The remaining lines that `find` outputs replace the command itself, separated by spaces instead of newlines. These lines will be the pathnames of all *.c* files that it finds. The `head -1` command then prints the first lines of every such *.c* file.

### Control Flow in the Shell

All shells are designed as programming languages with the usual palette of control flow keywords and operators. The syntax varies from one shell to another. This description is about `bash` specifically.

The most useful and important control flow constructs to learn are simple branching using `if...then...else` instructions, and looping using `for` loops. `bash` also has `while` loops, `until` loops, `case`(`switch`) statements, and `select` statements, but I don't describe them here. You can read about them in the `Compound Commands` section of the `bash` man page.

There are two types of `for` loops, an iterative one like the one commonly found in high-level languages, and a list-based one, present in most modern languages. Their syntax is slightly different and semicolons play a critical role. The following shows the typical form of the list-based `for` loop:

```
for name in list ; do list of commands ; done
```

Those semicolons must be present if the command is on a single line. You can instead replace any of them by a newline, for example:

```
for name in list
    do
    list-of-commands
    done
```

Here are some examples of its use:

```
$ for i in 1 2 3 ; echo $i ; done
1
2
3
```

This just prints the numbers 1, 2, and 3, one per line.

```
$ for i in $(seq 1 200) ; do echo $i ; done > nums1-200
```

This second example creates a file named *nums1-200* that contains the numbers 1 to 200, one per line. It should convince you that this loop, combined with command substitution, can be powerful.

```
$ for file in *.o ; do cp  $file ${file}_bkup ; done
```

This example copies each file in the current directory whose name ends in *.o* to one whose name is appended with a `_bkup` extension. It shows how us-

ing file globs can save you lots of time, and it also introduces the `cp` command (See "Creating, Removing, and Copying Files and Links" on page 81 for more details). Notice too that braces were used to enclose the variable named `file` in the third example, because `file` is a variable name but `file_bkup` is not. The braces are needed so that `bash` can expand `$file` rather than trying to expand `$file_bkup`.

The second form of `for` loop is like the standard C iterative `for` loop except that double-parentheses are needed:

```
$ for (( i = 0; i <= 10; i++ )) ; do echo $i ; done
1
2
--snip--
10
```

This prints the numbers 1 through 10, one per line, whereas in this example:

```
$ for (( i = 0; i <= 10; i++ )) ; do echo -n -e "$i\t"; done; echo
1    2    3    4    5    6    7    8    9    10
```

the numbers 1 through 10 are printed on one line tab-separated. The `-n` option tells `echo` to suppress the newline, and the `-e` option turns on interpretation of backslashed characters such as `\t` (the tab character). The `echo` after the keyword `done` adds a newline so that the prompt appears on a new line.

The branching construct in Bourne-like shells such as `bash` works as follows:

```
if list-of-commands
then
    true-branch-list-of-commands
[else
    false-branch-list-of-commands
]
fi
```

The square brackets indicate that the `else` part is optional; they aren't part of the command. The `then` part is executed if the last command in the *list-of-commands* is successful, in other words, if its exit status is zero. If there is an `else` part, it's executed if the last command's exit status is non-zero. For example, the following command

```
$ if ls /opt/info &> /dev/null
 then
    echo /opt/info exists
 else
    echo ls failed
 fi
```

prints `/opt/info exists` if */opt/info* exists and prints `ls failed` if it doesn't.

The `if` construct was originally designed to require a list of commands as the condition to be evaluated. To branch based on a conditional expression instead, you had to enter the `test` command after the `if` keyword. The `test` command was a separate program created specifically for this purpose. But `bash` also has a built-in command of the same name now. Its options are slightly different from the `test` program. To determine which one you get when you enter **`test`**, enter the command

```
$ which test
```

The output will be the pathname to the file or will indicate that it's a shell builtin. To override the default, you can enter the full pathname, such as **`/usr/bin/test`** to use the `test` program, or you can enter **`builtin test`**, to use the shell builtin version of `test`. In either case, the `test` command is followed by a conditional expression. When you run it, its return value is the truth value of the expression, either `true` or `false`. As an example, consider the following:

```
$ if test -x myfile
  then
      echo myfile is executable
  else
      echo cannot run myfile
  fi
```

This demonstrates how to use the `test` command with the `-x` operator, which checks whether a file exists and you are able to execute it.

Because there are slight differences between the two versions of this command, and you may not know which one you're running, I'll show you an alternative syntax for the `test` command. Instead of writing

```
    test expression
```

you can write

```
    [ expression ]
```

in which the square brackets are part of what you enter. The preceding example is therefore equivalent to

```
$ if [ -x myfile ]
  then
      echo myfile is executable
  else
      echo cannot run myfile
  fi
```

To confound things further, `bash` also has a `[[ ]]` operator that has a more extensive set of expressions. Consult the man pages for further information about it.

The kinds of conditional expressions that you can form are very extensive:

- File tests
- Numerical comparisons
- String comparisons
- Tests of the state of variables
- Negation, conjunction (and-ing), disjunction (or-ing) of any of them

File tests include tests of whether a file is readable, writeable, executable, is a directory, is a regular file, and so on. There are more than 20 such tests. Below are a few examples.

**`test -x myfile`** true if `myfile` exists and is executable by you

**`test -L myfile`** true if `myfile` exists and is a symbolic link

**`test -s myfile`** true if `myfile` exists and is not empty

**`test file1 -nt file2`** true if `file1` is newer than `file2`

String tests use the operators `=` and `!=`. If you use the `[[ ]]` syntax, you can also use `<` and `>` for string comparisons. The comparisons are based on the value of your `LC_COLLATE` environment variable, which determines the lexicographical ordering of characters. This means that `"Elephant" < "ant"` is true if your settings sort uppercase before lowercase and false otherwise. `LC_COLLATE` is one of several environment variables that are collectively known as your *locale*. Locales are introduced briefly in Chapter 3 and in more depth in Chapter 4.

The best way to learn about the various conditional expressions is to enter the command **`help test`** and study the full list. The following list contains the expressions you'll probably find most useful:

**`-a myfile`** True if `myfile` exists.

**`-d myfile`** True if `myfile` is a directory.

**`-e myfile`** True if `myfile` exists.

**`-r myfile`** True if `myfile` is readable by you.

**`-s myfile`** True if `myfile` exists and is not empty.

**`-w myfile`** True if `myfile` is writable by you.

**`-x myfile`** True if `myfile` is executable by you.

**`-N myfile`** True if `myfile` has been modified since it was last read.

**`-z string`** True if `string` is empty.

***`string1 = string2`*** True if the two strings are equal.

***`string1 != string2`*** True if the two strings are not equal.

**`! `*`expr`*** True if *expr* is false.

***`expr1 -a expr2`*** True if both *expr1* AND *expr2* are true.

***expr1 -o expr2*** True if either *expr1* OR *expr2* is true.

***arg1 op arg2*** Where *arg1* and *arg2* are numeric and *op* is one of `-eq -ne -lt -le -gt -ge`.

## Shell Scripts

If you find yourself repeating a particular sequence of commands frequently, you can make your work more productive by putting that sequence of commands into a file for later execution. We call a file containing a sequence of commands written in a shell language a *shell script*. If the commands are written in `bash`, it's called a `bash` script. If it's written in the C shell language, it's called a `csh` script, and so on.

Once you've created a shell script, and assuming that you've taken a few steps to make it an executable file, you can enter its name on the command line to run that script. For example, if you've written a shell script named *myfirstscript* in the current working directory, you would enter

```
$ ./myfirstscript
```

to run its commands.

Although most of the shells have very similar syntax rules, there are variations among them and what's true about the `bash` shell is not necessarily true about the Korn shell or the C shell. Therefore, to be clear, everything in this section is specifically about `bash`, although much of it may be true of other shells.

Here's a simple example that illustrates two key ideas:

*script1*

```
#!/bin/bash
# This script does nothing except print "Hello world; code responsibly!"
# Its purpose is to show what the first line looks like and
# to introduce comments.
#
# Written by Stewart Weiss, Jan. 1, 2000

echo "Hello world; code responsibly!"
```

The first line tells the shell to run the interpreter `/bin/bash` using the rest of the file as its input. In effect it says, "I am a bash script." Although that line is not strictly necessary, if you want to be able to run scripts just by entering their names, you need it. Its form is

```
#! pathname-of-interpreter
```

In this case the interpreter program is `/bin/bash`. If you want to write a Perl script, the first line would be:

```
#!/usr/bin/perl
```

That two character symbol, `#!`, is called the *shebang* symbol by those in the Unix world.

The next thing to note is the # comment delimiter; bash ignores anything after it on the same line (except when the second character after it is !.) Most shells use this comment delimiter.

Your scripts should always contain comments; scripts are programs, and like all programs they should be documented for all the same reasons.

In order to be able to run a script by entering its name, you also need to make that file *executable*, which means adding execute permission to the file's permission mode. Use the chmod command to alter the permissions on any file that you own. For example:

```
$ chmod +x filename
```

This makes *filename* executable by everyone.

You can pass arguments to scripts on the command line, to be used inside the script, which usually makes the script more versatile. This is where certain bash parameters come into play. Parameters denoted by numbers other than 0 are called *positional parameters*. Thus, 1, 2, 3, and so on, are positional parameters. When you enter a command, bash stores the successive words from the command line in successive positional parameters (1, 2, 3) up to the number of distinct words it finds on the command line. (If you enclose a string in single quotes, it will treat that entire quoted string as a single word even if there are spaces in it.) The very first word on the command line, which is the program to run, is stored in parameter 0. Thus, in a script you can access the command line arguments with the notation $1, $2, $3, and so on, and the program name with $0. If you enter more than nine arguments, you need those curly brackets to enclose the parameters whose numbers are more than one digit, such as ${12}.

Two other parameters that are useful in scripts are $# and $*. Your script can access the number of arguments on the command line with the parameter $#, and the entire set of words after the command in $*.

The following script, which is stored in an executable file named *script2*, demonstrates how these two parameters can be used:

*script2*
```
#!/bin/bash
# This script demonstrates the use of positional parameters.

echo "The command that you typed is $0."
if [ $# -gt 0 ] ; then
 ❶ echo -n "The first word after the command is '$1' "
   echo "and number of words is $#"
   echo "The entire set of words is:"
   echo "'$*'"
else
   echo You did not enter anything after $0.
fi
```

It uses the -n option to echo ❶, which tells it not to print a newline, so that the next output will be on the same line. It also uses a test to check whether any words appeared after the command name.

When we run it as follows:

```
$ ./script2  This is a test
The command that you typed is ./script2.
The first word after the command is 'This' and the number of words is 4
The entire set of words is:
'This is a test'
```

we see that it displays the first command line argument, the number of command line arguments, and the entire set of arguments. We now run it without any command line arguments:

```
$ ./script2
The command that you typed is ./script2.
You did not enter anything after ./script2.
```

The script detects that there are no arguments and lets me know this. It would be even better if it told me how to use it correctly, with some sort of statement such as:

```
    usage: script2 [list of words]
```

For more information about scripting, read the man page of the shell you are using.

### Shell Behavior, Shell Variables, and the Environment

Shell variables were introduced and described earlier in this chapter in "Parameters and Variables". Many of these variables affect the behavior of the shell, such as `IFS`, which stores a list of characters that `bash` treats as whitespace, and `PS1`, which defines the prompt that `bash` displays for you.

Some of `bash`'s variables are part of the environment when `bash` starts running. They're made available to `bash` when it starts. For example, the environment variable named `HOME`, which has the absolute path name of the user's home directory, is also a `bash` variable. Another environment variable named `PATH` is a colon-separated list of pathnames of directories that the shell searches to find command names that are entered. This `PATH` variable is also accessible in `bash`.

An easy way to understand the difference between shell variables and environment variables is by analogy to the difference between global variables and local variables in a program. Global variables are defined outside of the program, in file scope. They are like environment variables, which exist outside of the shell. In contrast, locals are defined inside the scope of the program; they're analogous to shell variables. Just as both global and local variables are accessible to the program, environment and shell variables are accessible to `bash`. Like globals and locals, they're stored in different places and receive initial values in different ways.

You can see the full list of bash's shell variables by entering the set command. You can see the value of any bash variable, whether it's from the environment or not, with the echo command, as in:

```
$ echo Number of this command in history is $HISTCMD
Number of this command in history is 528
$ echo $HOSTNAME is running $OSTYPE on machine type $MACHTYPE
harpo is running linux-gnu on machine type x86_64-pc-linux-gnu
$
```

The POSIX.1-2017 standard [17] lists 16 different environment variables that can influence the shell's behavior.

When you first log in, the kernel initializes your environment and starts up a new process for your login shell, passing it a reference to that environment. It gets the initial values of your environment variables from a few different sources, including various system files such as */etc/environment*, */etc/profile*, and */etc/bash.bashrc*, as well as user specific files in your home directory, such as *.bash_profile* and *.bashrc*.

Shell variables that do not come from the environment, such as OSTYPE, MACHTYPE, and HOSTNAME, get their initial values from other sources, such as configuration files.

Once you've logged in and you're working within a shell, when you run a command or any program, a new process is created and it inherits the environment of the shell from which you ran it. More generally, whenever a new process is created, it inherits a copy of its parent's environment, even if the parent is not a shell. This is how programs know what their current working directories are, for example, or which users are their owners, or the values of various flags and masks. The environment is a key element in making things work. It's like a set of global variables available to all processes.

You're free to customize the environment of a shell by defining new variables or redefining the values of existing variables. The syntax is shell-dependent. In the Bourne shell and bash, variables that you define, which we'll call *user variables*, are not automatically placed into the environment. To place them into the environment, you have to *export* them using the export command. Exporting a variable essentially makes that variable available to all processes that you run afterward, including any subshells. For example, if you frequently visit a directory that has a long pathname and you want to access it easily from anyplace in the directory hierarchy, you could create a variable that stores the pathname, such as the following:

```
$ export LECTURES=/home/stewart/teaching/unixclass/lecturenotes/
```

This command puts LECTURES into the environment with the value
/home/stewart/teaching/unixclass/lecturenotes/
so that it can be inherited by future commands and processes. This command does the same:

```
$ LECTURES=/home/stewart/teaching/unixclass/lecturenotes/; export LECTURES
```

Now you can navigate to that directory by entering **cd $LECTURES**. The convention is to use uppercase names for environment variables and lowercase names for variables that you define and do not export. The following session demonstrates what happens when you don't export and when you do:

```
$ WORKDIR=/home/stewart/scratch
$ echo $WORKDIR
/home/stewart/scratch
$ bash      # Start a new shell
$ echo The value of WORKDIR is $WORKDIR
The value of WORKDIR is
$ exit      # Exit new shell to previous one
$ export WORKDIR
$ bash      # Start a new shell again
$ echo The value of WORKDIR is $WORKDIR
The value of WORKDIR is /home/stewart/scratch
```

We started by assigning a value to `WORKDIR` without exporting it. It is a `bash` variable but not in the environment. When we start a subshell by entering **bash**, it does not get a copy of this variable, so the `echo` command outputs an empty string for `$WORKDIR`. We exit the subshell and this time, export the variable `WORKDIR`. Now we start a subshell again and run `echo`, passing `$WORKDIR`. Now it sees it and can output its value.

## Type of Commands

Commands are not always executable programs stored in files. In general, a command might be any of the following types:

- A *binary executable file*, such as `ls`, which is the file */bin/ls*.
- A *shell builtin* (defined in Chapter 1, in the "Shells" section on page 12), such as `alias`, `cd`, and `exit`.
- An *alias* (alternate name) for another command, defined either by you or by an administrator of the computer using the `alias` builtin command, as in `alias rm='rm -i'`, which replaces the dangerous `rm` command with a version of it that always asks before deleting anything.
- A *script*, such as a shell script or a script in another scripting language. Examples include `ruby`, `Javascript` (`js`), and `perl`.

Sometimes the same command name can denote more than one of these. For example, commands like `pwd` and `echo` are both shell builtins and executable files. You can determine the type of the command using the `type` `bash` builtin, which takes one or more command names:

```
$ type who ls cd backup
who is /usr/bin/who
ls is aliased to `ls -F --color=tty'
```

```
cd is a shell builtin
backup is /home/stewart/bin/backup
```

This tells you whether it's a file, an alias, or a builtin.

If a command is reported by `type` as being a file, you can use the `file` command to determine whether it's a binary file or something else:

```
$ file /home/stewart/bin/backup
/home/stewart/bin/backup: Bourne-Again shell script, ASCII text executable
```

The `file` command attempts to detect exactly the type of file. In this case it read the first line and found that it started with the `#!/bin/bash` command.

## File Permissions

Before we explore the various commands for working with files, you need to understand how file permissions work. In this context the word *file* refers to all file types, including directories.

Unix has a very simple but powerful method of protecting files. To provide a way for users to control access to their files, the inventors of Unix devised a rather elegant and simple access control system:

- A file has an owner, called its *user*. The owner has *user access* to the file.
- A file is also associated with a group of users, called its *group*. Usually the owner is a member of the file's group. The owner of a file can assign any of the groups to which the owner belongs to that file (with the `chgrp` command.) Users in that group other than the owner have *group access* to the file.
- Everyone who is neither the user nor a member of the file's group is in the class known as *others* and has *others* access to the file.

Thus, the set of all access classes is partitioned into three disjoint subsets: user access, group access, and others access, which you can remember with the acronym *UGO*.

Every file has three modes of access: read, write, and execute.

***Read access*** The ability to view file contents. For directories, this is the ability to view the contents of the directory using the `ls` command.

***Write access*** The ability to change file contents or certain file attributes. For directories, this implies the ability to create new links in the directory, to rename files in the directory, or remove links in the directory. This might sound counterintuitive. The ability to delete a file from a directory does not depend on whether one has write privileges for the file, but on whether one has write privileges for the directory.

***Execute access*** The ability to execute the file. For a directory, execute access is the ability to `cd` into the directory and as a result, the ability to run programs contained in the directory and run programs that need to

access the attributes or contents of files within that directory. In short, without execute access on a directory, there is little you can do with the files contained in it.

For each of the three classes of users, three protection bits define the read, write, and execute privileges afforded to members of the class. For each class, if a specific protection bit is set, then for anyone in that class, the particular type of access is permitted. Thus, there are three bits called the **r**ead, **w**rite, and e**x**ecute bits for the user (`u`), for the group (`g`), and for others (`o`), for a total of nine bits. These bits, which are called *mode* or *permission bits*, are usually expressed in one of two forms: as an octal number, or as a string of nine characters.

The nine-character permission string is best understood as three groups of three characters, where the characters representing the high order bits are to the left:

```
r w x        r w x         r w x
user bits    group bits    others bits
```

In a permission string, we usually represent a set bit with the letter associated with it, and an unset bit with a hyphen. Many commands use this convention as well. Some examples are shown here:

**`rwxrw-r--`** The user (the owner) has read, write, and execute permission, the group has read and write but not execute, and others have only read permission.

**`r-xr-xr-x`** Everyone has read and execute permission but not write permission.

**`rwxr-xr--`** The user has read, write, and execute permission, the group has read and execute permission, and others, only read access.

**`rw-r-----`** The user has read and write access, the group has read access only, and no one else can do anything with the file.

The mode string can also be represented as a three-digit octal number, by treating each group of three bits as a single octal digit. We can use the C ternary operator, `?:`, to express the conversion of permission letters to an octal value as

```
octal_value = r?4:0 + w?2:0 + x?1:0
```

In other words, the read bit is 4, the write bit is 2, and the execute bit is 1, which results in the following table of values for each group of three bits:

```
rwx   rw-   r-x   r--   -wx   -w-   --x   ---
 7     6     5     4     3     2     1     0
```

For example, to compute the octal number for the mode `rwxrw-r--`, break it into three substrings `rwx rw- r--` and then use the above conversions on each substring: the user sum (`rwx`) is 7, the group sum (`rw-`) is 6 and the others sum (`r--`) is 4, resulting in octal number 764.

In addition to the mode bits, a file's permission string is usually displayed with a single character code that represents the file type. This character is written to the left of the mode string and can be one of the following:

- `-` A regular file
- `d` A directory
- `b` A buffered special file
- `c` A character special file
- `l` A symbolic link
- `p` A pipe
- `s` A socket

The ones you're most likely to see are directories, regular files, and symbolic links.

## Viewing and Modifying File Attributes

You'll discover that some of the commands you try to use fail because you don't have the permission to use them. Knowing how to view and change permissions will help you avoid this problem.

To see several attributes of a file, including its mode, use the `-l` option (for ***l****ong listing*) to the `ls` command. When `ls` is given this option, for each file, it prints the mode string, the number of links to the file, the username of the owner, the group name of the group, the number of bytes in the file, the last modification time, and the file's name in the given directory. For example, to look at the attributes of the file named *meeting_notes* in the current working directory, enter the following:

```
$ ls -l meeting_notes
-rw-r--r-- 1 stewart faculty 3304 Sep 22 13:05 meeting_notes
```

This file is a regular file (type is `-`) and can be read and modified by me, its owner (`rw-`). Anyone in the `faculty` group (`r--`) can read it, and anyone else (`r--`) can read it.

The `ls` command has many other options for controlling the information that it displays. Read its man page for details.

You can also use the `stat` command to display even more attribute information, as shown here:

```
$ stat meeting_notes
  File: `meeting_notes'
  Size: 3304     Blocks: 8   IO Block: 4096   regular file
Device: 18h/24d Inode: 1318        Links: 1
Access:(0644/-rw-r--r--)Uid:(1220 sweiss)Gid:(400 faculty)
Access: 2010-12-20 13:20:04.582733000 -0500
Modify: 2010-09-22 13:05:11.271251000 -0400
Change: 2010-09-22 13:05:11.278893000 -0400
```

For another way to access size data, you can get a count of bytes, words, and lines with `wc`:

```
$ wc meeting_notes
156 387 3304 meeting_notes
```

The output shows that there are 156 lines, 387 words, and 3,304 bytes in the file. Other options provide different information.

The most important commands that alter the attributes of a file are:

**`chmod`** ***`mode [,mode]... files`*** Change the permissions on *`files`*

**`chown`** ***`owner files`*** Change the ownership of *`files`*

**`chgrp`** ***`group files`*** Change the group ownership of *`files`*

**`touch`** ***`files`*** Update timestamps of *`files`* or create empty ones if they don't exist

Of these, `chmod` is the one you'll need the most. The command's name is a mnemonic: ***ch****ange* ***mod****e*. The `chmod` command has a complex set of options, and you can use it with permissions as letters or octal numbers.

In the simplest case, a mode is of the form

```
user-designation  operator    type-of-permission
```

where the user designation is one or more of the letters `ugoa`, the operator is one of `+-=`, and the type of permission is one or more of `rwx`. No spaces are allowed between these parts! The letter `a` is for *all users* and is short for `ugo`. The `+` operator adds the given permissions to the designated users, and the `-` operator removes those permissions. The `=` operator sets the permissions to the ones specified, removing any others that might have existed. Examples follow, using the single file named *myfile* to illustrate:

**`chmod u+rw myfile`** Adds read and write permission for owner of `myfile`.

**`chmod g+rx myfile`** Adds read and execute permission for group of `myfile`.

**`chmod go-w myfile`** Removes write permission for everyone in group and others for `myfile`, so only the owner can modify `myfile`.

**`chmod u=rwx myfile`** Sets the mode to `rwx------` for `myfile`.

**`chmod u=rw,og=r myfile`** Sets read and write for owner, read-only for everyone else.

**`chmod a-x myfile`** Removes execute permission on `myfile` for all users.

**`chmod +x myfile`** Absent any user designation, applies to all, so this adds execute permission to `myfile` for everyone.

**`chmod 755 myfile`** Gives `myfile` permission.`rwxr-xr-x`

**`chmod 640 myfile`** Gives `myfile` permission `r-xr--r--`.

I've not described all the ways we can use the `chmod` command. Read its man page to learn more.

The `touch` command is pretty convenient for creating empty files, especially when you're using the `make` command, since it will come in very handy for forcing files to be recompiled. The `make` command is described in Appendix F. When I need a bunch of empty files, I also use `touch`:

```
$ ls    # List the directory to see files don't exist yet.
$ touch f1 f2 f3 f4 f5  # Create the files
$ ls -l
-rw-r--r-- 1 stewart faculty 0 Jan  9 12:40 f1
-rw-r--r-- 1 stewart faculty 0 Jan  9 12:40 f2
-rw-r--r-- 1 stewart faculty 0 Jan  9 12:40 f3
-rw-r--r-- 1 stewart faculty 0 Jan  9 12:40 f4
-rw-r--r-- 1 stewart faculty 0 Jan  9 12:40 f5
```

This shows that `touch` created five empty files within the same minute of time.

## Working with Directories

You don't need to know many commands in order to do most directory-related tasks in Unix. The principal tasks fall into these categories:

**Navigation** Changing or identifying the current working directory

**Display** Showing the contents of directories

**Creation** Creating new directories

**Restructuring** Renaming, moving, and removing directories

**Content modification** Changing the content of directories

Following is a list of the basic commands with explanations of the simplest forms of usage. I don't describe the various options or details of each command's usage. For that you should read their respective man pages.

**`pwd`** Print the absolute pathname of the working directory. The *p* in `pwd` stands for *print*, but it does not print on a printer. In Unix, *printing* means displaying on the screen. (That's why the C instruction `printf()` sends output to the display device, not the printer.)

**`cd [`*`dir`*`]`** Change the working directory to *dir*, if *dir* is given, and to the home directory if no argument is given. The `cd` command (think *change directory*) changes your current directory. You give it the name of a directory, either as an absolute pathname or as a relative pathname.

**`ls [`*`dir1`*`] [`*`dir2`*`] ...`** Display the contents of *dir1*, *dir2*, and so on, if they're supplied; otherwise, display the contents of the current working directory.

**`mkdir `*`dir1`*` [`*`dir2`*` ...]`** Create new directories named *dir1* (and *dir2*, and so on). If *dir1* (and *dir2*) are filenames without slashes in them, they are created in the current working directory; otherwise, they're considered to be pathnames, and they're created in the directories specified by their

paths. The `mkdir` command is the only one that creates a directory. If the name you choose already exists, `mkdir` will fail.

**`rmdir` *`dir1`* `[` *`dir2`* `] ...`** Remove the *empty* directory *dir1* (and *dir2*, and so on). You cannot use `rmdir` to remove a non-empty directory. See the next command.

**`rm -r` *`dir1`* `[` *`dir2`* `] ...`** Remove all links in all directories *dir1* (*dir2*, and so on), and remove the directories themselves. This command deletes one or more directories and their contents, but it's not reversible, so be careful. There is no "trashcan" associated with `rm` from which you can restore deleted files. The `rm` command, without the `-r` option, is described in the next section. Commands that delete files, such as `rm`, and commands that create or rename files, are actually modifying directories (again, they're covered in the next section).

**`mv`** This command renames or *moves* both files and directories (also described in the next section,"Working with Files").

Changing directories and being in a directory are imprecise phrases. When you `cd` to a directory named *dir*, you may think of yourself as being *in dir*, but that's not true. What is true is that the *dir* directory is now your current working directory, and every process you run from the shell in which you changed directory, including the shell process, will use *dir* by default when it's trying to resolve relative pathnames.

## Working with Files

We classify commands for working with files as those that display file contents but do not modify them, those that view file attributes, and those that work as editors that can modify the files in one way or another. Editors are a separate topic, which we don't cover in this book.

### *Displaying Files*

The following list shows the basic commands for viewing the contents of files, not editing or modifying them. For all of the commands listed here, if the file arguments are omitted, the command reads from standard input, and if the argument list can be multiple files, the command processes them in the order they appear on the command line. In Appendix D, I describe a class of commands called *filters*, which are programs that provide sophisticated filtering of ordinary text files.

**`cat [`*`files`*`]`** Display all file contents.

**`more [`*`files`*`]`** Display all file contents a screen at a time.

**`less [`*`files`*`]`** Display all file contents a screen at a time with more options.

**`head [-`*`n`*`] [`*`file`*`]`** Display the first *n* lines of a file; the default is n = 10.

**`tail [-`*`n`*`] [`*`file`*`]`** Display the last *n* lines of a file; the default is n = 10.

**`diff` *`file1 file2`*** Compare two files line by line.

**`cmp` *`file1 file2`*** Compare two files byte by byte.

The names of the `more` and `less` commands have an interesting story. The `more` command, which existed long before there was a `less` command, was created to allow you to read a page at a time, and also to jump ahead in the file with regular expression searches, using the same search operator from `vi`. However, it wasn't easy to go backward in the file with `more`. Navigation is better in modern versions of `more`.

The `less` command was written and released by Mark Nudelman in 1985 to do more than `more`, but someone he knew suggested the name `less` (see *http://www.greenwoodsoftware.com/less/faq.html#history*). I like to think that the person who named it was playing with the often quoted adage by Ludwig Mies Van der Rohe, the German-born Modernist architect, "Less is more," intended to convey that the less there is, the better. Maybe not, but no matter why they named the enhanced version of `more`, `less`, `less` does more than `more`. The `less` command is the most versatile means of viewing files, and you should learn how to use it.

The `diff` command comes in handy when you want to know whether two files are exactly the same, and if not, how they differ. By default, if they're identical, `diff` returns with no output. If they differ, it displays the differences. If you use the `-s` option to `diff`, it displays a message when the two files are the same, which is sometimes preferred. A similar command is `meld`, which presents the two files side-by-side in a separate pop-up window. The `cmp` command is useful for comparing binary files.

### Creating, Removing, and Copying Files and Links

When you create, remove, or rename a file, you're modifying the directory in which that file's name appears. Therefore, you need to have write permission on any directory in which a file is created or removed, regardless of which command does it. In fact, technically all of the commands in the list that follows should be categorized as directory modification commands, but because they also have effects on files, I single them out here. Almost all of these commands have more than one form, but only a few of the different forms are listed.

**`ln` *`f1 f2`*** Creates a new link for file *`f1`* named *`f2`*.

**`ln` *`f1`* `[` *`f2 ...`* `]` *`dir`*** Creates a new link for each file, with the same name, in *`dir`*.

The `ln` command can be used to create new names only for files, not directories. If the new name already exists, it displays an error message. You can read about this command's many options on its man page. A simple version of `ln` is named `link`.

**`rm` *`files`*** Deletes the links for all given *files*. The `rm` command is irreversible; once a link is removed, it cannot be recovered. This command

does not remove directories unless the `-r` option is given to it, in which case it deletes an entire hierarchy rooted at the given directory.

**`mv` *`file1 file2`*** Renames *file1* with the *new* name *file2*. Note that, if *file2* is an existing name of a non-directory file, it will be replaced, provided that *file1* is not a directory. If *file2* is an existing directory, this is the form of `mv` that follows next. Because `mv` will silently overwrite existing files, you should use `mv -i`, which prompts you before overwriting them.

**`mv` *`files dir`*** Moves all files into the destination directory *dir*.
In this form, the last argument must be an existing directory name, and all preceding arguments are moved into that directory. The preceding arguments can be existing directories, as long as moving them does not create a cycle in the directory hierarchy, such as trying to make a child into a parent.

**`cp` *`file1 file2`*** Copies *file1* to a file with the name *file2*.

**`cp` *`files dir`*** Copies all files into the *dir* destination directory.
The `cp` command works almost the same way as the `mv` command, except that it replicates files instead of moving them. The main difference is that `cp` does not accept directories as arguments, except as the last argument, unless you give it the `-r` option. In other words `cp -r` recursively copies the directories. Note that `cp` makes copies of files, not just their links, so changes to either file are not reflected in the other.

We'll go through some examples to demonstrate how these commands work.

```
$ cd branch                     # Change working dir to branch
$ ls                            # Display current contents of branch
project.c
$ mv project.c finalversion.c   # Rename project.c
$ ls                            # Display branch again to see the changes
finalversion.c
```

This shows that the `mv` command changed the link *project.c* to *finalversion.c*. We need to use `ls` to see that it did this because we've really just changed the *branch* directory by replacing the link named *project.c* with a link named *finalversion.c*.

In contrast, consider this:

```
$ cd master
$ ls
io.c io.h main.c ui.c  ui.h utils.c  utils.h
$ mkdir headers; mv *.h headers
$ ls
headers io.c main.c ui.c   utils.c
```

In this case, we created a new directory named *headers* in directory *master* and used `mv` not to rename files, but to move their links into *headers*.

In this example

```
$ rm hwk1.o project.o main.o
```

the `rm` command removes three file links from the current working directory. If these files have no other links in other directories, the attributes and contents are also deleted from the filesystem, otherwise just the names are removed.

In this example

```
$ ln project.c main.c utils.c /sources/myrepos/project2/
```

we use `ln` to create a new link for each of *project.c*, *main.c*, and *utils.c* in the */sources/myrepos/project2/* directory with their same names.

This last example

```
$ cp -r main.c utils.c utils.h images ../version2
```

copies the three files *main.c*, *utils.c*, and *utils.h*, and the directory *images*, into the directory *../version2*.

## Summary

A shell is a programmable, substitutable, command line interpreter with many useful and powerful features. The shell is not part of the kernel but instead runs in user space, the layer above it.

Modern shells provide I/O redirection, interactive command line editing, a history mechanism, filename substitution, control flow mechanisms (such as loops, branching, and selection statements, backgrounding, and job control), aliasing, and scripting. Some commands are built into the shell so that files do not need to be loaded to execute them.

All files have three different types of access: read, write, and execute. The permissions for each type of access can be granted independently to the owner of the file, to a group of users associated with the file, and to everyone else. You need to learn a relatively small set of commands in order to perform routine tasks such as navigating through the directory hierarchy, creating and maintaining files and directories, and viewing and changing their permissions.

To learn more about using `bash`, read the *Bash Reference Manual* [19]. You can download latest version as a PDF from the GNU website (*https://www.gnu.org/home.en.html*). You can also read the `bash` man page or the Info-Tex page for `bash`.

## Exercises

1. Look at the man page for the `shuf` command, and then write a command that generates a random permutation of the integers from 1 to 100 in a file named *permutation100* in your current working directory.

2. Write a script named *check_procs* that runs the `ps` command every 10 seconds a total of five times.
3. Modify the preceding script so that it accepts as a command line argument the number of seconds between runs of the `ps` command.
4. Write a sequence of two commands that creates an empty file named *shopping_list* that you can read and write but that cannot be read or written by anyone else.
5. Using nothing but the `cat` command and redirection, show how to add the lines

```
milk
bread
eggs
flour
sugar
```

   to the *shopping_list* file. (Hint: pressing CTRL-D terminates input.)
6. Read the man page for `chmod` and write octal expressions that represent the following permissions.
   (a) Owner can read and write; group can write but not read; and others can read.
   (b) Everyone can execute, and the owner can read and write.
   (c) The owner can read; the group can read, and others can't do anything.
7. Write a shell script that makes a copy of every file in the current working directory with the same name and an extension *.copy*. For example, the file named *myfile* would have a copy named *myfile.copy*.

# 3

# FUNDAMENTALS OF SYSTEM PROGRAMMING

The first two chapters provide a basic foundation for working in Unix as an ordinary, non-programming user. In this chapter we turn our attention to the core concepts that you need to understand, not just for writing programs in general, but for writing system programs. I'll cover several basic concepts of programming in a Unix environment as well as concepts specifically related to system programming. In addition, I'll explain how to solve certain problems that are common to all projects, such as handling errors, parsing and checking the command line, and obtaining environment strings.

We'll start with the topics related to general programming and then move on to the fundamentals of system programming, including system calls, the relationship between system calls and libraries, handling system call and library function errors, and making your programs portable.

Whereas ordinary programs typically interact only with a user and the user's files, system programs usually interact with many different types of

system resources. Writing them requires a deeper understanding of the system interfaces to these resources as well as their structure and purpose.

Here are a few examples that show why you need to know how system programs work. If your program has to acquire data from a shared resource, such as a file that another process may have already opened, you'll be better equipped to write it if you know what happens when files are opened, how Linux manages open files, and how processes interact with them. If your program has to control a different type of shared resource, such as a terminal window, you'll need to understand how terminals work.

A completely different problem is how to design a program that might get a delivery of data not when it asked for it, but at some later, unpredictable time. This is an example of *asynchronous I/O*, which we explore in Chapter ??. Similarly, a process may need to pause its execution until some other process has performed some other task, as when a program plays a game of chess against another program. Neither program is allowed to make a move until the other has finished its turn. Writing these types of programs requires familiarity with the system resources for *process synchronization*.

## Object Libraries

Most likely, almost every program you've written has made calls to functions you didn't write, but that are part of some library installed on your system. The functions that you call to read from or print to the screen are contained in a library, most likely a standard library, such as the C Standard Library (if you used `printf()`, for instance) or the C++ Iostreams Library (if you used the insertion operator of the `ostream cout` object). You may not have thought much about libraries before, but they play a key role in programming.

When you've been writing programs for a while, you might realize that you keep writing certain functions over and over again for different projects. To avoid rewriting them each time, perhaps you copy them from one directory to another, possibly tweaking them a bit depending on how you plan to reuse them. Suppose you discover while working on your latest project that one of the functions you're reusing in this way has a bug. You can fix it in the current copy, but then you'll have to find all those other projects that use that function and fix the bug in them as well. It's not a very efficient organizing principle.

Wouldn't it be better if you could create a repository of those frequently used functions in such a way that each new project could just link to it? Although such a repository could be a collection of source code files, it would be even better if it were a bundle of *object modules*, code that's already compiled and ready to link into a program.

One advantage of an object code bundle instead of a source code bundle is that you don't have to compile it every time. Also, if you plan on sharing your work with others, you could distribute the object code and not worry that it might be modified, unintentionally or otherwise, or possibly broken. Those issues are possible if you distribute just the source code. If you did distribute the object code, you'd most likely need to distribute a

header file that contained all declarations of the functions and other symbols contained in the object code. In Unix systems, doing so isn't just possible, it's also relatively easy. Unix has tools that let you create your own libraries and tools that can view and modify libraries. Appendix A contains detailed instructions on how to create libraries in Unix.

An *object library*, also called a *software library*, is a file that bundles together, in a structured way, the compiled object code from multiple functions so that programs can call them easily. Libraries aren't stand-alone executables; they don't have a `main()` function, and you can't run them. They contain function implementations and sometimes type definitions and constants needed by those functions or by code that calls them. Figure 3-1 depicts a hypothetical library named *libsnw.a* containing three object modules.

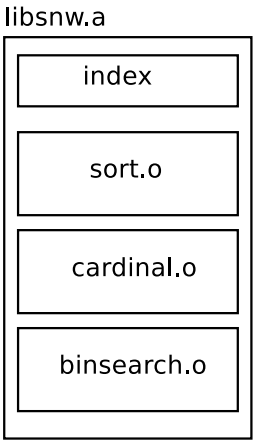



*Figure 3-1: A small object library with three modules and an index that serves as a table of contents*

The index in Figure 3-1 is essentially a look-up table that contains the addresses relative to the start of the file of all symbols defined in the library, which makes those symbols easy to find.

System calls are usually very low-level primitives. They do very simple tasks, because the Unix operating system was designed to keep the kernel as small as possible. For the same reason, the kernel typically doesn't provide many routines that do similar things. For example, the kernel has a single function to perform a read operation, and when it reads from storage devices such as disks, it reads large blocks of data from a specified device to specified system buffers. It doesn't have a different system call to read a character at a time, nor one that reads formatted input, both of which are useful functions to have. In short, just a single kernel function performs all input operations!

To compensate for the kernel's simplicity, Unix designers augmented the programming interface with an extensive set of higher-level routines that are kept in libraries. These routines provide a much richer set of primitives for programming in Unix. Many library functions ultimately make system calls to the kernel, but some don't because they don't need any kernel ser-

vices. We say that these functions operate entirely *in user space*, meaning that they never need kernel services.

Unix systems also contain libraries for various specialized tasks, such as asynchronous input and output, sharing memory, terminal control, login and logout management, and so on. Using any of these libraries requires that the library's header file be included in the code with the appropriate `#include` directive (for example, `#include <termios.h>`), and sometimes, that the library be linked explicitly because it isn't in a standard place. Volume 3 of the Unix Manual Pages contains man pages for all functions that are part of libraries.

### Static and Dynamic Libraries

Unix systems support two kinds of libraries: static and shared. A *static library*, short for *statically linked library*, is a library whose code can be linked to the program statically, after the program is compiled, to create the program executable file. In other words, the linker copies the library functions referenced by the program out of the library and inserts them into the program executable, after which it resolves all unresolved symbols to enable jumps into and out of those functions. Static libraries in Unix have filenames that end in *.a*, such as *libm.a* and *libc.a*. The *.a* suffix is a reminder that these libraries used to be called *archives*.

In contrast, a *shared library* is a library whose object code is not copied into the executable, but is instead linked to the program at runtime. *Runtime* is the interval of time during which the program is actually running. Because the code in these libraries is linked at runtime, they're also called *dynamically linked libraries*.

**NOTE** *The fact that a shared library is also called a dynamically linked library doesn't imply that they're the same as what Microsoft calls a* DLL. *While DLL is short for* dynamically linked library *also, DLLs are different from Unix shared libraries. I'll use the term* shared library *so as not to cause any confusion.*

With shared libraries, calls to functions or references to other symbols in the library are linked only when the program actually executes the calls or accesses the symbols for the first time. Shared libraries have names ending in *.so*, possibly followed by a numeric suffix of the form *.<number>*, such as *libc.so.6* where the number refers to a specific version. The *.so* suffix is short for *shared object*.

Static linking, which was the original form of linking used in most operating systems, including Unix, resolves references to externally-defined symbols such as functions, by copying the library code directly into the executable file when the executable file is built. The *linkage editor*, also called the *link editor*, or simply the *linker*, performs static linking. The term *linker* is a bit ambiguous, so I avoid using it. The `ld` program is the static linker in Linux.

The primary reason to use static linking, perhaps now the only reason, is that statically linked executables are self-contained and can run reliably on

multiple platforms. For example, a program might use a particular verssion of a graphical toolkit such as GTK that may not be present on all systems. If the toolkit's libraries are statically linked into the executable, the executable can run on other systems with the same machine architecture without requiring that the users on those systems install the specific library files.

When a library is dynamically linked to a program, the linkage editor inserts records into the program for symbols from the library to indicate that these symbols will be resolved when they are first reached during the program's execution. When the program is loaded into memory, the dynamic linker checks whether that library is already in memory and, if not, finds a place in memory for it and loads it. As the program executes, each time a new symbol is reached, the dynamic linker links it to the library. Programs can experience slightly longer running time with dynamic linking because whenever an unresolved symbol is found and must be resolved, there's a bit of overhead in locating the library and linking to it.

Linux systems have two dynamic linkers: `ld.so` and `ld-linux.so`. The former links and loads the old-style executable format known as *a.out* binaries, and the latter links and loads executables in the modern *Executable and Linking Format (ELF)*. ELF is a standard format for executable files, object files, and libraries. It replaces the older *a.out* format and the *Common Object File Format* (*COFF*), which was created to replace *a.out*. ELF was developed by UNIX System Laboratories and has been adopted by almost all Unix vendors.

### The Advantages of Shared Libraries

Shared libraries have several advantages over static ones. One is that, because the executable program file does not contain the code of the libraries that must be linked to it, the executable file is smaller. Its reduced size means that loading into memory is faster and it uses less space on disk as well. Shared libraries also result in more efficient use of memory. Instead of multiple copies of a library being physically incorporated into multiple programs, a single memory-resident copy of the library is linked to each program, reducing the amount of memory in use. Shared libraries are designed so that when processes execute their code, it isn't modified.

Another advantage of linking to shared libraries is that, when a library is updated, the programs that link to it do not need to be modified or recompiled, provided that the library interfaces aren't changed by the update. For example, if bugs are discovered and fixed in these libraries, all that's necessary is to obtain the modified libraries and install them. In contrast, if libraries are statically linked, all programs that use them would need to be recompiled.

Still other advantages are related to security issues. Hackers often try to attack applications through knowledge of specific addresses in the executable code. Methods of deterring such types of attacks involve randomizing the locations of various relocatable segments in the code. With statically linked executables, only the stack and heap address can be randomized; all instructions always have a fixed address in when the executable is run. With

dynamically linked executables, the kernel has the ability to load the libraries at arbitrary addresses, independent of each other, so that library code can have different addresses in each run, which makes such attacks much harder.

### Commands to Query a Library's Contents

We have a choice of commands for seeing what a library file contains. For static libraries, we can use the `ar` command. This command can print a wide range of information; in the simplest case, we can print out a table of contents with its `t` option. We don't need a hyphen before the `t` because technically it's not an option but an *operation code*. For example, to see the objects in the C++ standard library, you can enter:

```
$ ar t /lib/gcc/x86_64-linux-gnu/11/libstdc++.a
compatibility.o
--snip--
array_type_info.o
atexit_arm.o
atexit_thread.o
atomicity.o
bad_alloc.o
--snip--
```

You can also use the `objdump` command to view executable program files and shared libraries. The `-a` option prints the index with information about the original object files:

```
$ objdump -a libsnw.a
In archive libsnw.a:

sort.o:     file format elf32-i386
rw-r--r-- 1220/400   2032 Apr 23 21:56 2007 sort.o

cardinal.o:     file format elf32-i386
rw-r--r-- 1220/400   1580 Apr 23 21:56 2007 cardinal.o

bsearch.o:     file format elf32-i386
rw-r--r-- 1220/400   1784 Apr 23 23:13 2007 bsearch.o
```

The `-t` option limits output to static symbols:

```
$ objdump -t libsnw.a
In archive libsnw.a:

sort.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    df *ABS* 00000000 sort.cpp
00000000 l    d  .text 00000000 .text
```

```
00000000 l    d  .data 00000000 .data
00000000 l    d  .bss 00000000 .bss
00000000 l    d  .gcc_except_table 00000000 .gcc_except_table
00000000 l    d  .gnu.linkonce.t._ZStgtIcSt11char_traitsIcESaIcEEbRKSbIT_T0_T1_ES8_
00000000 l    d  .eh_frame 00000000 .eh_frame
00000000 l    d  .note.GNU-stack 00000000 .note.GNU-stack
--snip--
```

The man page for `objdump` explains how to read its output. For shared libraries, you can use the `nm` command with the `-D` or `--dynamic` option. The following shows how to use it to view the dynamically linkable symbols in the C standard library.

```
$ nm -j -D  /lib/x86_64-linux-gnu/libc.so.6
--snip--
_IO_do_write@@GLIBC_2.2.5
_IO_doallocbuf@@GLIBC_2.2.5
_IO_enable_locks@@GLIBC_PRIVATE
_IO_fclose@@GLIBC_2.2.5
_IO_fdopen@@GLIBC_2.2.5
_IO_feof@@GLIBC_2.2.5
_IO_ferror@@GLIBC_2.2.5
_IO_fflush@@GLIBC_2.2.5

--snip--
```

The `-j` forces `nm` to print just the symbols and suppress other information.

Another tool that you can use is the `readelf` command, which can display the contents of any ELF object file. The `readelf` utility is designed to display information about ELF files in general. On some systems such as Solaris, you need to use `elfdump` because `readelf` is not available.

To understand the output of `readelf`, you need to understand the structure of ELF files and the output of the `readelf` command itself. But if all you want to do is check what functions or other symbols are in an executable, you can enter **`readelf -s` *`elf-file`* `| more`**, and you'll see a large amount of output, a screenful at a time. For example, I can run `readelf` on a program, say *myprogram*, that was linked to a *libutils.so* shared library, and see all symbols, as shown here:

```
$ readelf -s myprogram
Symbol table '.dynsym' contains 17 entries:
Num:    Value Size Type     Bind     Vis      Ndx Name
  0: 00000000    0 NOTYPE   LOCAL    DEFAULT  UND
  1: 00000000    0 FUNC     GLOBAL   DEFAULT  UND ❶ show_time
  2: 00000000    0 NOTYPE   WEAK     DEFAULT  UND __gmon_start__
  --snip--
```

The fact that `show_time` ❶ has a value of 0 means that it is not yet bound to an address. This is to be expected, because the actual binding will not take place until runtime.

To learn more, first read the man page for ELF and then read the page for `readelf`. You can also download the specification of ELF from various websites such as the Linux Foundation (*https://refspecs.linuxfoundation.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/elf-generic.html*).

Two other tools, `hexdump` and `od`, short for *octal dump*, are sometimes useful. Each can display a file's raw, uninterpreted bytes starting from byte 0, with byte addresses, in various output formats such as character when possible, hexadecimal, octal, and decimal.

### Libraries Called by a Program

Another useful tool for determining which shared libraries are linked into your program is `ldd`. You can give it the names of one or more executables or object modules, and it will print those dependencies. The following listing shows how to run it on our `hello` executable, built from the *hello.c* program from Listing 1-2:

```
$ ldd hello
    linux-vdso.so.1 (0x00007ffdbe564000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fc2e26a4000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fc2e28f6000)
```

This shows that `hello` is linked only to the dynamic linker *ld-linux-x86-64.so.2* and the GNU standard C library, *libc.so.6*, as well as a library named *linux-vdso.so.1*. We don't need to know much about this library; it's used by the C Standard Library at runtime to solve some performance issues. Section 7 of the man page for `vdso` explains its purpose in more detail.

Let's look at what dynamic libraries the `ls` program uses:

```
$ ldd /bin/ls
    linux-vdso.so.1 (0x00007ffd591a7000)
    libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007efc6271e000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007efc624f6000)
    libpcre2-8.so.0 => /lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x00007efc6245f000)
    /lib64/ld-linux-x86-64.so.2 (0x00007efc62793000)
```

This output shows that `ls` is linked to two libraries besides the linking loader and the C standard library. The *libpcre2* library has functions for working with Perl regular expressions, and *libselinux* is the SELinux runtime library. *SELinux* is a security system for Linux that defines access controls for the applications, processes, and files.

In Chapter 11 we'll cover the use of the `ltrace` and `strace` tools, which we can use to see which functions from these libraries are actually called when a program runs.

### The Standard C Library

You'll find several different C library implementations across Unix systems, but the one you'll most likely encounter on a GNU/Linux system is the GNU C Library, GNU's implementation of the C Standard Library. People refer to it as *glibc*. The name of the Standard C Library is expected to be *libc.so* on Unix systems, whether it is the GNU implementation or another. You can run the `ldd` command mentioned previously against any C program to see the absolute pathname to *libc.so.<n>*, where *<n>* is the latest version number. In the previous example, *libc.so.6* is the file */lib/x86_64-linux-gnu/libc.so.6*.

On most Linux systems, this file's execute bit is set, so that just by running the file, you'll see version information, as shown here:

```
$ /lib/x86_64-linux-gnu/libc.so.6
GNU C Library (Ubuntu GLIBC 2.35-0ubuntu3.1) stable release version 2.35.
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 11.2.0.
libc ABIs: UNIQUE IFUNC ABSOLUTE
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
```

Because the behavior of some functions in different versions of the library may be different, you might need to design a program so that its execution flow depends upon which library version is installed. GNU/Linux systems provide a way to do this. They have a `gnu_get_libc_version()` function, which returns a pointer to a string containing the version number. Your program can use this number in conditional instructions, so that at runtime, it can alter its behavior depending on the version. Its synopsis is as follows:

```
#include <gnu/libc-version.h>
const char *gnu_get_libc_version(void);
```

The following program demonstrates its use; it just prints out the version number:

*get_glibc_version.c*

```
#include <gnu/libc-version.h>
#include <stdio.h>
int main()
{
    printf("The version of glibc is: %s\n",gnu_get_libc_version());
    return 0;
}
```

We can compile it and run it as follows:

```
$ gcc get_glibc_version.c -o get_glibc_version
$ get_glibc_version
The version of glibc is: 2.35
```

This tells us that we have 2.35 installed on this system.

## System Calls

An ordinary function call is a jump into and return from a routine that is part of the code linked into the program making the call, regardless of whether the routine is statically or dynamically linked to the code. A *system call* is like a conventional function call in that it causes a jump to a routine followed by a return to the caller. But it's significantly different because it's a call to a function that is a part of the Unix kernel.

It's easy to tell whether a function is a system call or a library function. System call man pages are in Section 2, whereas library functions are in Section 3. If when you read the man page for a function, its `SYNOPSIS` shows you that you need to include *unistd.h*, then it's a system call. If the function is in Section 3, the *unistd.h* is not required.

The code that's executed during a system call is actually kernel code. Since the kernel code accesses hardware and contains privileged instructions, it must be run in privileged mode. Since only the kernel runs in privileged mode, this mode is also commonly called *kernel mode* or *superuser mode*. Therefore, during a system call, the process that made the call runs in kernel mode. Unlike an ordinary function call, a system call causes a change in the execution mode of the processor; systems usually implement this with a *trap instruction*.

**NOTE** *A `trap` is a machine instruction that changes the processor mode and jumps to a specific location in memory. In older systems, the trap is implemented with the `int 0x80` instruction. Linux kernels from 2.6 and later use the `sysenter` instruction, and the GNU C library glibc 2.3.2 and later use `sysenter`.*

The kernel supports a fixed number of system calls on any given system. The `syscalls` man page lists the names of all calls supported on the system. Each call is associated with a number that's used as an index into a table of addresses to which control is transferred inside the kernel. These numbers are system-dependent, but each has a symbolic name defined by a macro. For example, the symbolic name for the `getpid()` system call number is `__NR_getpid` (as well as `SYS_getpid` for backward compatibility.) As of this writing, the latest Linux kernel has about 450 different system calls. The trap instruction is typically invoked with a parameter that specifies which system call to run.

The number of parameters in system calls varies, and the method by which they're transferred to the kernel depends on how many there are. Linux systems use a combination of two different methods:

**Register method** Parameters are placed into known registers in a specific order. When the number of parameters exceeds the number of available registers, the block method is used instead.

**Block method** The parameters are stored in a block of consecutive bytes in memory, and the address of the block is passed in a register.

Linux 5.15.0 does not allow more than six parameters to a system call.

Processes don't usually invoke system calls directly. Instead they call wrapper functions. A *wrapper function* for a function named `foo()` does very little other than repackage the parameters of the call to `foo()`, call `foo()`, collect its return value, and possibly supply it in a different form to the caller. The GNU C library *glibc* has wrapper functions for almost all system calls.

Wrapper functions for system calls usually have the same name as the call itself. They also have to execute the trap instruction to trap to kernel mode and when the system call returns, restore user mode.

A wrapper is said to be *thin* if it does almost nothing but pass the arguments in and receive the return values. The GNU C library wrapper functions are often very thin, doing little more than copying arguments to the right registers before invoking the system call and then setting the value of a global error variable.

Sometimes a wrapper is not so thin, as when the library function has to decide which of several alternative functions to invoke, depending upon what is available in the kernel. The `truncate()` system call is a good example. It can truncate a file to a specified length, discarding the data beyond that length. The original `truncate()` function could handle only lengths that could fit into a 32-bit integer. When filesystems were able to support very large files, a newer version named `truncate64()` was added. The newer function can process lengths representable in 64 bits. The wrapper for `truncate()` decides which one is provided by the kernel and calls it.

The following list summarizes the steps that take place during a system call:

1. The user program makes a system call to the wrapper function in the library.
2. The wrapper function copies the arguments of the call off of the stack and puts them into the registers where the kernel expects them.
3. The wrapper executes the trap, passing the number of the system call as its argument. This causes the mode switch to supervisor mode and the jump to the kernel's system call handler.
4. The kernel's system call handler uses the number passed to it to access the system call vector at that offset. The vector contains the address in system space of the actual kernel code for that call.
5. The actual call code is executed, and it passes the return value back to the system call handler.

6. The handler passes the return value to the wrapper. If an error occurred, the wrapper function makes the error code available to the program.

Figure 3-2 illustrates these steps schematically from the moment the system call is executed in a user program until it returns from the call.

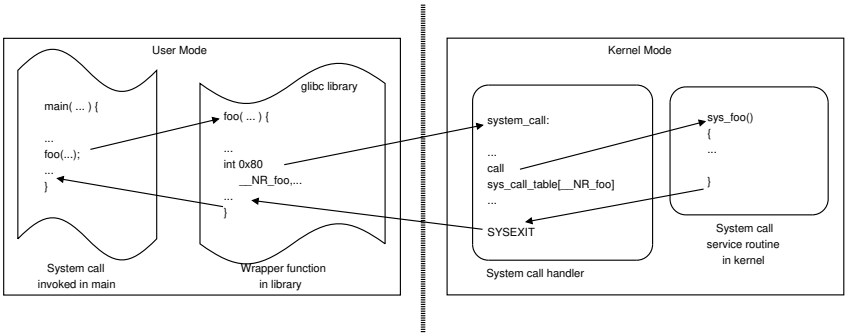



*Figure 3-2: A sample detailed system call execution flow*

Some system calls don't have wrappers in the library, and for those, the programmer has no other choice but to invoke the system call with the `syscall()` function, passing the system call's number and arguments. Generally speaking, for a system call named `foo`, its number is defined by a symbolic constant named either `__NR_foo` or `SYS_foo`. These macro definitions are exposed by the header file *sys/syscall.h*, which you'd need to include in the code. They may not be in that file itself, but in an included file, such as *<sm/unistd_32.h* or *asm/unistd_64.h*. The man page for `syscall()` lists the headers to include.

An example of a system call without a wrapper is `gettid()`, which returns the caller's thread ID. It's the same as `getpid()` for a process with a single thread. The *gettid_demo.c* program in Listing 3-1 uses `syscall()` to call `gettid()` and prints the returned ID on the screen:

*gettid_demo.c*

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

int main(int argc, char *argv[])
{
    printf("Thread id %ld\n", syscall(SYS_gettid));
    /* Could also pass __NR_gettid */
    return 0;
}
```

*Listing 3-1: A program that uses the `syscall()` function to make a system call*

Because `gettid()` has no arguments, it isn't necessary to pass anything other than the system call number to `syscall()`. If it did have arguments, they would be passed as parameters after the call's number.

In summary, some of the services that a program needs are satisfied by the following:

- Calling a library function that doesn't need to make a system call.
- Calling a library function that does make a system call.
- Making a system call through a wrapper function
- In rare cases, using `syscall()` to make a system call.

Figure 3-3 illustrates these various paths to the kernel.

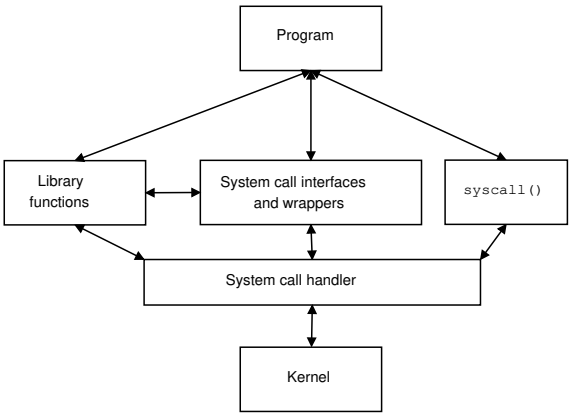



*Figure 3-3: The different control paths for obtaining services, showing the relationship between library function calls and system calls*

The figure shows, for example, that if a program calls a library function that needs to make a system call, the path to the kernel is through "library functions," "system call interfaces," and then the "system call handler."

## Handling Errors from System Calls and Library Functions

A good program should be robust enough to terminate normally even when system calls and library functions return errors, which means you need to understand how these errors are returned and know the tools at your disposal for handling them. System calls and library functions use two different methods for indicating that an error occurred.

### *Handling System Call Errors*

Almost all system calls return a negative number when an error occurs in their execution. The absolute value of that number is meaningful; it indicates the type of error. The fact that it is negative is what indicates that it failed. A handful of system calls don't behave this way, but their man pages indicate when that's the case. When the system call returns to the C library

wrapper function, if the return value is negative, the wrapper stores the absolute value of the return value into a static variable named `errno`, defined in the *errno.h* header file. It also returns `-1` to the calling program. By including *errno.h* in your program, you can read the value stored in `errno`.

A robust program should check the return value of system calls and handle every possible error. Read the man page to find the list of error values that the particular system call can return. In the `ERRORS` section on that page, you'll see the list of error codes that can be returned, expressed as symbolic constants. You don't need to know the actual numbers, just their symbolic names.

**USING THE ERRNO VARIABLE**

Your program must not declare the `errno` variable. Because `errno` is declared in *errno.h*, including the header includes its declaration. If you put another declaration of it in your program, it would hide the real `errno` variable and the one your program uses wouldn't contain the error values. Also, the program must inspect `errno` immediately after the system call because, if your program calls any other function or makes another system call before it inspects that variable, the error value may be lost. It stores only the error value of the most recent call.

You can also enter the **`errno -l`** command to see the list of all possible error codes from all system calls. This command is part of the `moreutils` package, which may not be installed on your system. If you see an error message like the following after entering that command, you need to install the package:

```
$ errno -l
EPERM 1 Operation not permitted
ENOENT 2 No such file or directory
ESRCH 3 No such process
EINTR 4 Interrupted system call
EIO 5 Input/output error
ENXIO 6 No such device or address
E2BIG 7 Argument list too long
ENOEXEC 8 Exec format error
--snip--
```

Let's work through an example to demonstrate how to put this together. We'll use a small program that makes a relatively simple system call to `gethostname()`, which stores the hostname of the computer in its first parameter or returns `-1` if it fails. Because `gethostname()` can return just a few possible error values, it's a good choice for showing how to handle errors. The first step is to read the `gethostname()` man page to understand how to call it and respond to the errors. The man page, which also documents `sethostname()`, shows us its prototype:

```
#include <unistd.h>
int gethostname(char *name, size_t len);
```

The type of the second parameter, `size_t`, is an unsigned integer type that is defined by the POSIX.1 standard. Unix systems that conform to the standard employ this type for all symbols that are supposed to store the size of any kind of object. It's our first example of a Unix system type.

The man page explains the behavior of `gethostname()`:

> `gethostname()` returns the `NULL`-terminated hostname in the character array `name`, which has a length of `len` bytes. If the `NULL`-terminated hostname is too large to fit, then the name is truncated, and no error is returned. POSIX.1-2017 states that if such truncation occurs, then it is unspecified whether the returned buffer includes a terminating null byte.

Based on this explanation, our program must check the value returned and handle the error, because if the `name` array was truncated and is missing the terminating `NULL` byte, the program will generate some type of fault, most likely a segmentation fault, when we try to print the name.

The `ERRORS` section on the man page for `gethostname()` lists three possible errors:

**EFAULT** When `name` is an invalid address

**EINVAL** When `len` is negative

**ENAMETOOLONG** When `len` is smaller than the actual size

This list implies that we should have code to handle each case. Because there are only three, a sequence of `if` statements can handle them.

Listing 3-2 contains a complete program, *gethostname_demo.c*, that demonstrates one way to handle the errors from the call to `gethostname()`:

*gethostname_demo.c*

```
#include <unistd.h>❶
#include <stdio.h>
❷ #include <errno.h>

void main()
{
    char  name[4];   /* Declare string to hold returned value */
    size_t len = 3;  /* purposely too small so error is revealed. */
    int   returnvalue;

    returnvalue =  gethostname(name, len); /* Make the call */
 ❸ if ( -1 == returnvalue ) {
        switch ( errno ) {
        case EFAULT:
            printf("A bad address was passed for the string name\n"); break;
        case EINVAL:
            printf("The length argument was negative.\n"); break;
        case ENAMETOOLONG:
```

```
            printf("The hostname is too long for the allocated array.\n");
        }
    }
    else
        printf("%s\n", name);
}
```

*Listing 3-2: A program that demonstrates how to handle system call errors by inspecting the errno variable*

The program needs to include *unistd.h* ❶ because `gethostname()` is a system call. It includes *errno.h* ❷ in order to use the `errno` variable. If the `if` condition ❸ is true, an error occurred and the switch statement selects a custom error message to print, after which the program terminates. If not, the program prints the name returned in the `else` clause. I purposely made the array too small for most machine names so that when this program is run we get to see the error message. By changing the array size to a large enough number, we prevent the error from occurring. The following run of the program shows what it outputs, assuming the hostname is *harpo* and the executable is named *gethostname_demo*:

```
$ gethostname_demo
The hostname is too long for the allocated array.
```

An alternative to writing your own messages based on the value in `errno` is to use either the `perror()` library function declared in *stdio.h* or the `strerror()` library function declared in *string.h*. Both of these functions are *locale-aware*, which means that if the program in which they're called is being run by a user who uses a language other than English, the message will be translated into that language, provided that the system supports it. Locales are described briefly in "Internationalization" on page 107 and covered in more depth in Chapter 4.

The `perror()` function writes a message onto the standard error stream describing the last error encountered during a call to a system or library function. Its synopsis is

```
#include <stdio.h>
void perror(const char *s);
```

This function prints the string argument followed by a predefined message. Usually you pass it the name of the function as the string, as shown in *perror_demo.c*, displayed in Listing 3-3. This program doesn't need a `switch` statement because the selection logic is in `perror()`. It just calls `perror()` instead.

*perror_demo.c*

```
#include <unistd.h>
#include <stdio.h>

void main()
{
    char   name[4];   /* Declare string to hold returned value. */
```

```
    size_t len = 3;   /* Purposely declared too small so error is revealed */
    int   returnvalue;

    returnvalue =  gethostname(name, len); /* Make the call. */
    if ( -1 == returnvalue )
        perror("gethostname");
    else
        printf("%s\n", name);
}
```

*Listing 3-3: A program that uses perror() to handle system call errors*

Running this program, assumed to be compiled to *perror_demo*, shows what `perror()` prints:

```
$ perror_demo
gethostname: File name too long
```

The major drawback to relying on `perror()` is that, by removing the `switch`, we've eliminated the chance to take different actions depending on the type of error. We can't, for example, terminate the program for some errors and not others. A lesser drawback is that we cannot customize the error message. Like using `errno`, your program must call `perror()` immediately after the call, because otherwise it won't have the message for the error from the call.

The `strerror()` function's synopsis is

```
#include <string.h>
char *strerror(int errnum);
```

It returns a pointer to a string containing the error message for the error number passed to it. Therefore, `strerror(errno)` is the error message associated with `errno`. We can modify the preceding example by replacing the call to `perror()` with a call to `strerror(errno)`. Here's the changed portion of the program, which I've named *strerror_demo.c*:

*strerror_demo.c*

```
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
--snip--

    returnvalue =  gethostname(name, len); /* Make the call */
    if ( -1 == returnvalue )
        printf("gethostname failed: %s\n", strerror(errno));
    else
        printf("%s\n", name);
}
```

This function is not safe to use in a multithreaded program. In Chapter 13 we introduce two thread-safe versions of this function, which are both described on the `strerror` man page.

### *Handling Errors from Library Functions*

Library functions don't necessarily respond to errors in the same way as system calls. In general they fall into four different categories with respect to error handling:

- Functions that behave exactly the same way as system calls, returning `-1` and setting the value of `errno`.
- Functions that do not return `-1` on error but do write the value of the error into `errno`, such as the C `malloc()` function, which returns a `NULL` pointer on error and sets the value of `errno` to the only possible error it can have, `ENOMEM` (out of memory).
- Functions that don't use `errno` for reporting the type of error, such as the character I/O functions `fgetc()` and `getc()`, which return `EOF` on error and don't set `errno`.
- Functions from the *Pthreads* library, which we discuss in Chapter 13, that return 0 on success and a positive number as an error value on failure.

The only way to know how to handle the errors for the specific function your program is using is to read its man page.

## Portability

*Portability* refers to the degree to which your program can run on other computers with little or no modification of the code itself. If, for example, your code uses features only available in GNU/Linux and you try to run it on another Unix system without that support, it won't behave the same way, and you may not even be able to build it unless you modify the source code. Unix's haphazard growth is partly the cause of this problem, because over time, three major variants of Unix evolved: BSD, GNU/Linux, and System V (see Chapter 1 ,"Unix History and Standards") with different features and capabilities, and people created standards to specify how those various systems were supposed to behave. One Unix system can have functions with the same names as another but whose semantics are different because they evolved in different variants. We need to know which version of a function our program uses when we compile it on the development machine, and whether it will be the same when we compile the program on a different machine.

If you're distributing source code to be built on other computers, ideally you would design it so that it will compile into an executable whose behavior is what you expect even on other computers.

Portability is tied to the concept of standards, because, for example, if your program is intended to adhere to the POSIX.1-2017 standard but must be built on a system conforming to a different, perhaps older, POSIX standard, you need to know how to design the code so that it uses features available on the other computer when the ones you hoped to use aren't available. The macro preprocessor's ability to compile code conditionally based on the values of macro objects is the key to solving this problem.

### Feature Test Macros

A *feature test macro* is a macro designed to expose *features* such as constant and function prototypes in a header file when a program is compiled. For example, the following code is found in the header *stdio.h*:

```
#ifdef __USE_GNU
  /* Close all streams.  ... */
  extern int fcloseall (void);
#endif
```

In this example, the declaration of the `closeall()` function will be exposed, meaning included in the program when you use the `#include <stdio.h>` directive, only if the symbol `__USE_GNU` is defined when the preprocessor reaches that directive in the program's code.

The header files and other source code files in the libraries and system call interfaces contain these conditional compilation directives in order to enable or disable the inclusion of various features. These feature test macros are designed to allow the libraries and interfaces to conform to multiple standards.

Feature test macros cannot be used to ensure that your program conforms to a limited standard, because they won't prevent you from including header files that haven't been written to conform to their use. They're primarily intended to control which standards to follow in the code.

Let's consider how to use them to control which features you want to enable or disable in your program. Suppose you're about to embark on a new project and want to use some functions you've never used before. Suppose one of them is the C `getline()` function, which has many versions. When you look at its man page, you'll probably see something like the following description:

```
NAME
       getline, getdelim - delimited string input

SYNOPSIS
       #include <stdio.h>
       ssize_t getline(char **lineptr, size_t *n, FILE *stream);
       ssize_t getdelim(char **lineptr, size_t *n, int delim, FILE *stream);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):
```

```
       getline(), getdelim():
           Since glibc 2.10:
               _POSIX_C_SOURCE >= 200809L
           Before glibc 2.10:
               _GNU_SOURCE
  --snip--
```

The page explicitly mentions `Feature Test Macro Requirements`. What are they, and how are you supposed to use this information?

If the `SYNOPSIS` section of a function's man page lists feature test macro requirements, it means that the given prototype or constant declaration will be read by the preprocessor only if the macro is defined *before* including *any* header files, not just the one in which it is declared, but all of them, as in

```
#define __GNU_SOURCE
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
--snip--
```

If you understand how the header files use these definitions, the code will make much more sense to you. I'll use a simplified version of the *stdio.h* header file to illustrate, because the actual header file is much more complex. The declaration of the prototype for `getline()` in this file looks roughly like this:

```
#ifdef __GNU_SOURCE
  ssize_t getline (char **__lineptr, size_t *__n, FILE *__stream);
#endif
```

Unless the symbol `__GNU_SOURCE` is defined when the preprocessor reads the "`#ifdef __GNU_SOURCE`" line, the `getline()` function will be skipped over. Therefore, in order for your program to use this version of the function, you need to define that symbol before any header file is included, like so:

```
#define __GNU_SOURCE
#include <stdio.h>
 --snip--
```

Doing this causes the lines that "`#ifdef __GNU_SOURCE ... #endif`" protects to be read.

The man page in essence tells us that if we want to use either of the two functions `getline()` or `getdelim()`, if our version of *glibc* is 2.10 or later, we need to include the definition

```
#define _POSIX_C_SOURCE 200809L    or any number >= 200809
#include <stdio.h>
```

If our version of *glibc* is older than 2.10, we need to use this macro:

```
#define _GNU_SOURCE
#include <stdio.h>
```

If you don't remember how to find which version of *glibc* you have, see "Standard C Library" on page 93.

As an alternative to defining the macro in the program source code, we can enable the definition when we compile the code on the command line using the `-D` option to `gcc`:

```
$ gcc -D__GNU_SOURCE  myprog.c  -o myprog
or
$ gcc -D_POSIX_C_SOURCE=200809L  myprog.c -o myprog
```

Some feature test macros are intended to make your program more portable by preventing nonstandard definitions from being exposed. Other macros serve the opposite purpose, exposing nonstandard definitions that aren't exposed by default. The syntax of the feature test macros on the man page uses the logical-or and logical-and operators, `||` and `&&`. The example shown in the `feature_test_macros` man page is for the `acct()` function. It's not important what this function does:

```
SYNOPSIS
    #include <unistd.h>
    int acct(const char *filename);

  Feature Test Macro Requirements for glibc (see  feature_test_macros(7)):

    acct():
        Since glibc 2.21:
            _DEFAULT_SOURCE
        In glibc 2.19 and 2.20:
            _DEFAULT_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)
        Up to and including glibc 2.19:
            _BSD_SOURCE || ❶ (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)
--snip--
```

The interpretation of the logical-or operator `||` ❶ is that in order to obtain the declaration of `acct()` from `<unistd.h>`, either of two options can be applied. One is to include the macro definition `#define _BSD_SOURCE` before including any header files. The other option contains a logical-and. You can include the macro `#define _XOPEN_SOURCE` but only if it has a numeric argument after it whose value is less than 500.

The following list describes a few common macros we'll encounter when reading code and man pages:

**`_POSIX_SOURCE`** Exposes the functionality from the POSIX.1 standard as well as all of the ISO C features.

**_POSIX_C_SOURCE** Controls which POSIX functionality is made available, determined by its assigned value.

**_XOPEN_SOURCE** Exposes features from POSIX.1, POSIX.2, and X/Open standards.

**_GNU_SOURCE** Applies only to the *glibc* library, exposes everything in ISO C89, ISO C99, POSIX.1, POSIX.2, BSD, SVID, X/Open, LFS, and all GNU extensions. (If POSIX.1 conflicts with BSD, POSIX take precedence.)

**_BSD_SOURCE** Exposes functionality derived from 4.3 BSD Unix, ISO C, POSIX.1, and POSIX.2.

**_SVID_SOURCE** Exposes functionality derived from SVID (System V Interface Definitions), ISO C, POSIX.1, POSIX.2, and X/Open.

Read the `feature_test_macros` man page to get a good understanding of why they're needed and how you can use them in general. We'll revisit them in later chapters as the need arises.

### *Other Portability Issues*

As we start to develop system programs, we'll see that other factors affect how portable they are, including

- The sizes of various data types.
- The values of configuration parameters.
- The sizes and ordering of data members in structures.
- The set of macros actually available in header files.

We'll address these issues as they arise.

## System Limits

All Unix systems set limits on system resources and properties, such as the maximum length of a filename or a pathname, and the maximum length of a username. Various standards specify minimum values for these maximums. For example, POSIX.1-2017 specifies that `_POSIX_NAME_MAX` is the least value that any conforming system can use as the maximum length of a filename. These specified values are called *system limits*.

A portable application needs to know what these limits are on each system on which it runs, and should be able to adjust its use of resources accordingly. There are a few different means for getting these limits, depending on their category. POSIX.1-2017 divides system limits into three such categories:

**Runtime invariant** Those whose values are constant for any particular Unix system.

**Pathname variable** Filesystem-related limits whose values can vary on a single system, depending on which filesystem they limit.

**Runtime increaseable** Those whose values can be increased at runtime.

For example, most runtime invariant limits are defined in the header file *limits.h*. We explore the various ways in which a program can get and set system limits in Chapter 11.

## Internationalization

In the early days of computing, almost all software was developed for English speakers. Now, computer systems are used throughout the world, and we need to design software so that it accommodates local languages and cultural conventions. Sometimes differences in cultural conventions can lead to ambiguities with serious consequences. Two simple examples illustrate this issue:

- In the United States, people express dates in the format *MM/DD/YYYY*, where *MM* is a two-digit month, *DD* is a two-digit day of the month, and *YYYY* is a four-digit year, such as 07/11/2023. In Europe, the convention is *DD/MM/YYYY*. If a program is transported from one side of the Atlantic to another, and dates are input or output, it would be hard to know which date is meant by 07/11/2023. Is it November 7th or July 11th, 2023?
- In the United States, people use commas to separate the three-digit decimal groups of large numbers, and they use the period as the *radix character*, commonly called the decimal point, when numbers are written in base 10, such as 1,048,576.00. In Europe, people use periods to separate the three-digit decimal groups of large numbers, and they use comma as the *radix character*, as in 1.048.576,00. Programs designed to parse only one representation will fail unless they know in which environment they're running.

Several other cultural conventions differ from one region to another, such as the written language, paper sizes, monetary units, time units, and measurement units.

The concept of a locale is intended to consolidate the cultural differences that affect the computational environment. POSIX.1.2017 and SuSv4 [17] simultaneously define a *locale* as "the definition of the subset of a user's environment that depends on language and cultural conventions." When a program is designed so that it works correctly no matter where it is used, and performs input and output consistent with the location in which it is run, we say the program has been *internationalized*. *Internationalization* is the process of writing programs that accommodate variations in locales across the globe. Unix systems are required to provide certain basic support for internationalizing programs. Chapter 4 contains an introduction to this complex topic.

## Processing the Command Line and Environment

In a Unix environment, commands such as

```
$ gcc -o main  main.c utils.c fileio.c
```

and

```
$ rm -r  file1 file2 file3 dir
```

are examples of simple commands, defined in Chapter 2. In a simple command, a whitespace-separated list of words may follow the name of a program. In most shells, the *whitespace* characters that separate words are either SPACE or TAB. Newline characters terminate commands and do not separate their words. A *word* is usually defined to be any sequence of non-whitespace characters not containing shell reserved characters unless they are escaped with a backslash or enclosed in single-quotes. The first word of a simple command is typically the name of a program to be executed, which might be an executable file or a shell builtin. In the preceding two examples, the programs are **gcc** and **rm**, respectively. In shells that conform to POSIX.1-2017, the command can begin with one or more variable assignments as well; we'll see examples of this in Chapter 4. In the rest of this section, when I use the term *command*, I mean a simple command.

When you type a command and press ENTER, the shell makes the words following the command name available to the program executing that command. The program needs to distinguish between the words that are non-option arguments to the command, such as `main.c`, `utils.c`, and `fileio.c` from the previous example, and those that are command options, such as `-o main` and `-r`. In this section we'll explain how you can design your programs to extract words from the command line, separate them into options and arguments, and obtain the values of any environment variables that may influence the behavior of the program.

### Extracting Command-line Arguments

To start, we'll assume that all of the words following the command name are non-option arguments. In "Option Handling" on page 117, we'll remove this assumption and revisit how to process a command line that contains options.

When you enter a command followed by its arguments, pressing ENTER to complete it, the shell program sees this command line as a list of words separated by whitespace characters. The words that follow the program name, assuming it has no options or redirection operators, are the command's *arguments*, also called the *command-line arguments*. In the first example at the start of this section, the arguments are `main.c`, `utils.c` and `fileio.c`. In the second example, they are `file1`, `file2`, `file3`, and `dir`.

In Unix and in other POSIX-compliant operating systems, the operating system in conjunction with the shell arranges for this list of words, which includes both the program name and the command-line arguments, to be

made available to the program itself as a NULL-terminated array of strings passed into the second parameter of the main() function. The shell takes care of parsing the line, finding the arguments, finding the redirection operators, and possibly evaluating variables and other expressions. For example, in the command

```
$ ls dir1 dir2 > listing  dir3
```

the arguments the shell finds are dir1, dir2, and dir3. The phrase > listing is a redirection; you're allowed to put redirections between those arguments, even though it's a confusing thing to do.

A program's main() function is allowed to have no parameters, as in

```
int main () { /* Program here ... */ }
```

but in this case it's unable to access its command-line arguments. The C standard requires compliant implementations of C (C compilers) to accept a main() function with two parameters, as follows:

```
int main ( int argc,  char * argv[]  ) { /* Program here ... */ }
```

The first parameter is an integer that specifies the number of words on the command line, which includes the name of the program, implying that argc $\geq$ 1. The second parameter is a NULL-terminated array of C strings that stores all of the words from the command line, including the name of the program, which is always in argv[0]. The command line arguments, if they exist, are stored in argv[1], . . ., up to argv[argc-1].

Although many programs use argc and argv as the names of these parameters, there's nothing special about them; they can be any valid identifiers you choose. It's a convention to use the names argc and argv, but you'll often find programs that use ac and av instead. You could name them foo and bar, for instance, but that would be pretty bad programming style. Figure 3-4 illustrates how the arguments are made available to the program.

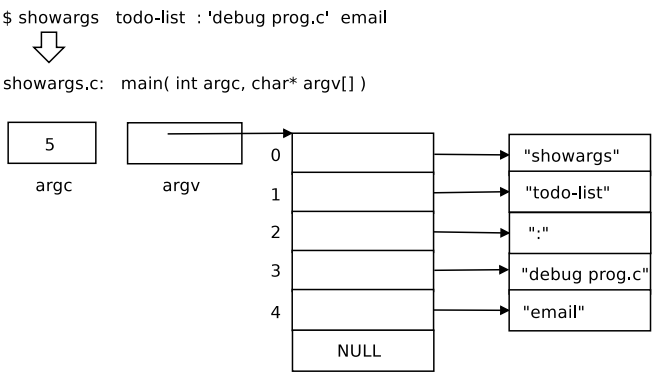



*Figure 3-4: How the command line arguments are passed to a program*

Notice that a sequence of words enclosed in single-quotes with embedded white space is a single word in the argument list. Also observe that the last element in the argument array is a NULL byte.

Listing 3-4 is a simple program, *printargs1.c*, that shows one way for a program to access its command-line arguments in a program. It displays the name that the user enters to execute the program, followed by the command line arguments that it receives from the shell, numbered to show their positions, one per line.

*printargs1.c*

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("%s arguments:\n", argv[0]);
    for ( int i = 1; i < argc ; i++ )
         printf("%d: %s\n", i, argv[i]);
    return 0;
}
```

*Listing 3-4: A program that prints its command line arguments*

Notice that the last argument is in argv[argc-1], not argv[argc]. Because the array's last element is a NULL byte, we can also iterate through the arguments until the condition argv[i]==NULL is true, as shown here:

*printargs2.c*

```
#include <stdio.h>

int main( int argc, char* argv[])
{
    int i = 1;
    printf("%s arguments: ", argv[0]);
    while ( argv[i] != NULL )
        printf("%d: %s\n", i, argv[i++]);
    return 0;
}
```

*Listing 3-5: A program that prints its command line arguments until it finds the NULL bytein the argv[] array*

Using pointer arithmetic, we could dispense completely with the index variable i. This is left as an exercise.

### Accessing the Environment

Within a program, you can also access any of the environment variables that were inherited by the program. I'll show three different ways to do this.

### Using the getenv() Function

You can use `getenv()` to retrieve the value of any environment variable in the environment passed to the program. We saw its use in our introduction to environments in Chapter 1. Its synopsis is as follows:

```
#include <stdlib.h>
char *getenv(const char *name);
```

Given `name`, it searches the environment list for a variable matching `name`, and if it finds one, it returns a pointer to its value; otherwise, it returns `NULL`. For example, the program in Listing 3-6 prints the value of the `HOME` environment variable, unless it's not in the environment, in which case it prints an error message.

*getenv_demo.c*

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
 ❶ char *path_to_home;
    path_to_home = getenv("HOME");
    if ( NULL == path_to_home )
        printf("The HOME variable is not in the environment.\n");
    else
        printf("HOME=%s\n", path_to_home);
    return 0;
}
```

*Listing 3-6: A program that demonstrates how to use `getenv()` to access the environment*

The `getenv()` function returns a pointer to the string inside the actual environment list, not to a copy of that string, which means that if you modify it in your program, you're modifying the environment. It also means that in your program, the variable that you declare to receive the return value should be a `char*` that is not assigned local storage. For example, `path_to_home` ❶ would be declared incorrectly by making it an array of characters, such as:

```
    char path_to_home[256];
```

since this allocates storage for it and makes `path_to_home` a constant `char` pointer. The function wouldn't be able to assign a value to it and the compiler will flag it as an error.

POSIX.1-2017 allows an implementation of this function to store the string whose address is returned in a statically allocated storage location, which means it will be overwritten by a subsequent call. If you intend to call it again, copy the return value to a local variable. For example, the following code may not work on some systems, because by the time that the value of `home` is evaluated, the storage has been overwritten by the return value of `getenv()` in `user = getenv("USER")`.

```
--snip--
```

```
    char *home, *user;
    home  = getenv("HOME");
    if ( NULL != home ) {
        user = getenv("USER");
        if ( NULL != user )
            printf("USER=%s and HOME=%s\n", user, home);
    }
--snip--
```

Instead, you could use a string copying function such as `strncpy()` to copy the return value into `home`, as in

```
    char home[256];
    strncpy(home, getenv("HOME"), 255);
```

If you do this, make sure to include the *string.h* header file, since the declarations of the string copying functions are there.

### Using the environ Variable

When a program starts, it's given access to an externally defined global variable named `environ` of type `char**`, which is initialized to point to the start of the environment list inherited by the program, as illustrated in Figure 3-5.

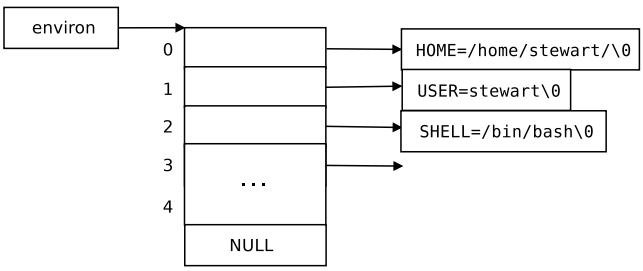



*Figure 3-5: The `environ` pointer*

In the figure, instead of enclosing the environment strings in quotes, they're shown as sequences of characters terminated by a `NULL` byte (\0). The `environ` array is terminated by a `NULL` as well.

You can use this variable to access any of the environment strings by a sequential search through the list. The program in Listing 3-7 demonstrates how to use `environ` to print the values of all environment variables inherited by the program.

*environ_demo.c*

```
#include <stdlib.h>
#include <stdio.h>

extern char **environ; /* environ is declared extern because it */
                      /* is defined outside of the program.    */
int main()
{
```

```
    char **envp = environ; /* Set pointer to start of list */
    while ( NULL != *envp ) {
        printf("%s\n", *envp );
        envp++;
    }
    return 0;
}
```

*Listing 3-7: A program that uses char **environ to search the environment sequentially and print all environment strings*

In effect, *environ_demo.c* does exactly what the `printenv` command does.

If all you need are the values of a few environment variables, using `environ` is not the best way to obtain them because you'd need to search the environment linearly, and in the worst case, doing so would take time proportional to the size of the environment, in bytes, because each string comparison looks at every character of every variable name in the worst case. The `getenv()` function does this searching efficiently, so it's a better choice.

#### Using a Third Parameter to main()

The third method for accessing the environment list is to declare the `main()` function of any program with a prototype that has a third parameter:

```
int main( int argc, char* argv[], char* envp[])
```

This `envp` parameter points to the start of the environment list inherited by the program, in the same way that the `environ` variable does. You could then access the environment list with a loop such as:

```
    int n = 0;
    while (NULL != envp[n]  ) {
        /* Do something with envp[n] */
        printf("%s\n",envp[n++]);
    }
```

If you need only a few variables' values, it's better to use `getenv()`. Even though many systems support this feature, POSIX.1-2017 doesn't support it, which implies that on some systems, your code won't work if you use it, so I advise you not to use it.

### Reporting Usage Errors

A program that expects one or more command line arguments must check whether it was provided what it expected. Otherwise, it will attempt to access locations in the array of arguments that don't exist, resulting in a fatal error.

For example, suppose you write a program that expects the names of two files on the command line, the first being the name of a file to open for reading and the second being the name of a file to open for writing. Let's

say the program executable is named `myprog`. The correct usage of `myprog` would be of the form:

```
$ myprog inputfile outputfile
```

The command line must have at least three words for this program to run properly. If there are more than three, it can ignore the extras. The program should be allowed to run only if the first parameter to `main()` (`int argc`) is at least 3. The program in Listing 3-8 demonstrates how to check for correct usage properly:

*usagecheck_demo.c*

```
#include <stdio.h>
#include <stdlib.h>

int main ( int argc,  char * argv[]  )
{
    if ( argc < 3 ) { /* Too few arguments */
        /* Handle the incorrect usage here */
     ❶ fprintf(stderr, "usage: %s file1 file2\n", argv[0]);
        exit(1);
    }
    printf("About to copy from %s to %s\n", argv[1], argv[2]);
    /* But no code for copying just yet */
    return 0;
}
```

*Listing 3-8: A program that shows how to check for correct usage, printing a message if it is used incorrectly.*

If the user doesn't supply two or more arguments, the program exits after displaying a message, by calling the C `fprintf()` function❶, whose first parameter is the C Library file stream to which to print, in this case, the standard error stream (`stderr`.) Otherwise, it prints a message saying that it will copy from the first named file to the second. We'll see how to copy files in Chapter 5; for now, we just say we're doing so.

### Extracting the Program Name

Suppose that we compile *usagecheck_demo.c*, putting the executable into a different directory from our working directory. This command puts it into the *bin* subdirectory of our home directory:

```
$ gcc -o ~/bin/usagecheck_demo usagecheck_demo.c
```

The ~ character is a shell special character that expands to the pathname of a user's home directory. We'll now run `usagecheck_demo` from the current working directory without giving it the name of the output file:

```
$ ~/bin/usagecheck_demo infile
usage: /home/stewart/bin/usagecheck_demo file1 file2
```

When the program runs, the tilde ~ is expanded to the path */home/stewart* and `argv[0]` contains the entire pathname, */home/stewart/bin/usagecheck_demo*.

If you don't want to display the entire path name of the program, but prefer that it displays only the more concise message

```
usage:  usagecheck_demo file1 file2
```

regardless of where the executable is, then, before you print it, you have to strip off the leading part of the `argv[0]` string so that the only thing left is what comes after the final / character. There are two relatively portable ways to do this, one more general than the other.

One way is to use the `strrchr()` function declared in *string.h*, whose prototype is

```
char *strrchr(const char *source, int ch);
```

This function returns a pointer to the rightmost occurrence in `source` of the `ch` character. (In C, we can declare characters as `int`.) If `ch` isn't found in `source`, it returns a `NULL` pointer. An algorithm for displaying the characters of the program name after the final / can search for the rightmost slash in the pathname and, if found, display the string that follows it. If it isn't found, no leading directories exist in the path, so it can print the entire path. If it is found and is the rightmost character, it's a usage error, since it means the pathname ends in a slash. Trying to run a command whose name ends in a slash causes most shells to report an error. Listing 3-9 demonstrates this method:

*progname_demo.c*

```
#include <stdio.h>
#include <string.h>

int main(int argc, char * argv[])
{
    char *forwardslashptr;
    char *suffixptr = NULL;

    forwardslashptr = strrchr( argv[0], '/');
    if ( forwardslashptr != NULL )
     ❶  suffixptr = forwardslashptr+1;
    else
        suffixptr = argv[0];
    if ( suffixptr == NULL )
        fprintf(stderr, "Program name ends in a / character\n");
    else
        printf("Program name is %s\n", suffixptr);
    return 0;
}
```

*Listing 3-9: A program that strips the program name of any leading directories using `strrchr()`*

For those unfamiliar with C, the instruction `suffixptr = forwardslashptr+1;` ❶ performs *pointer arithmetic* to make `suffixptr` point to the first character after the forward slash. When pointer arithmetic appears in code, the compiler translates addition of an integer *n* to a pointer of type `basetype*` into the addition of `sizeof(basetype)*`*n* bytes to the pointer's value. It's worth remembering the `strrchr()` function, because it's a useful function for other purposes as well. For example, we can use it to get the suffix of a filename, or to get the portion of the filename before the suffix.

An easier, but less general, method of stripping the directories from the pathname in `argv[0]` is to use the `basename()` library function, of which there are both POSIX and GNU versions. Their prototypes are the same:

```
char  *basename(char *path);
```

but the POSIX function is declared in *libgen.h*, whereas the GNU version is declared in *string.h*. The POSIX function modifies `argv`, but the GNU version does not. Furthermore, the man page for `basename()` states that the POSIX version implemented in *glibc* has bugs. For all of those reasons, we'll use the GNU version to demonstrate. To use the GNU version, we need to define the `_GNU_SOURCE` macro before including any header files. Listing 3-10 shows the program:

*basename_demo.c*

```
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>

int main(int argc, char * argv[])
{
    char *progname;
    progname = basename(argv[0]);
    printf("Program name is %s\n", progname);
    return 0;
}
```

*Listing 3-10: A program that strips the program pathname of its leading directories using the `basename()` library function*

We'll compile this into an executable in the *~/bin* directory as we did with *usagecheck_demo.c* and run it in the same way:

```
$ gcc -o ~/bin/basename_demo  basename_demo.c
$ ~/bin/basename_demo
Program name is basename_demo
```

Only the program name is printed, not the full pathname.

### Extracting Command Line Options

Almost all commands have options, which might be short or long or both. Some shells allow the order of options and arguments to vary. For example, the following two command lines are equivalent:

```
$ gcc myprog.c  -o myprog -Wall -I includedir
$ gcc -Wall -o myprog -I includedir  myprog.c
```

POSIX.1-2017 requires that all options should precede all of the arguments, but some commands don't conform to this requirement. If a command has several short options, such as `-a`, `-b`, `-c`, and `-d`, none of which have arguments, we can write them in various combinations, such as

```
$ ssh -aCfGgKkMN
$ ssh -a -c  -CfGg -Kk -M -N
$ ssh -CfGg -Kk -M -a -c  -N
```

If options do have arguments, their arguments must follow them immediately, with whitespace allowed in between the option letter and its argument.

The Utility Syntax Guidelines of the POSIX.1.2017 standard (Section 12.2), contain rules about options that programs should follow to conform to the standard. In particular, a program that conforms to these requirements should support the following option syntax:

- One or more short options that have no option arguments, followed by at most one option that has an option argument, can grouped behind one hyphen (-) delimiter.
- The order of different options relative to one another should not affect program behavior, with one exception. Repeated options that have required arguments must be interpreted in the order that they appear. The `make` utility is an example of a command that allows this. It can have multiple `-f` options, and their order does matter.

GNU allows long options, but POSIX.1-2017 doesn't require them. Also, GNU allows options to have optional arguments, which POSIX.1-2017 forbids. Finally, GNU allows arguments to precede options, which POSIX.1-2017 forbids.

Writing a program that allows the user to enter options in various forms, and in any order, consistent with these requirements makes parsing the command line to find all of the options and their arguments a complex task. Fortunately, Unix systems usually have two library functions named `getopt()` and `getopt_long()` that can do this work for you. The latter is a GNU function that can parse command lines with long options, for which you need to define `_GNU_SOURCE` before the header file inclusions. Their combined man page is as follows:

```
GETOPT(3)                  Linux Programmer's Manual                 GETOPT(3)

NAME
```

```
       getopt, getopt_long, getopt_long_only, optarg, optind, opterr, optopt -
       Parse command-line options

SYNOPSIS
       #include <unistd.h>

       int getopt(int argc, char * const argv[],
                  const char *optstring);

       extern char *optarg;
       extern int optind, opterr, optopt;

       #include <getopt.h>

       int getopt_long(int argc, char * const argv[],
                  const char *optstring,
                  const struct option *longopts, int *longindex);
       int getopt_long_only(int argc, char * const argv[],
                  const char *optstring,
                  const struct option *longopts, int *longindex);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):
       getopt(): _POSIX_C_SOURCE >= 2 || _XOPEN_SOURCE
       getopt_long(), getopt_long_only(): _GNU_SOURCE

--snip--
```

Even though these are library functions, to use them you must include *unistd.h*. The variables `optarg`, `optind`, `opterr`, and `optopt` are externally defined, and you must not declare them in your program.

The man page explains everything we need to know to use these functions. If our program expects all arguments to follow all options, before the header files are included, it should define `_POSIX_C_SOURCE` with a value greater than or equal to 2, or define `_XOPEN_SOURCE`. If we want to allow a user to intermingle options and arguments, it doesn't need to define either of these macros.

As mentioned previously, we must define `_GNU_SOURCE` to use `getopt_long()`.

The `getopt()` function parses the command line arguments. Its first two arguments, `argc` and `argv`, are the argument count and array passed to the `main()` function. The third argument, `optstring`, is a string that identifies the options and their arguments. The string is interpreted according to the following rules:

1. A letter by itself is an option without arguments. For example, `b` represents `-b`.
2. A letter with a single colon `:` after it has a *required* argument, and `getopt()` will place a pointer to the argument in `optarg` if it exists, and

if it's missing, it will return `?`. (See Rule 5 about how a leading ':' in `optstring` is used).

3. A letter with a double colon after it `::` has an *optional* argument, and `getopt()` will place a pointer to it in `optarg` or will set `optarg` to 0 if it's missing.
4. If `getopt()` finds an undefined option, it will put the character in `optopt` , print an error message on `stderr`, and return `?`. You can set `opterr` to 0 to suppress the message. It will also perform these actions if a required option argument is missing.
5. If the leading character is a colon (`:`) then if `getopt()` finds a missing required option argument, instead of returning a `?`, it returns a colon (`:`), which makes it possible to distinguish the type of error. A `:` implies a missing option argument, and a `?` implies an invalid option character.

Let's look at a small program that uses `getopt()`. The option string `":hb::c:1"` specifies that `-h` and `-1` are options without arguments, `-b` is an option with an optional argument, and `-c` is an option with a required argument.

The `getopt()` function initializes the external variable `optind` to 1. When `getopt()` is called repeatedly, it returns each of the option characters from each of the option elements on the command line. When it can't find any more options, it sets `optind` to be the index in the `argv` array of the next element to be processed, and it returns -1. Thus, when it returns -1, `optind` is the index in `argv` of the first `argv` element that isn't an option. (By default, the GNU version of `getopt()` rearranges the contents of `argv` as it scans, so that eventually all the non-options are at the end.)

A program that uses `getopt()` to parse the command line should consist of two parts:

1. A loop that calls `getopt()` repeatedly to find and record all options, option arguments, as well as any errors in usage, after which it stores the command arguments in the `argv` array into suitable variables.
2. Conditional code that uses the presence or absence of options and arguments found earlier to control the program's execution.

Listing 3-11 demonstrates this idea. It uses the same set of options as in the example we just described. This program doesn't do anything other than printing a list of the options that it finds as well as its arguments, but it shows how to collect the options found into a set of variables to be used later by the program.

*getopt_demo.c*

```
#include <stdio.h>     /* For printf()          */
#include <stdlib.h>    /* For exit()            */
#include <unistd.h>    /* For getopt()          */
#include <string.h>

#define  TRUE  1
#define  FALSE 0
```

```
int main( int argc, char* argv[])
{
    int ch;
    char options[] = ":hb::c:1";
    int opt_h = 0;
    int opt_1 = 0;
    int opt_b  = 0;
    int opt_c  = 0;
    char b_arg[32] = "";
    char c_arg[32] = "";

    opterr = 0;  /* Turn off error messages by getopt(). */
    while  (TRUE) {
        /* Call getopt, passing argc and argv and the options string. */
        ch = getopt(argc, argv, options);
        /* It returns -1 when it finds no more options  */
        if ( -1 == ch )
            break;
        switch ( ch ) {
        case 'h':                       /* h is a switch (no arg). */
            opt_h = TRUE; break;
        case 'b':                       /* b has an optional argument. */
            opt_b = TRUE;
            if ( 0 != optarg )
                strcpy(b_arg, optarg); break;
        case 'c':                       /* c has a required argument. */
            opt_c = TRUE;
            strcpy(c_arg, optarg); break;
        case '1':                       /* 1 is a switch (no arg). */
            opt_1 = TRUE; break;
        case '?' :
            printf("Found invalid option %c\n", optopt); break;
        case ':' :
            printf("Missing required argument\n"); break;
        default:
           printf ("?? getopt returned character code 0%o ??\n", ch);
        }
    }
    /* Finished processing the command line */
    /* Process the options - in this case just print what was found. */
    printf("Options found:\n");
    if ( opt_h ) printf("-h \n");
    if ( opt_1 ) printf("-1 \n");
    if ( opt_b ) {
         printf("-b ");
         if ( strlen(b_arg) > 0 )
```

```
            printf("with argument %s\n", b_arg);
        else
            printf("with no argument \n");
    }
    if ( opt_c )
        printf("-c with argument %s\n", c_arg);

    /* optind  is the index of the first non-option word in the argv[] array. */
    /* If optind < argc, there is at least one word that is not an option. */

    if (optind < argc) {
        printf ("non-option ARGV-elements:\n");
        while (optind < argc)
            printf ("%s ", argv[optind++]);
        printf ("\n");
    }
    return 0;
}
```

*Listing 3-11: A program that parses the command line for options and arguments*

Listing 3-11 models the usual way to process the options, using a loop and an embedded `switch` statement in which the fact of finding an option is recorded in a variable associated with that option. This variable is checked later in the program. For example, if the `-h` option is a flag to indicate whether to print a help message, the `switch` code fragment would look like this:

```
    switch ( ch ) {
    --snip--
    case 'h':
        print_help = TRUE;
        break;
```

Then somewhere in the main program's body, we'd put code such as the following:

```
    if ( print_help )
        print_help_message(); print the help information
```

If the program allows the same option to be present multiple times on the command line with different arguments, the `switch case` for that option needs to store the successive arguments in a suitable data structure.

## Extracting Numbers from Strings

Command-line arguments and environment values are stored as strings, even if they represent numbers. When a program receives strings that are actually numeric, it needs to parse those strings to obtain the numbers they represent. Fortunately, various library functions can do this, although some are preferable to others.

Two different classes of functions convert strings to numbers: `atoi()` (and its cousins `atof()` and so on) as well as `strtol()` and its cousins. Table 3-1 below summarizes the functions available in a typical Linux system.

**Table 3-1:** String to Number Conversion Functions

| Function | Result type | Remarks |
|---|---|---|
| `atoi(const char *nptr)` | `int` | no error-checking |
| `atol(const char *nptr)` | `long int` | no error-checking |
| `atoll(const char *nptr)` | `long long int` | no error-checking |
| `atof(const char *nptr)` | `double` | no error-checking |
| `strtol(const char *nptr, char **endptr, int base)` | `long int` | sets errno on error |
| `strtoll(const char *nptr, char **endptr, int base)` | `long long int` | sets errno on error |
| `strtof(const char *nptr, char **endptr)` | `float` | sets errno on error |
| `strtod(const char *nptr, char **endptr)` | `double` | sets errno on error |
| `strtold(const char *nptr, char **endptr)` | `long double` | sets errno on error |
| `strtoul(const char *nptr, char **endptr, int base)` | `unsigned long int` | sets errno on error |
| `strtoull(const char *nptr, char **endptr, int base)` | `unsigned long long int` | sets errno on error |

The functions whose names are of the form `ato*` have some disadvantages. One is that they don't set the `errno` variable if errors occur, returning 0 instead, which makes it hard to distinguish between a numeric 0 and an error. Second, they don't do much error checking. Finally, they can be used only for base 10 numerals, which is not a major limitation, since that's the base we use most often. If you look at their man pages, you'll see that their use is discouraged. In spite of this, I'll occasionally code with `atoi()` when I'm writing software for my own use that will only be used a few times and I don't need to error-check, what people commonly call *throw-away code*.

It's a better idea to learn how to use `strtol()` and the related functions because they're more general and provide robust error-checking. I'll describe how to use `strtol()`; learning how to use the related functions that return numbers of types `unsigned long`, `long long`, `unsigned long long`, and so on, is similar.

The synopsis of the `strtol()` function is as follows:

```
#include <stdlib.h>
long strtol(const char *nptr, char **endptr, int base);
```

The first parameter (`nptr`) is a pointer to the string to be converted. If the second parameter (`endptr`) is not `NULL`, then after the call, `strtol()` will store the address of the first invalid character it finds into `*endptr`. The last parameter (`base`) is the base of the numeral, which can be any base from 2 to 36. If you expect base 10 numerals, set the `base` to 0. Listing 3-12 shows a simple program, *strtol_demo.c*, that converts base 10 numerals:

*strtol_demo.c*

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

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

    if (argc < 2) {
        fprintf(stderr, "Usage: %s str \n", argv[0]);
        exit(EXIT_FAILURE);
    }
 ❶ errno = 0;    /* To distinguish success/failure after call */
    val = strtol(argv[1], ❷ &endptr, 0);

    /* Check for various possible errors. */
 ❸ if (errno != 0) {
        perror("strtol");
        exit(EXIT_FAILURE);
    }
    /* errno == 0 */
 ❹ if (endptr == argv[1]) {
        /* The first invalid char is the first char of the string. */
        fprintf(stderr, "No digits were found\n");
        exit(EXIT_FAILURE);
    }
 ❺ if (*endptr != '\0')
      /* There are non-number characters following the number,
         which we can call an error or not, depending. */
       printf("Characters following the number: \"%s\"\n", endptr);

    /* If we reached here, strtol() successfully parsed a number. */
    printf("strtol() returned %ld\n", val);
    exit(EXIT_SUCCESS);
}
```

*Listing 3-12: A program that calls `strtol()` to convert its first argument to a number*

Let's study some of the details in Listing 3-12. We set `errno` to 0 ❶ so that after the call, if it's non-zero we'll know that an error occurred. We need to do this because the actual number might be 0, implying that we can't interpret a return value of 0 as an error.

We pass the address of `endptr`, not `endptr` itself ❷ to `strtol()`. After the call, `endptr` contains the address of the first invalid character. We also check when `strtol()` returns, whether `errno` is 0 ❸. If it isn't 0, a conversion error occurred and in this case we exit the program because the number might be out of range and we don't want to attempt to store it.

If `errno` is 0 ❹, there was no error, but it's possible that the string was not a number. If `endptr` points to the start of the string, it wasn't a number.

Finally, we check for a different possibility ❺. It's possible that the string is something like "1234abc", which has valid digits followed by non-digits. If `endptr` doesn't point to the end of the string, the string must have non-digits. It's best in this case to let the calling program know this.

If we run this program with several different types of input, we'll see how it behaves. Assume the executable is named `strtol_demo`.

```
$ strtol_demo  100000000000000
strtol() returned 100000000000000
$ strtol_demo -817238172
strtol() returned -817238172
$ strtol_demo +871237abns
Characters following the number: "abns"
strtol() returned 871237
$ strtol_demo kjasdksd
No digits were found
$ strtol_demo 712381723812736873236817236
strtol: Numerical result out of range
$ strtol_demo 032
strtol() returned 26
```

The very last run is revealing: the leading 0 is interpreted by `strtol()` as an indicator that the number is octal.

Because we'll need to convert strings to numbers frequently, in Chapter 4, we'll develop a few functions based on the `strto*` functions that we'll use in subsequent chapters of the book.

Before leaving this topic, however, let's consider another very simple way to extract the numeric value of a string using the `sscanf()` function. It's essentially the same as `scanf()` except it reads from a C string passed to it in its first parameter instead of from the standard input stream. Its synopsis is

```
#include <stdio.h>
int sscanf(const char *str, const char *format, ...);
```

Like `scanf()`, its return value is the number of items successfully read and converted to the format specified. By giving it the `%d` format specifier and passing the address of an integer as the second argument, we can obtain the integer value of the string. Listing 3-13 shows how to do this.

*str2int.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( int argc, char* argv[])
{
    int x;
```

```
    if ( argc < 2 ) {
        fprintf(stderr,"usage: %s <number>\n", argv[0]);
        exit(1);
    }
    sscanf(argv[1], " %d", &x);
    printf("The number is %d\n", x);
    return 0;
}
```

*Listing 3-13: A program that uses sscanf() to convert strings to numbers*

This program calls `sscanf()` just once. I wouldn't recommend using `scanf()` when your program needs to convert thousands of strings to numbers, because it is slower than the other methods I described . Also, like `atoi()`, it doesn't handle errors as robustly as `strtol()`.

## Summary

System programs make requests to the kernel for services that require kernel-level privileges through the use of system calls. System calls are function calls to functions implemented within the kernel.

A library is a file that bundles together the compiled object code from multiple functions so that they can be called from other programs. Some libraries are static and are linked to a program as the last stage of compilation, and others are dynamic, or shared, and are linked during execution of the program. Using dynamic libraries makes executable files smaller and loading faster, saves memory, and makes recompiling programs unnecessary if the libraries are updated without changes to their interfaces. Several command line tools can inspect libraries and executable files.

The C Standard Library contains a wide range of functions. Some of its functions make system calls, and others work only in user space. Both library functions and system calls have very specific ways of returning error information. Any program that you write must handle errors from these functions.

Because different Unix systems follow standards to varying degrees, making programs portable can be challenging. Using feature test macros is a well-supported method to improve the portability of your programs.

Internationalization is an aspect of programming that is not covered in a typical programming curriculum, but it's important in today's world because programs might run in a variety of different cultural environments. Modern programs should be designed to respond to its user's locale settings. In the next chapter, we'll see how to do this.

This chapter showed how to process the command line, extracting the arguments and options to commands, how to access environment variables, and how to parse strings to extract their numeric value when they represent numbers. It's laid the foundation for the rest of the book. Starting in the next chapter, we'll apply much of what we've just covered.

## Exercises

1. This exercise is open-ended. Navigate to the */usr/bin* directory on the host you're using. There, run the `ldd` command on every executable, and examine the sets of dynamic libraries to which each executable is linked. Which libraries are used the most? Which commands link to the most libraries?
2. The *printargs2.c* program in Listing 3-5 used an integer to iterate through the `argv[]` array. Write a version of it that does not print the argument numbers and does not use any local variables.
3. Write a program that prints out the words it receives on the command line in reverse order, one per line.
4. Write a program that prints out the words it receives on the command line sorted by their lengths, from shortest to longest, one per line. Words of the same length can be in any order.
5. The program *perror_demo.c* in Listing 3-3 purposely used an array of characters too small for the hostname. Read the man page that describes the *limits.h* header file, find the system constant that specifies the maximum hostname length, and rewrite the program so that this error cannot occur.
6. The `seq` command prints out sequences of numbers. In the simplest case, **`seq`** ***`num1 num2`*** prints every number from *`num1`* through *`num2`*. Write a program that implements this simple form of the command. If any arguments are missing, if they are not two integers such that the first is less than or equal to the second, it should print an error message.

# 4

# GETTING STARTED: TIME AND LOCALES

In this chapter I introduce the paradigm we'll employ to design and implement system programs in the rest of this book, and I'll present the common code used by the projects that we'll develop and explain its organization. We'll also explore how Unix represents dates, times, and other information that depends on regional and cultural norms, such as character sets, monetary units, and numbers. Finally, I'll explain the concept and implementation of locales and describe the steps needed to internationalize programs so that their interfaces conform to the settings users choose for their locales.

## Learning System Programming

Trying to learn all of the intricacies and details of the kernel API by reading through reference manuals and other documentation is not just a painstak-

ing task, but an ineffective means for learning how to write system programs. There are too many system calls and library functions to remember. My experience teaching computer science for roughly 40 years is that people often learn well by following examples and then solving programming problems related to the examples, using them as starting points for their code, and this is the paradigm I use here. Rather than concentrating exclusively on the manuals and knowledge bases, you'll learn the API little by little by writing programs that use it, exploring the documentation as needed to understand how to use the relevant parts of the API. You'll start with simple programs and over time increase the complexity of the projects.

This method wouldn't be possible if we were using a different operating system. Linux, like several other Unix distributions, is an open source operating system. Not only can we see its source code, but because of its licensing, we also can share it, redistribute it, and even modify it. For us, this means we're not infringing on any copyright when we share those sources here or elsewhere.

This strategy is not my own invention. In his book *Understanding Unix/Linux Programming* [16], Bruce Molay used a similar strategy for teaching system programming. Here, we'll use the following procedure:

1. Choose an existing command or program that interacts with the kernel, such as a shell utility.
2. Read the man page for that command to make sure we understand what the command does and what system resources it uses.
3. Using the man pages and other online information, investigate the system calls and kernel data structures that we discovered it uses.
4. Write a similar version of that command, not one that is identical, but one that adds, modifies, or removes features and that uses those same system resources.
5. After finishing the given exercise, evaluate how well we did, identifying areas in which we could improve the program.

By repeating this procedure over and over, we'll gradually familiarize ourselves with the relevant portions of the API as well as the resources needed to learn about it. Finding information in the man pages will become easier and easier the more we do this, and with enough practice, we'll be more comfortable in tackling a more difficult and complex command. When we first try to read the actual sources, we won't understand much, but over time it will get easier and easier.

The content of the man pages may not be identical from one system to another, since it depends on factors such as which version of Unix you're using, which updates have been applied, and what third-party software is installed. The man pages that I present in this book are the latest versions at the time of this writing. Pages in Sections 2 through 7 are from the Linux Man-Pages Project, *https://www.kernel.org/doc/man-pages/*. Pages in Sections 1 and 8 of the man pages are mostly from the GNU Project *https://www.gnu.org/*. The man pages that you see in this book may be slightly different from

the ones you see on your system. When following along, compare the pages to which you have access to these and make sure the programs you write are based on the man pages for your specific system.

## Organization of Common Code

As we develop the programs in this book, we'll discover that certain functions are common to multiple projects. For example, in many projects, we'll need functions to extract the numeric values of command-line arguments, as well as functions to handle errors of various kinds. Rather than implement these functions in every project independently, we'll put their definitions and prototype header files into a common directory and include the header files as needed. We'll also create a static library containing all of the object modules for these functions and link that library to every program that needs them. The same holds true for definitions of various types and macro constants that we might need. We'll place macros that define the maximum sizes of strings and other programming elements into the header file, *common_hdrs.h*.

For the programs that we'll create in this book, header files fall into one of two separate categories:

**System-wide headers** Those provided by the operating system distribution

**Local headers** Those containing declarations specific to the book's projects.

The source code distribution that accompanies the book, at *https://github.com/stewartweiss/intro-linux-sys-prog*, has two top-level directories, named *include* and *lib*. The *include* directory contains header files and *lib* contains libraries and object modules created for the book. All system-wide header files that may be needed in more than one project are included in the file *include/sys_hdrs.h*. The file *include/common_hdrs.h* includes *sys_hdrs.h* as well as local headers and therefore, by including *common_hdrs.h* in a project, we include the system header files as well as the ones we've created ourselves.

A fragment of *sys_hdrs.h* follows. The source distribution for the book contains the complete listing.

```
#include <sys/types.h>   /* Type definitions used by many programs  v */
#include <stdlib.h>      /* Prototypes of many C functions and macros */
#include <stdio.h>       /* C standard I/O library                    */
#include <string.h>      /* String functions                          */
#include <limits.h>      /* System limit constants                    */
#include <unistd.h>      /* Prototypes of most system calls           */
#include <errno.h>       /* errno and error constants and functions   */
```

Listing 4-1 contains the complete *common_hdrs.h* header.

*common_hdrs.h*

```
#ifndef COMMON_HDRS_H
#define COMMON_HDRS_H
```

```
#include "sys_hdrs.h"
/* Non-system headers */
#include "get_nums.h"    /* string to number conversions             */
#include "error_exits.h" /* error-handling and exit functions        */

/* Define various constants and types used throughout the examples.   */
#define  STRING_MAX  1024

/* Create a BOOL type */
#ifdef FALSE
    #undef FALSE
#endif
#ifdef TRUE
    #undef TRUE
#endif

#ifdef BOOL
    #undef BOOL
#endif
typedef enum{FALSE, TRUE} BOOL;

#define MAXLEN  STRING_MAX        /* Maximum size of message string        */

/* Definitions  used by locale-related programs */
#define FORMAT  "%c"              /* Default format string                 */
#define BAD_FORMAT_ERROR    -1    /* Error in format string                */
#define TIME_ADJUST_ERROR   -2    /* Error to return if parsing problem    */
#define LOCALE_ERROR        -3    /* Non-specific error from setlocale()   */

/* General errors */
#define READ_ERROR          -4    /* Incomplete read of a file            */
#define MEM_ERROR           -5    /* Insufficient memory                  */
#endif /* COMMON_HDRS_H */
```

*Listing 4-1: The* common_hdrs.h *include file*

The following lines are called a *header guard*:

```
#ifndef COMMON_HDRS_H
#define COMMON_HDRS_H
--snip--
#endif /* COMMON_HDRS_H */
```

See the Header Guards box if these are unfamiliar to you. Every header file should have a header guard to prevent multiple definition errors.

**HEADER GUARDS**

Suppose that a file named *func.c* contains an include directive `#include "common.h"`. When the macro preprocessor, `cpp`, sees this directive, it copies the named file *common.h* into a copy of *func.c* at the point at which the `#include` directive was found. Every included file is copied into this temporary copy of the file that `cpp` is processing.

Suppose that a second header file, *mylist.h*, which contains the prototypes for functions in *mylist.c*, uses some functions declared in *common.h* (as well as other functions), and it therefore includes *common.h*. Finally, suppose that the main program, *main.c*, uses functions declared in both *common.h* and *mylist.h*. Then *main.c* will contain these directives:

```
#include "common.h"
#include "mylist.h"
```

When you run the compiler to build the executable for *main.c*, `cpp` sees the `#include` directive to copy *common.h* and will copy it into its temporary copy of *main.c*. It then sees the `#include "mylist.h"` directive and copies the file *mylist.h* after it. But this file also includes *common.h*, so any definitions in *common.h* will now appear twice in the copy of *main.c* that `cpp` passes to the compiler, which will cause the compiler to report definition errors. A *header guard*, also called an *include guard*, is a conditional macro-based construction designed to prevent this.

By enclosing a header file, say named *file.h* in a conditional macro of the following form we prevent the file from being included twice:

```
#ifndef FILE_H
#define FILE_H
--snip--
#endif
```

This is because the first line `#ifndef FILE_H` has the meaning, "if the macro symbol `FILE_H` is not defined, continue reading and processing code until the matching occurrence of `#endif`. In this case, the line immediately after this conditional test, `#define FILE_H`, causes `cpp` to store the definition of `FILE_H`. On the other hand, if when `#ifndef FILE_H` is executed, the symbol `FILE_H` is defined, then `cpp` skips reading code until immediately after the matching `#endif`. This implies that any code enclosed in the header guard will be included only once, and the multiple definitions cannot occur.

Notice that the *common_hdrs.h* file in Listing 4-1 includes a header named *get_nums.h*, as well as *error_exits.h*. In the following section, "Functions for Extracting Numbers," we discuss the first of these and in "Common Error Handling Functions" on page 136 we discuss the second.

### Functions for Extracting Numbers

The *get_nums.h* header declares prototypes and constants for functions that extract the numeric values of string data. (In Chapter 3 we saw how to use

strtol() for this purpose.) The header contains prototypes for two functions based on the use of strtol() as well as associated constants that those functions use. We create these functions because the error handling that should be done after calling the library functions can be lengthy. Rather than putting that error-handling code into every program that calls the function, we can integrate it into separate functions, like wrapper functions, making the calling programs smaller.

We name the two functions in that header get_long() and get_int(). The part of the header file containing the get_long() prototype and the defined constants is shown in Listing 4-2.

```
#ifndef GET_NUMS_H
#define GET_NUMS_H

#include "sys_hdrs.h"

❶ /* Flags to pass to functions */
#define NO_TRAILING           1     /* Forbid trailing characters     */
#define NON_NEG_ONLY          2     /* Forbid negative numbers        */
#define ONLY_DIGITS           4     /* Forbid strings with no digits  */

❷ /* Return codes */
#define VALID_NUMBER          0     /* Successful processing                 */
#define FATAL_ERROR          -1     /* ERANGE or EINVAL returned by strtol() */
#define TRAILING_CHARS_FOUND -2     /* Characters found after number         */
#define OUT_OF_RANGE         -3     /* int requested but out of int range    */
#define NO_DIGITS_FOUND      -4     /* No digits in string                   */
#define NEG_NUM_FOUND        -5     /* Negative number found but not allowed */

/** get_long()
    On successful processing, it returns VALID_NUMBER and stores the resulting
    number in *value, otherwise it returns one of the non-zero error codes
    and puts a suitable message into msg. flags is used to decide whether
    trailing characters, negative values, and zeros for strings without any
    digits are allowed or should be errors.
 *  @param  char*     arg     [IN]  String to parse
 *  @param  int       flags   [IN]  Flag specifying how to handle anomalies
 *  @param  long*     value   [OUT] Returned long int
 *  @param  char*     msg     [OUT] If not empty, error message
 *  @return int       VALID_NUMBER or a negative error code indicating the
                      type of error
 */
int  get_long(char *arg, int flags, long *value, char *msg );
--snip--
#endif
```

*Listing 4-2: Portions of the get_nums.h header file*

The `get_long()` function is designed to allow the caller to specify whether to accept certain types of strings, such as those with trailing non-numeric characters, or to accept a zero when the string has no digit at all, or to report negative values. Its return value, if negative, indicates some type of anomaly or error ❷. If there are no errors or anomalies, it returns `VALID_NUMBER`, which is defined as zero. Callers can easily ignore the specific error codes or take different actions depending on which they are.

The prototype has four parameters. The first is the string to be parsed. The second argument is an integer interpreted by the function as a set of flags. The following list includes the three possible flags ❶ and their meanings:

**`NO_TRAILING`** If passed into the function, it will return a `TRAILING_CHARS_FOUND` value for any string containing trailing non-numeric characters, including those that have no digits at all, returning the value of the digits it found.

**`NON_NEG_ONLY`** If passed into the function, it will return `NEG_NUM_FOUND` if the numeric value is negative.

**`ONLY_DIGITS`** If passed into the function, it will return `NO_DIGITS_FOUND` if the string has no digits and set `*value` to zero.

Because they're independent, they can be or-ed into a flag to pass to the function, as in

```
int flag = 0;
flag = flag | NO_TRAILING | ONLY_DIGITS;
```

The third parameter is a pointer to a location that can store the `long int` on successful return.

The fourth argument is the location in which to store an error message if things go wrong. Thus, if `get_long()` returns `VALID_NUMBER`, the number is in `*value`. If it returns anything else, the error message that it constructs is in `msg`.

Because the definition of `get_long()` is lengthy, to conserve space, I omit parts of it as well as comments in Listing 4-3. The book's source code distribution has the complete listing.

*get_long() partial*

```
int  get_long(char *arg, int flags, long *value, char *msg )
{
    char *endptr;
    long val;
    errno = 0;
    val = strtol(arg, &endptr, 0);

    if (errno == ERANGE) {
        if ( msg != NULL )
            sprintf(msg, "%s\n", strerror(errno));
        return FATAL_ERROR;
    } else if ( errno == EINVAL && val != 0 ) { /* Bad base; shouldn't happen */
```

```
            if ( msg != NULL )
                sprintf(msg, "%s\n", strerror(errno));
            return FATAL_ERROR;
        }
        if (endptr == arg) {
         ❶ if ( flags & ( ONLY_DIGITS | NO_TRAILING ) ) {
                if ( msg != NULL )
                    sprintf(msg, "No digits in the string\n");
                return NO_DIGITS_FOUND;
            }
            else { /* Accept a zero result */
                *value = 0;
                return VALID_NUMBER;
            }
        }
        if (*endptr != '\0') { /* Non-number characters follow the number. */
         ❷ if ( flags & NO_TRAILING ) {
                if ( msg != NULL )
                    sprintf(msg, "Trailing characters follow the number: \"%s\"\n",
                    endptr);
                return TRAILING_CHARS_FOUND;
            }
        }
        --snip--
        *value = val;
        return VALID_NUMBER;
}
```

*Listing 4-3: A partial listing of the get_long() function*

The function uses `sprintf()` to construct the message string. We use it the way we would use `printf()`, except we give it a string parameter preceding the format string. Instead of printing to standard output, it prints to the string. In the various places where we might call `sprintf()`, we first check that this pointer is not `NULL` before attempting to pass it to `sprintf()`.

After calling `strtol()`, it checks for the two possible error values that it might return. The `ERANGE` code implies a non-recoverable error (number out of the range of `long int`), and `EINVAL` implies either that there were no digits or that the base was bad. Since we set `base` to zero, implying actual base-10 numbers, we shouldn't get an `EINVAL` code unless there were no digits, but to be safe, we check for this possibility. For either of these errors, the caller receives the `FATAL_ERROR` code.

If `strtol()` succeeded, we start the process of looking at the flags that the caller sent and deciding whether the returned value violates any of them. Checking the flags uses the bitwise *and* and *or* operators ❶. For example, the expression `( flags & ( ONLY_DIGITS | NO_TRAILING )` bitwise-ands the bits of `flags` with the bitwise-or of `ONLY_DIGITS` and `NO_TRAILING`. The expression is true if and only if `flags` has one or both of these bits set. If `NO_TRAILING` is set

but `endptr` is not at the end of the string ❷, it implies that there are trailing characters and that the caller wants to be informed about it, so the function returns `TRAILING_CHARS_FOUND`. It returns `VALID_NUMBER` if the flag is not set because it implies that the calling code doesn't care whether there are trailing characters. Similar logic applies to the remaining flags, but that code is not shown.

The `get_int()` function, displayed in Listing 4-4, is much shorter, because it just calls `get_long()` and checks whether the number is within range for an integer, using the system constants, `INT_MAX` and `INT_MIN`.

*get_int()*

```
int  get_int(char *arg, int flags, int *value, char *msg )
{
    long val;
    int res =  get_long(arg, flags, &val, msg );

    if ( VALID_NUMBER == res ) {
        if ( val > INT_MAX ) {
            sprintf(msg, "%ld is out of range\n", val);
            return OUT_OF_RANGE;
        }
        else if ( val < INT_MIN ) {
            sprintf(msg, "%ld is out of range\n", val);
            return OUT_OF_RANGE;
            }
        else {
            *value = val;
            return VALID_NUMBER;
        }
    }
    else { /* get_long failed in one way or another */
        return res;
    }
}
```

*Listing 4-4: A function to get the integer value of a string*

Observe that `get_int()` doesn't have to check for any errors other than the numbers being out of range. It just passes the other error codes from `get_long()` to its caller. What may not be obvious is that the message that `get_long()` constructs will also be passed to the caller if `get_int()` doesn't overwrite it for a number that is out of range.

I wrote a couple of programs to call these functions (available in the source code distribution), passing various flags, to illustrate some of their error handling. The following listing shows some of their runs:

```
$ test_get_long   32kjk   # with NO_TRAILING passed
return:  TRAILING_CHARS_FOUND.
message: Trailing characters follow the number: "kjk"
$ test_get_long   32kjk   # without NO_TRAILING passed
```

```
return:  VALID_NUMBER
$ test_get_long   sdfskjk   # with ONLY_DIGITS passed
return:  NO_DIGITS_FOUND
message: No digits in the string
$ test_get_int   -28734898798798798798879879
return:  FATAL_ERROR
message: Numerical result out of range
```

In the first run, I passed the `NO_TRAILING` flag, and it reported seeing non-digit characters attached to the number. In the second run, I gave it the same input but without that flag, and it silently accepted the number without error. In the third run, I gave it a string with no digits and passed the `ONLY_DIGITS` flag, and it rejected the input, identifying the error. The last run was to show that it would successfully handle inputs that are too large and out of range.

### Common Error Handling Functions

All of the projects in this book share a few error handling functions, whose declarations are consolidated into a header file named *error_exits.h*, in the *include* top-level directory. Having a few, common error-handling functions simplifies the programs and reduces redundant code. Listing 4-5 shows the header file.

*error_exits.h*

```
#ifndef ERROR_EXITS_H
#define ERROR_EXITS_H
#include "sys_hdrs.h"

/** error_message()
   This prints an error message associated with errnum on standard error
   if errnum > 0. If errnum <= 0, it prints the msg passed to it.
   It does not terminate the calling program.
   This is used when there is a way  to recover from the error.           */
void error_mssge(int errornum, const char * msg);

/** fatal_error()
   This prints an error message associated with errnum on standard error
   before terminating the calling program, if errnum > 0.
   If errnum <= 0, it prints the msg passed to it.
   fatal_error() should be called for a nonrecoverable error.            */
void fatal_error(int errornum, const char * msg);

/** usage_error()
   This prints a usage error message on standard error, advising the
   user of the correct way to call the program.                          */
void usage_error(const char * msg);

#endif /* ERROR_EXITS_H */
```

*Listing 4-5: The header file with declarations of common error-handling functions*

The `error_message()` function expects a number and a string. If the number is positive, it is an `errno` value and it uses `strerror()` to format a string describing that error and prints that string to standard error. If the number is not positive, it prints the string passed into it instead.

The `fatal_error()` function is the same as `error_message()` except that it terminates the program after printing the message, calling `exit()` with the system-defined `EXIT_FAILURE` number as its argument. The `usage_error()` function prints a usage message on standard error and terminates the program. Listing 4-6 provides their implementations.

*error_exits.c*

```
void error_mssge(int errornum, const char * msg)
{
    if (errornum > 0) /* an errno value */
        fprintf(stderr, "%s\n", strerror(errornum));
    else /* project-defined error number - ignore and just print msg */
        fprintf(stderr, "%s\n", msg);
}

void fatal_error(int errornum, const char * msg)
{
    error_mssge(errornum, msg);
    exit(EXIT_FAILURE);
}

void usage_error(const char * msg)
{
    fprintf(stderr,"usage: %s\n", msg);
    exit(EXIT_FAILURE);
}
```

*Listing 4-6: Three error handling functions*

Although the `fatal_error()` and `usage()` functions look similar, it's convenient to have a separate function for displaying a message specifically when the user ran the program incorrectly.

## File Organization

The numeric parsing and error-handling functions just described are used by almost all programs in this book. To facilitate using them, I place their source code into a single top-level directory named *common*. The *include* and *lib* directories are at the same level as *common*, and each chapter has a directory at this same level, containing the sources for all programs referenced in the chapter. The *include* directory contains all header files that the projects include, and *lib* contains a static library named *libspl.a* containing all common object files from the *common* directory. This directory structure is depicted in Figure 4-1.

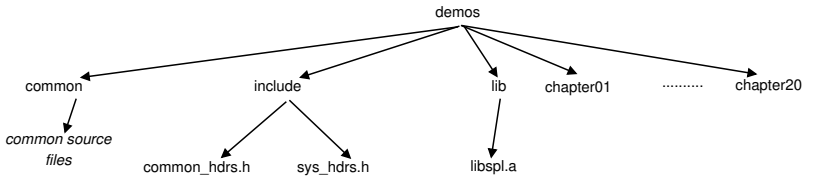



*Figure 4-1: The structure of the demo program directories with common code*

To create the *libspl.a* library, I use the GNU `ar` command. Appendix A contains an explanation of this command and detailed instructions for how to create static and shared libraries in general.

## Research for Our First System Program

The first program we'll write is a warm-up exercise, a relatively easy one that won't require much background knowledge. Its purpose is to show how we'll go through the steps described previously. We'll write a command that displays the current date and/or time in various formats. We know the operating system maintains this information somewhere, because lots of applications display the date and time. This exercise also has a few side benefits. We'll learn how Unix represents and processes dates and times. We'll learn the various components of the API related to time and date, and when we're finished, we'll have a few utility functions that'll be useful in later projects.

The first step is to check whether a command like this already exists and try to mimic its behavior. We can search the man pages to find such a command using `apropos`. The most obvious keywords to try first would be *date* and *time*. Since we're searching for commands, we limit the search to Section 1, first trying to match keyword *date*, as follows:

```
$ apropos -s1 date
aa-features-abi (1)  - Extract, validate and manipulate AppArmor feature abis
apport-bug (1)       - file a bug report using Apport, or update an existing ...
apport-collect (1)   - file a bug report using Apport, or update an existing ...
autoreconf (1)       - Update generated configuration files
autoupdate (1)       - Update a configure.ac to a newer Autoconf
--snip--
```

On my system, 64 lines of output were displayed, but I'm showing only the first five lines. Several of these lines are descriptions of commands that have nothing to do with dates or times. Why is this? Remember that keyword searches in their simplest form display any short descriptions that contain the keyword, even as a substring. We need to request an exact match instead:

```
$ apropos -s1 -e date
cal (1)              - displays a calendar and the date of Easter
date (1)             - print or set the system date and time
date (1posix)        - write the date and time
hp-timedate (1)      - Time/Date Utility
idevicedate (1)      - Display the current date or set it on a device.
```

```
mate-time-admin (1)  - set date and time
ncal (1)             - displays a calendar and the date of Easter
timedatectl (1)      - Control the system time and date
```

From this short list we can see that there are two man pages for a command named `date`, one in Section 1 and the other in Section 1POSIX.

Here's part of the man page for `date` in Section 1POSIX:

```
DATE(1POSIX)              POSIX Programmer's Manual             DATE(1POSIX)

PROLOG
       This manual page is part of the POSIX Programmer's Manual.  The Linux
       implementation of this interface may differ (consult the  corresponding
       Linux manual page for details of Linux behavior), or the interface may
       not be implemented on Linux.
NAME
       date — write the date and time

SYNOPSIS
       date [-u] [+format]
       date [-u] mmddhhmm[[cc]yy]
--snip--
```

This page describes what a POSIX-conforming version of the `date` command should do; it's a specification. It warns us that it isn't necessarily what is running on our Linux system.

Let's look at the page in Section 1. That page begins as follows:

```
DATE(1)                          User Commands                         DATE(1)

NAME
       date - print or set the system date and time

SYNOPSIS
       date [OPTION]... [+FORMAT]
       date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]]
--snip--
```

This version has more options. If we were running a version conforming to POSIX, several options wouldn't be available.

To start, we'll just run the command without options to see its output:

```
$ date
Thu Dec  8 02:46:53 PM EST 2022
```

This output raises a few questions:

- It displays not just the current date, but the time of day and the day of the week. How does it know what day of the week it is?

- It outputs the string EST, which is short for *Eastern Standard Time*. This implies that the system stores time zone information and that there's some way to determine in which time zone we're running. How can we find that information?
- How does it retrieve the current time? Is this a system call?
- How does date choose the format of date/time to print?

The rest of the Section 1 man page shows that the command has several useful options that aren't available in POSIX. If we try a few, we'll see that we're running the Linux version of the command:

```
$ date -d'next Thu'
Thu Jan 26 12:00:00 AM EST 2023
$ date  -d'next month'
Wed Feb 22 02:02:28 PM EST 2023
$ date -d"2038-01-19 03:14:07 UTC"  # time of end of Unix Epoch
Mon Jan 18 10:14:07 PM EST 2038
$ date -d'5 years ago'
Mon Jan 22 02:36:24 PM EST 2018
```

The -d option lets us request that date print dates other than the current one, which is a useful feature, and it even allows expressions such as *five years ago* and *next Thursday*. This option is not detailed much in the man page. Instead, its author wrote the following note there: "The date string format is more complex than is easily documented here but is fully described in the info documentation."

More important for us right now is that it has options for controlling the format of the output date/time string. We can change the format by supplying an option of the form +"*FORMAT*" where *FORMAT* is a string that contains ordinary character sequences as well as character sequences called *format specifications*, each of which is introduced by a % character and followed by a second character called a *format specifier character*. Each format specification defines one or more pieces of date or time information formatted in a particular way. For example, %m is replaced on output by a two-digit month number, such as 04. The ordinary character sequences in *FORMAT* (called *literals*) are output exactly as they're written in the string. For example, the format "The month is %m" is output as "The month is 04" if the current month is April. Some common format specifiers are %a, which is replaced by the three-letter weekday name, such as Sun for Sunday, %b, which is replaced by the three-letter month name, such as Dec for December, and %D, which is replaced by a date in the form *mm/dd/yy*, such as 01/01/1972. Appendix C contains a comprehensive list of specifiers. Here are a few examples of output when the date is the end of the Unix Epoch, January 19, 2038 at 03:14:07 UTC:

```
$ date +"%A, %D"     # full day name, literal comma, and American date
Monday, 01/18/38
$ date                # default format
Mon Jan 18 10:14:07 PM EST 2038
```

```
$ date +"%c"          # locale's date and time
Mon 18 Jan 2038 10:14:07 PM EST
$ date +"It is %A at %R." # full day name, 24-hour time
It is Monday at 22:14.
```

Notice that the format string contains a mix of format specifiers and literals. The literals are displayed, uninterpreted, where they appear relative to the format specifiers. Also observe that the `%c` format specifier produces the default format used by `date` and that this is called the *locale's date and time* in the documentation.

Several of the format specifiers refer to the user's locale in their descriptions. For example, the `%a` and `%A` are the locale's abbreviated and full weekday names, respectively. In the United States, these are names such as *Sun* and *Sunday*, respectively, but we have yet to see what they would be if we could choose a different locale. We'll see how to do that later in this chapter in "Working with Locales" on page 169.

The man page for `date` doesn't specify what its default format is. We can see what it looks like, and we know that it's the locale's format, but we don't know why it's in that form. However, the `SEE ALSO` section tells us that the full documentation is available in two places: the Info documentation and on the GNU website page (*https://www.gnu.org/software/coreutils/date*). Both sources state that "invoking `date` with no FORMAT argument is equivalent to invoking it with a default format that depends on the `LC_TIME` locale category. In the default C locale, this format is `+"\%a \%b \%e \%H:\%M:\%S \%Z \%Y"`, so the output looks like `Thu Mar  3 13:47:51 PST 2005`. Clearly, we need to know more about locales to understand this explanation, but for now we'll focus on writing some simple programs that behave like `date`, and we'll explore locales later in this chapter.

Our first goal is to write a much simpler version of the `date` command without any of its command line options to reproduce its default behavior. Once we do that, we'll add the ability to customize the output format using format specifiers, and after that we'll see how to make a version of it that's sensitive to locale settings. We'll name the first version of the command `spl_date1` and name the program's source code file, *spl_date1.c*.

## Designing the spl_date Program: Version 1

For `spl_date1`, we'll follow this program logic:

1. Get the current time.
2. Format the time in the default form of the `date` command, storing it into a string.
3. Print the string on standard output.

The first two steps imply that we need to understand how time is represented in Unix and find ways to change its representation to different kinds of human-readable forms.

Whenever you embark on a new project, it's a good idea to search for help in Section 7 of the man pages. Section 7 contains overviews of various topics. It might have an overview on the topic of interest, in this case time. If so, it will contain background concepts and possibly references to functions that you might need.

To check whether Section 7 has a page that might help, enter this command: **`apropos -s7 -e time`**, It returns, among others, the following pages of interest:

```
RESET (7)            - restore the value of a run-time parameter to the defau...
SET (7)              - change a run-time parameter
bootparam (7)        - introduction to boot time parameters of the Linux kernel
SHOW (7)             - show the value of a run-time parameter
sys_time.h (7posix)  - time types
systemd.time (7)     - Time and date specifications
time (7)             - overview of time and timers
time.h (7posix)      - time types
time_namespaces (7)  - overview of Linux time namespaces
```

Of these, the `time (7)` man page looks like the best starting point. It summarizes what we need to understand about time in Unix and Linux.

### About Calendar Time in Unix

The man page explains that Unix has two types of time: process time and real time. *Process time* is the time a process spends in the CPU. In Chapter 5, I cover how to obtain process time. *Real time* is elapsed time measured from some reference point. When that reference point is not fixed, real time is called *elapsed time*. *Timers* are objects in Unix that can be set and used to keep track of elapsed time. They're important, and we'll revisit them in later chapters, but they're not what we want right now. *Calendar time* is real time with respect to a particular fixed time point called the Epoch. The *Epoch* is defined as January 1, 1970 at 00:00:00 UTC. It's approximately when the first Unix system was released. *UTC* is short for *Coordinated Universal Time*, which used to be called *Greenwich Mean Time*. The acronym is correct; the order of the letters was a compromise by the international advisory group that created it. See the NIST.gov website for an explanation (*https://www.nist.gov/pml/time-and-frequency-division/nist-time-frequently-asked-questions-faq*). Since its inception, in Unix, calendar time has been measured as the number of seconds elapsed since the Epoch.

### Broken-Down Time

The `time (7)` man page also mentions a type of time representation called *broken-down time*, which is a time representation that's broken down into various commonly used components. As Robert Grudin put it in *Time and the Art of Living* [6]:

> Our units of temporal measurement, from seconds on up to months, are so complicated, asymmetrical and disjunctive so as to make coherent mental reckoning in time all but impossible. ... It is as though architects had to measure length in feet, width in meters and height in ells; as though basic instruction manuals demanded a knowledge of five different languages.

We measure time in years, months, weeks, days, hours, minutes, and seconds, but we also use days of the week, days of the month, months of the year, and so on. These units have little consistency, other than the number of seconds in a minute and minutes in an hour being the same. A broken-down time structure consolidates all of this information into a single data structure, called a `struct tm`, which is used by several functions that convert time and date formats from one form to another. The man page mentions some of them and suggests looking at the `ctime()` man page. If we look at that page, we see that `asctime()`, `ctime()`, `gmtime()`, `localtime()`, `mktime()`, `strftime()`, and `strptime()`, as well as various thread-safe versions of these, are all time-conversion functions. We'll examine these functions in "Time Conversion Functions" on page 146 to decide which we should use, but first we'll examine the `struct tm` data structure, which is defined in the *time.h* header file as follows:

```
struct tm {
    int tm_sec;    /* Seconds (0-60) */
    int tm_min;    /* Minutes (0-59) */
    int tm_hour;   /* Hours (0-23) */
    int tm_mday;   /* Day of the month (1-31) */
    int tm_mon;    /* Month (0-11) */
    int tm_year;   /* Year - 1900 */
    int tm_wday;   /* Day of the week (0-6, Sunday = 0) */
    int tm_yday;   /* Day in the year (0-365, 1 Jan = 0) */
    int tm_isdst;  /* Daylight saving time */
};
```

The fields have their expected meanings, but there are two details to note:

- The `tm_sec` field stores the number of seconds after the minute, which is normally in the range 0 to 59, but it can be up to 60 to allow for leap seconds.
- The `tm_isdst` field is a flag that indicates whether daylight saving time is in effect at the time described. The value is positive if daylight saving time is in effect, zero if it is not, and negative if the information is not available.

Now that we know that functions exist to convert formats and that they use this broken-down time structure, we can turn to the question of how to get the current time.

### Getting Calendar Time in Unix

The `time (7)` man page suggests two different system calls for obtaining the calendar time: `clock_gettime()` and `time()`. From the man page for `clock_gettime()`, we learn the following.

- The general-purpose `clock_gettime()` function provides the time of a specified clock. Its synopsis is:

```
#include <time.h>
int clock_gettime(clockid_t clockid, struct timespec *tp);
```

  The function is given the ID of a clock and a pointer to a `timespec` struct, in which, on return, it stores the time on that clock, measured in nanoseconds.

- The predefined constant `CLOCK_REALTIME` of type `clockid_t` is the ID of the clock that keeps track of calendar time. The `timespec` struct is defined in *time.h* as:

```
struct timespec {
    time_t   tv_sec;        /* seconds */
    long     tv_nsec;       /* nanoseconds */
};
```

  The `time_t` type is a signed integer type whose size is implementation dependent, but is at least 32 bits.

- To make sure the function is exposed in the header files, there is a feature test macro requirement:

```
_POSIX_C_SOURCE >= 199309L
```

  This implies that we need to set `_POSIX_C_SOURCE` to that value or higher before including any header files.

- The man page also states that on POSIX systems on which this function is available, the symbol `_POSIX_TIMERS` is defined in *unistd.h* to a positive value. If we want to design a program to choose a different method of getting calendar time in case this function is not available, we would insert conditional code of the following form:

```
#if _POSIX_TIMERS > 0
    /* use clock_gettime() to get time */
#else
    /* use some other function such as time() */
#endif
```

- In the `SEE ALSO` section, it suggests a few pages that we should read: `gettimeofday()` and `time()`, both in Section 2. We should also read the man page for *time.h*.

The time() system call is much simpler to use and understand. Its man page tells us the following.

- time() returns the number of seconds since the Epoch.
- Its synopsis is

```
#include <time.h>
time_t time(time_t *tloc);
```

  The argument is a pointer to an integer of type time_t, but it's allowed to be NULL because its return value is also the current time.
- When tloc is NULL the function cannot fail, obviating the need for error-handling.

Before we decide which of these two functions to use, we look at the man page for the gettimeofday() function suggested in the SEE ALSO Section. Its synopsis is

```
#include <sys/time.h>
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

where the tv argument is a pointer to timeval struct defined in *<sys/time.h>)* as follows:

```
struct timeval {
       time_t      tv_sec;     /* seconds */
       suseconds_t tv_usec;    /* microseconds */
};
```

On return, this stores the number of seconds and microseconds since the Epoch. The timezone struct pointer tz should always be set to NULL because it has been *deprecated*.

**NOTE** *Whenever you see a feature marked as deprecated in documentation, avoid using it. If the organization that supports the software has deprecated it, that means it will no longer support it and it will become obsolete.*

In fact, the CONFORMING TO section notes that the function itself has been marked as obsolete in POSIX.1.2017, and it recommends using clock_gettime() instead.

The choice is thus reduced to clock_gettime() and time(). The difference is in how the returned time is represented and its granularity. Since the tv_sec field of the struct timespec returned by clock_gettime() is the number of seconds since the Epoch, it should be the same as the value returned by time(). For our program, we don't need sub-second granularity, so there's little benefit to using clock_gettime(). On the other hand, it's a more adaptable function. Another factor to consider is performance. To check whether there is a price to pay for obtaining the finer resolution of the timespec structure, I wrote two programs that called each function 10 million times and measured their elapsed times when run on my x86-64 system running Linux

5.15.0. The program that called time() required 0.032 seconds, whereas the one running clock_gettime() required 0.171 seconds. Repeated runs had similar results. Taking everything into consideration, we'll choose time() for getting the current time in this first version of our program.

### Time Conversion Functions

The functions asctime(), ctime(), gmtime(), localtime(), and mktime() are all described on the same man page. Entering **man ctime** will display it. Here's the relevant portion of that page, with some lines removed:

```
#include <time.h>

char *asctime(const struct tm *tm);
char *ctime(const time_t *timep);

struct tm *gmtime(const time_t *timep);
struct tm *localtime(const time_t *timep);

time_t mktime(struct tm *tm);
```

We observe that

- asctime() is given a broken-down time struct and returns a string,
- ctime() is given a time_t value and returns a string,
- gmtime() and localtime() are each given a time_t value and return a pointer to a broken-down time struct,
- mktime() is given a broken-down time struct and returns a time_t value, and
- None of these functions require time resolution smaller than seconds, reinforcing our decision to use time() instead of clock_gettime() to get the current time.

This shows that we need to use either asctime() or ctime() to create a formatted time string, but if we use asctime() we need to convert from calendar time to the broken-down time first. Reading the DESCRIPTION section reveals that the difference between gmtime() and localtime() is that gmtime() converts its time_t argument to broken-down time expressed in UTC, whereas localtime() converts its time_t argument to broken-down time expressed relative to the user's specified time zone. We'll have more to say about time zones later in this chapter, in "About Time Zones" on page 165.

The CONFORMING TO section of that page, however, states that both asctime() and ctime() are marked as obsolete and that strftime() should be used in their place, ruling them out. Reading the man page for strftime(), we learn that it's a much more powerful function than either of them:

```
$ man strftime
  --snip--
```

```
SYNOPSIS
    #include <time.h>
    size_t strftime(char *s, size_t max, const char *format,
                    const struct tm *tm);
DESCRIPTION
       The strftime()  function formats the broken-down time tm according to
       the format specification format and places the result in the character
       array s of size max.
  --snip--
```

The rest of its description tells us that `strftime()` lets us customize the output date and time string by using a *format specification*, which is its third parameter. In fact, the set of format specifiers is almost identical to those used by the `date` command, making our job fairly easy. We just have to use the default format of `date`, which we know is `%c`. (Although the man page states that there's a format specification `%+` that produces a string in the exact same format as the `date` command, it isn't supported in *glibc version 2*.) Thus, to obtain a string in the same format as `date` such as:

```
Tue Jan 24 11:24:07 AM EST 2023
```

we'll pass the format specification `%c` to `strftime()`.

We can put together the *spl_date1.c* program based on what we've learned. Figure 4-2 displays the program logic.

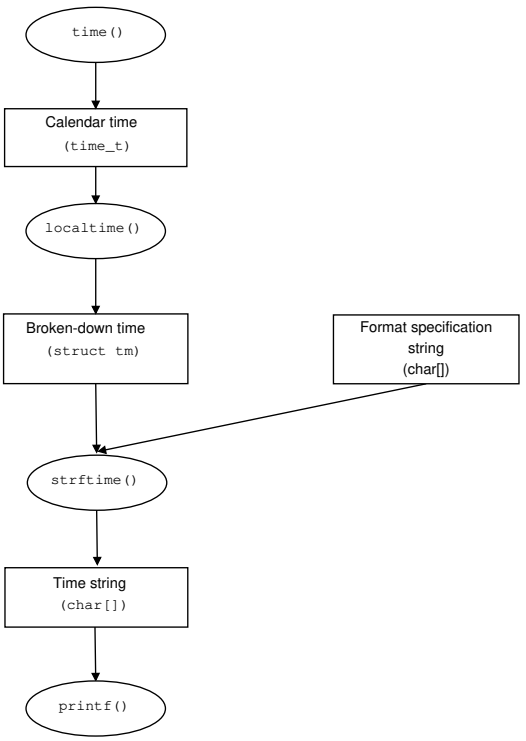



*Figure 4-2: Program flow of spl_date1.c*

We use `time()` to get the current time in calendar time units and pass that return value to `localtime()`, which constructs a broken-down time object. We pass that in turn to `strftime()` in addition to the format specification `%c`, hard-coded into the call. Finally, we print out the string produced by `strftime()`. The resulting program, *spl_date1.c*, is displayed in Listing 4-7 with some comments omitted to save space.

*spl_date1.c*

```
#define _GNU_SOURCE
#include "common_hdrs.h"
#define FORMAT  "%c"

int  main(int argc, char *argv[])
{
    char       formatted_date[200];
    time_t     current_time;
    struct tm *broken_down_time;

    current_time = time(NULL); /* Get the current time. */

    /* Convert current time into broken-down time. */
    broken_down_time = localtime(&current_time);

    if (broken_down_time == NULL)
        fatal_error(EOVERFLOW, "localtime");

    /* Create a string from the broken down time using the %c format. */
    if (0 == strftime(formatted_date, sizeof(formatted_date),
             ❶ FORMAT, broken_down_time) ) {
        fatal_error(EXIT_FAILURE, "Conversion to a date-time string"
                    " failed or produced an empty string\n");
    }
    printf("%s\n", formatted_date);
    return 0;
}
```

*Listing 4-7: The first version of the spl_date program*

Rather than hard-coding the `%c` format specifier ❶ directly into the call to `strftime()`, we make it a macro and pass the macro definition. This makes it easier to change the program in the next version.

## Designing the spl_date Program: Version 2

We'll now improve *spl_date1.c* by replacing the hard-coded format specifier with a variable, so we can run it using different date formats specified on the command line. For example, it would be useful if we could enter commands such as the following:

```
$ spl_date +"Today is %A. Current time: %R"
```

```
Today is Sunday. Current time: 13:45
```

In this way different users could see the time in their format of choice.

The changes to the program to do this are fairly simple. First, we'll need a variable to store a format string. We'll declare a string variable named `format_str` of sufficient length to store reasonably sized formats, which we do by defining a `MAXLEN` macro constant equal to `STRING_MAX` (declared in *include/-common_hdrs.h* as 1024), and declaring `format_str` to be size `MAXLEN`. Second, instead of calling `strftime()` with the hard-coded format, we'll call it with the new variable:

```
strftime(formatted_date, sizeof(formatted_date), format_str, broken_down_time)
```

We also need to check whether the command has an argument, and if so, whether it starts with a `+` and whether it's small enough to fit into `format_str`. If so, we can pass the string following the `+` to `strftime()`. If that string isn't a valid format string, `strftime()` will return an error that we can report; otherwise, we print the string that it produces. If there's no argument to the program, we just print the current time in the default format. This logic is expressed in the following code:

```
if ( argc < 2 )  /* No argument - use default   */
    strcpy(format_str, FORMAT);
else {
    if ( argv[1][0] == '+' )  /* Argument starts with + */
        if ( strlen(argv[1]+1) < MAXLEN ) /* Format is small enough. */
            strncpy(format_str, argv[1]+1, MAXLEN-1);
        else {  /* format_str exceeds size we allocated. */
            fprintf(stderr, "%s: format string length is too long\n",
            basename(argv[0]));
            exit(EXIT_FAILURE);
        }
    else {
        fprintf(stderr,"%s: format should be +\"format-string\"\n",
                basename(argv[0]));
        exit(EXIT_FAILURE);
    }
}
```

We put all of these changes together into a second version of the program, named *spl_date2.c*; partially displayed in Listing 4-8. Th complete program is in the source code distribution for the book.

*part of spl_date2.c*

```
--snip--
#define FORMAT  "%c"
#define MAXLEN  STRING_MAX        /* Maximum size of message string */

int  main(int argc, char *argv[])
{
```

```
    char       formatted_date[MAXLEN];
    time_t     current_time;
    struct tm *broken_down_time;
    char       format_str[MAXLEN];
    char       err_msg[MAXLEN];        /* For error messages        */

    if ( argc < 2 )                    /* No argument - use default */
        strcpy(format_str, FORMAT);
    else {
        if ( argv[1][0] == '+' )       /* Argument starts with +.    */
            if ( strlen(argv[1]+1) < MAXLEN )
                strncpy(format_str, argv[1]+1, MAXLEN-1);
            else {
                sprintf(err_msg, "format string length is too long\n");
                fatal_error(BAD_FORMAT_ERROR, err_msg);
            }
        else {
            sprintf(err_msg,"%s: format should be +\"format-string\"\n",
                    basename(argv[0]));
            fatal_error(BAD_FORMAT_ERROR,err_msg);
        }
    }
--snip--
    if (0 == strftime(formatted_date, sizeof(formatted_date),
             format_str, broken_down_time) ) {
        fatal_error(BAD_FORMAT_ERROR, "Conversion to a date-time string"
                    " failed or produced an empty string\n");
    }
    printf("%s\n", formatted_date);
    return 0;
}
```

*Listing 4-8: A partial listing of the second version of `spl_date`, allowing an optional user-supplied format string argument*

Following are a few runs of this program that show how it handles some possible errors and produces the expected output:

```
$ spl_date2
Mon Feb 13 11:01:22 2023
$ spl_date2 today
spl_date2: format should be +"format-string"
$ spl_date2 +"Today is day %e of %B. It is now %r"
Today is day 13 of February. It is now 11:04:27 AM
$ spl_date2 +"a very long string, longer than 200 characters  ..."
spl_date2: format string length is too long
```

The last run is given a format string whose length exceeds the size of the buffer that the program uses to show how it handles this error. You can see that it detects it and exits without crashing.

## Designing the spl_date Program: Version 3

The preceding program wasn't too hard to develop, but it prepared us to go one step further and add the ability to display dates in the past and future by allowing the user to specify lengths of time to add to or subtract from the current time. This program will be a bit more challenging to write.

### *Specifying the User Interface*

First, we need to decide how the user should call the program. Having done that, we'll write a precise specification against which we can test our implementation. Since the ability to add an amount of time is optional, the program should have a command line option with a required argument that specifies the amount of time to add or subtract to the current time. We'll call the amount of time that we want to add or subtract, the *time-adjustment*, and we'll use the option character `-d`, (for ***d****ifference*.) We should also get into the habit of providing a help feature for every nontrivial program we write. The Unix convention is to give commands a `-h` option that displays usage information for the command. If that option is present, the convention is to ignore all other words on the command line. To summarize, our program's synopsis should be of the form

```
$ spl_date3   [-h] | [-d "time-adjustment"] [+"format-specification"]
```

where the ***`time-adjustment`*** argument to **`-d`** is a string that we need to define, and the ***`format-specification`*** is the same as it was in `spl_date2`.

#### Specifying Time Adjustments

When we write amounts of time in non-computer contexts, we understand that the expressions "1 month, 8 days," "one month and eight days," and "one month, eight days" are equivalent amounts of time. If we allowed users to enter amounts of time with that degree of flexibility, we'd be making the task of parsing the input much harder than if we limited the form of the input to something simpler. It amounts to a trade-off between what's easy for the user and what's easy for the programmer. Since we're not trying to write production software yet, we need a compromise that provides a convenient interface for the user and a relatively easy syntax to parse.

The solution I've chosen is to allow the user to enter time differences in the customary units we use, namely years, months, weeks, days, hours, minutes, and seconds, but not to enter phrases such as "next Monday" or "last month," which, while very convenient, would add more parsing to the program. To make the parsing easier, we'll require the user to enter numerals rather than words for the amounts. For example, the program should accept a phrase such as "2 years 3 weeks" but not "two years three weeks" or "two

years, three weeks." The goal of the project is to learn about programming with dates and times, not to spend time parsing strings efficiently.

Also, we'll give users the ability to enter times in the past by allowing negative numbers for the time quantities, so we'll accept a phrase such as "−4 hours 5 minutes," which could also be entered as "−3 hours −55 minutes."

To simplify the program, we'll forbid fractional amounts, such as "3.5 days," but we'll allow users to enter the same unit multiple times. For example, we'll allow users to enter a time adjustment such as:

```
1 year 4 months -2 days -3 weeks +1 day
```

The way that I've formulated this, commas between the units are not allowed.

I'll write a specification of the time adjustment using the following grammar, which uses the same syntax as the man page synopsis:

```
time-adjustment = <num> <time-unit> [<num> <time-unit> ... ]
num             = [+|-]<integer>
time-unit       = year[s] month[s] week[s] day[s]  hour[s] minute[s] second[s]
integer         = [1-9][0-9]...
```

Notice that the time units can have an optional `s` on the end, that numbers can start with an optional `+` or `-`, and that they cannot start with leading zeros. If they use a leading zero, the number will be interpreted as an octal number.

Here are some examples:

```
$ spl_date3  -d "1 year 2 months"
$ spl_date3  -d " +1 year 2 months" +"%D %r"
$ spl_date3  -d " 3 weeks 5 days 4 hours 30 minutes" +"%D %r"
$ spl_date3  -d" -5 months -3 days"  +"%D"
```

The first shows the date one year and two months from today in the default format. The next shows the same date but in a different format of the form "mm/dd/yyyy hh:mm:ss AM|PM". The third shows the date that is three weeks, five days, four hours, and 30 minutes from the current time using the same format as the preceding example. The last shows a date five months and three days earlier, using the default format.

### Fuzzy Time

The last consideration before we start to map out the program logic concerns the fuzziness of months and years as units of time. The number of days in a month depends on the month, and the number of days in a year changes for leap years. If we subtract one month from July 31st what is the date? Since there is no June 31st, is it June 30th? If you read the Info page for the `date` command, you'll see that its implementation uses the rule that adding (or subtracting) a month increments (or decrements) the month number, and if the date does not exist in that month, it's adjusted to the

nearest date that's valid. We can test how the real `date` adjusts these dates using its `--date=` option:

```
$ date --date='Dec 29, 2022  +2 months'
Wed Mar  1 12:00:00 AM EST 2023
# Because Feb 29 does not exist, it makes it one day after Feb 28,
which is Mar 1.
$ date --date='Dec 30, 2022  +2 months'
Thu Mar  2 12:00:00 AM EST 2023
# Because Feb 30 does not exist, it makes it two days after Feb 28,
which is Mar 2.
$ date --date='Mar 30, 2022  -1 month'
Wed Mar  2 12:00:00 AM EST 2022
# It uses the same idea as before. It does not matter whether it got the
day by adding or subtracting.
$ date --date='Dec 30, 2023  +2 months'
Fri Mar 1 12:00:00 AM EST 2024
# It knows that 2024 is a leap year, so it makes it one day past Feb 29.
```

For consistency, our program should use this same date calculation logic, but this raises the question, is there a library function that does this calculation, or do we have to implement it ourselves? If we return to the man page for `ctime()`, we'll see that it has relevant information about the `mktime()` function:

> The `mktime()` function modifies the fields of the `tm` structure as follows: `tm_wday` and `tm_yday` are set to values determined from the contents of the other fields; if structure members are outside their valid interval, they will be normalized (so that, for example, 40 October is changed into 9 November); `tm_isdst` is set (regardless of its initial value) to a positive value or to 0, respectively, to indicate whether DST is or is not in effect at the specified time . . .

In short, `mktime()` encapsulates the corrections for invalid dates and times used in the `date` command, saving us from having to implement this logic ourselves. Therefore, we can add time adjustments to a broken-down time structure `bd_time` and call `mktime(bd_time)` to have `mktime()` normalize the time for us.

### *Program Logic*

How this version of `showdate` differs from the preceding one will guide the changes in the program logic. The first step is to list the changes:

- We have to add option parsing.
- We have to parse the time adjustment, if it's present, into the numbers of seconds, minutes, hours, and so on, that need to be added (or subtracted) from the current time.

- We need to add the time adjustment to the current time and display the resulting time.

We can incorporate these differences into the program's control flow, ignoring error handling for the moment, in the following sequence of steps:

1. Parse the command line, checking whether the `-d` or `-h` options are present.
2. If `-h` is present, ignore all other arguments and options, print out help information, and exit.
3. Otherwise, if `-d` is present, allocate memory to store its argument and copy the argument into that memory.
4. If there is a format specification, copy it into a string of sufficient size.
5. Obtain and store the current time into a `time_t` variable using `time()`.
6. Convert the current time into a broken-down time representation using `localtime()`.
7. If `-d` is present, parse the argument, creating a temporary broken-down time structure that stores the time to add to the current time in terms of years, months, days, and so on, and add the value of the temporary structure to the broken-down current time.
8. Use the `strftime()` function to format the output string representation of the broken-down time.
9. Print the formatted string using `printf()`.

Figure 4-3 shows the control flow with the new logic shown in boldface.

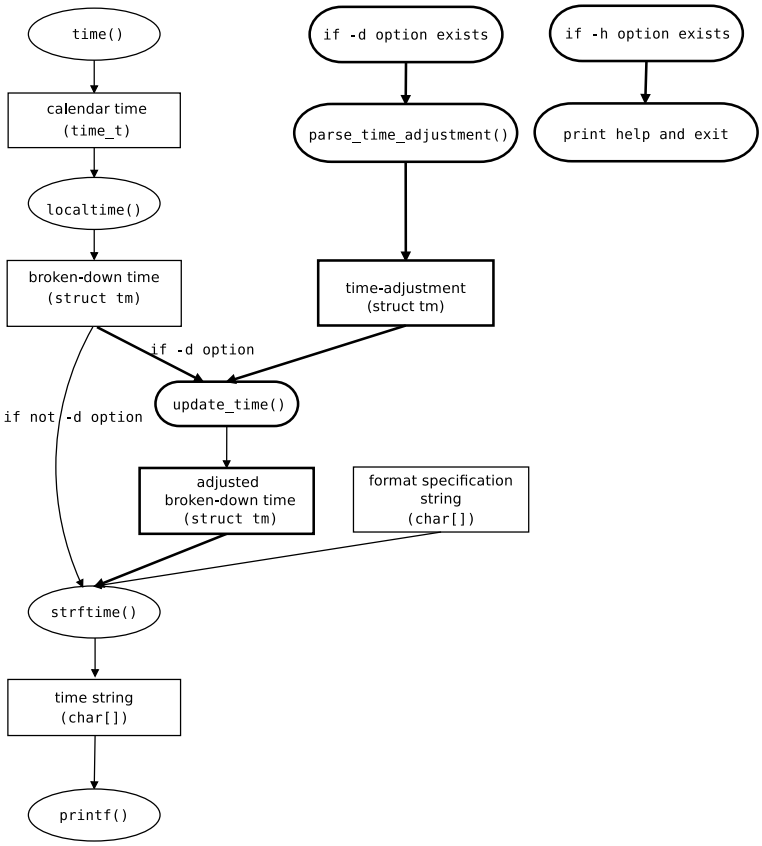



*Figure 4-3: The control flow of showdate3.c*

The next step is to prototype and design the function that parses the time-adjustment string. Since this function should receive a time-adjustment string as its input and create a broken-down time representation of that string as its output, a reasonable prototype for it is the following:

```
int parse_time_adjustment( /* IN */  char* time_adjust_string,
                           /* OUT */ struct tm* datetm );
```

Following convention, the return value is an indication of success or failure, and the broken-down time structure is a result parameter. Given the string `time_adjust_string` passed to the program following the `-d` option, and the address of a broken-down time structure, `parse_time_adjustment()` parses the string and sets the individual members of the structure to the amounts specified in the `time_adjust_string` passed to it. For example, if the `time_adjust_string` is

```
2 years   4 months  12 days  -6 hours  -2 days
```

it should set the structure's members as follows:

```
datetm->tm_year = 2
```

```
datetm->tm_mon  = 4
datetm->tm_mday = 10
datetm->tm_hour = -6
--snip--
/* All other members set to zero */
```

In principle, we can design this code from scratch and parse the string without any need to call a library function, reading each character from left to right, processing them as needed. For example, we could skip white space, build numbers when we see a plus or minus sign or a digit, and build time-unit strings when the characters are alphabetic. Processing this way makes one pass over the string and is the fastest possible approach. However, we need to process only the command line, not thousands of large strings, implying that the amount of time we'll save with this approach is imperceptible. It would be far better to take advantage of existing library functions that have been well tested, even if we end up making two passes across the string.

**PERFORMANCE AND DESIGN CONSIDERATIONS**

There's usually a trade-off between code that's easy to read and code that performs well. In designing a system program, we should certainly aim for good performance, but we also want to write code that's easy to understand and maintain. What principles can guide the algorithms we choose?

- Code that is not executed much doesn't need to be fast because even if it is a few orders of magnitude slower than it could be, it won't add any noticeable amount to the total running time. In contrast, code that's executed frequently should be fast.
- It's safer to use code that has been already written and tested thoroughly than to write new code to solve the exact same problem.
- Code that will be in service a long time should be easier to maintain than code that you know will be obsolete sooner.

I usually ask myself these questions when I design algorithms and need to decide how to make the trade-offs.

What functions can we use? Again, the first step is to consult the man pages. If we try using **`apropos -s3 string`** or **`apropos -s3 -e string`** to see which man pages in Section 3 are related to strings, we'll get a very long list that we can search by hand. On the other hand, we could see if there's a man page named `string`. If we do that, we'll discover a new resource:

```
$ man string
STRING(3)                 Linux Programmer's Manual                STRING(3)

NAME
       stpcpy,  strcasecmp,  strcat, strchr, strcmp, strcoll, strcpy, strcspn,
```

```
       strdup, strfry, strlen, strncat, strncmp, strncpy,  strncasecmp,  strp
       brk,  strrchr, strsep, strspn, strstr, strtok, strxfrm, index, rindex -
       string operations

SYNOPSIS
       #include <strings.h>

       int strcasecmp(const char *s1, const char *s2);
              Compare the strings s1 and s2 ignoring case.

       int strncasecmp(const char *s1, const char *s2, size_t n);
              Compare the first n bytes of the  strings  s1  and  s2  ignoring
              case.
--snip--
```

We could also read the *string.h* man page, but this one is better because the *string.h* man page is a POSIX page saying what should be present in a POSIX-compliant system, whereas this one is what actually is on our system. On GNU/Linux all of the functions listed in the *string.h* man page are available, possibly with different behavior than POSIX.1-2017 requires. No matter which you choose, it will be informative and will provide guidance and clues for choosing the right tool for the job.

If you study the list of functions in the *string.h* man page, you'll see one named `strtok()` with this prototype:

```
#include <string.h>
char *strtok(char *s, const char *delim);
```

The description states that it extracts *tokens* from the string `s` that are delimited by one of the bytes in `delim`. Tokens are pieces of a string to be parsed. The `strtok()` library function is a great tool for the job of breaking up a line into tokens separated by any types of delimiters. For example, if you're given a comma-separated values (CSV) file and need to extract its fields, you could use this function passing a comma as a delimiter.

The `delim` string is the set of characters that act as delimiters. If the string is `:,;`, then each of those characters will be treated as a character separating two tokens. For our purpose, we set `delim = " \t"` because the tokens in the time-adjustment string are separated by whitespace characters, including tab characters.

The first time we call `strtok()`, we pass the string to be parsed in the first argument. Its return value will be a pointer to the first token it finds. All returned tokens are terminated with a null byte (`\0`) so that string-processing functions can be used safely with them. In subsequent calls, we pass the `NULL` pointer in the first parameter. If there are no more tokens, it returns `NULL`, so the standard way to use it is essentially as follows:

```
char* delim = " \t"; /* Space and tab */
char* token;
token = strtok(mystring, delim);
```

```
while ( token != NULL ) {
    /* Process the token just found. */
    token = strtok(NULL, delim);
}
```

The `strtok()` function actually makes a copy of the string that you pass it, and as it finds each token, it replaces the delimiter at the end of it with a terminating null byte, `\0`.

Since our program expects the time-adjustment string to be a sequence of pairs of the form *<number> <whitespace> <time-unit>*, each iteration of the loop should call `strtok()` twice, the first time to get a number and the second to get a time unit. We'll declare the following variables:

```
char* delim   = " \t";     /* Space and tab             */
char* token;               /* Returned token            */
int   number;              /* To store number token     */
char  err_msg[STRING_MAX]; /* For error messages        */
int   flags   = ONLY_DIGITS | NO_TRAILING;
int   res;                 /* Return value of get_int() */
```

By setting `flags` to `ONLY_DIGITS | NO_TRAILING`, we reject numbers that have any non-digits following them and strings that have no digits at all where numbers are expected.

The pseudo-code structure of the loop is:

```
token = strtok(time_adjust_string, delim);
while ( token != NULL ) {
    res =  get_int(token, flags, &number, err_msg );        /* Get number */
    /* If error, handle it. */
    token = strtok(NULL, delim);   /* Get time unit, such as year, month. */
    /* If error, handle it. */
    /* Add number of time units to appropriate member of datetm .         */
    token = strtok(NULL, delim);   /* Try to get the next number.         */
}
```

The function tries to get the first token before entering the loop. If successful, it enters the loop and calls `get_int()` to extract the number from the returned token, exits for any possible errors from a failed call to `get_int()`, and calls `strtok()` to get the associated time unit. It exits if the time unit is missing, and otherwise adds the amount of time to the `datetm` structure before calling `strtok()` again. Listing 4-9 contains the complete function implementation, with some comments omitted to save space.

*parse_time_adjustment()*

```
int parse_time_adjustment( char* time_adjust_string, struct tm* datetm )
{
    char* delim   = " \t";     /* Space and tab             */
    char* token;               /* Returned token            */
    int   number;              /* To store number token     */
    char  err_msg[STRING_MAX]; /* For error messages        */
```

```
    int   flags   = ONLY_DIGITS | NO_TRAILING;
    int   res;                /* Return value of get_int() */

    token = strtok(time_adjust_string, delim);
    while ( token != NULL ) {
        res =  get_int(token, flags, &number, err_msg ); /* Get an integer. */
        if ( VALID_NUMBER != res )
            fatal_error(res, err_msg);
        /* Number is quantity of time-adjustment unit to be found next
           get next token in time adjustment, should be a time unit. */
        token = strtok(NULL, delim);
        if ( NULL == token )
             /* End of string encountered without the time unit. */
            fatal_error(TIME_ADJUST_ERROR, "missing a time unit\n");

        if ( NULL != strstr(token, "year"))         datetm->tm_year += number;
        else if ( NULL != strstr(token, "month"))   datetm->tm_mon += number;
        else if ( NULL != strstr(token, "week"))    datetm->tm_mday += 7*number;
        else if ( NULL != strstr(token, "day"))     datetm->tm_mday += number;
        else if ( NULL != strstr(token, "hour"))    datetm->tm_hour += number;
        else if ( NULL != strstr(token, "minute"))  datetm->tm_min += number;
        else if ( NULL != strstr(token, "second"))  datetm->tm_sec += number;
        else
            fatal_error(TIME_ADJUST_ERROR,
             "Found invalid time time_unit in amount to adjust the time\n");
        token = strtok(NULL, delim);
    }
    return 0;
}
```

*Listing 4-9: The `parse_time_adjustment()` function*

To add the time adjustment to the `datetm` structure, I use the `strstr()` function, also described in that `time` man page. This is essentially a substring searching function. Its man page shows the prototype:

```
#include <string.h>
char *strstr(const char *haystack, const char *needle);
```

As the parameter names suggest, it searches for the first occurrence of substring `needle` in string `haystack`, returning a pointer to that occurrence or `NULL` if it's not there.

You might wonder why I use a sequence of cascading `if` statements in `parse_time_adjustment()` instead of a `switch` statement. In C, the `switch` statement requires an integer type, but I need to compare strings, which are not an integer type. There are more efficient ways to do this, but since this code is executed only relatively few times, and since it's clear and simple, it's suitable.

The last function we'll use is one that adds the values from one broken-down time structure into another, which I name `adjust_time()`. It's displayed in Listing 4-10.

*adjust_time()*

```
int adjust_time( struct tm* datetm, struct tm* time_to_add )
{
    datetm->tm_year += time_to_add->tm_year;
    datetm->tm_mon  += time_to_add->tm_mon;
    datetm->tm_mday += time_to_add->tm_mday;
    datetm->tm_hour += time_to_add->tm_hour;
    datetm->tm_min  += time_to_add->tm_min;
    datetm->tm_sec  += time_to_add->tm_sec;

    errno = 0;
    mktime(datetm);
    if ( errno != 0 )
        fatal_error(errno, NULL);
    return 0;
}
```

*Listing 4-10: A function that adds time amounts to a broken down time structure and normalizes the fields*

This function is pretty straightforward. The only point to emphasize is that it's possible for `mktime()` to fail and because of this, the function checks for an error after the call and terminates the program if something went wrong.

We display fragments of the *spl_date3.c* program in Listing 4-11, with the preceding functions omitted to save space.

*spl_date3.c*

```
#define _GNU_SOURCE               /* Needed for get_long() */
#include "common_hdrs.h"

#define FORMAT  "%c"              /* Default format string */
#define MAXLEN  STRING_MAX        /* Maximum size of message string */
#define BAD_FORMAT_ERROR    -1    /* In case user supplied bad format */
#define TIME_ADJUST_ERROR   -2    /* Error to return if parsing problem */

--snip--
    definitions of parse_time_adjustment and adjust_time are not shown
       in order to save space

int  main(int argc, char *argv[])
{
    char       formatted_date[MAXLEN]; /* String storing formatted date    */
    time_t     current_time;           /* Timeval in seconds since Epoch   */
    struct tm *bdtime;                 /* Broken-down time                 */
    struct tm  time_adjustment= {0};   /* Broken-down time for adjustment  */
    char       format_string[MAXLEN];  /* String storing  format spec     */
    char       usage_msg[512];         /* Usage message                    */
```

```
char       ch;                    /* For option handling              */
char       options[] = ":d:h";    /* Getopt string                    */
BOOL       d_option = FALSE;      /* Flag to indicate -d found        */
char       *d_arg;                /* Dynamic string for -d argument   */
int        d_arg_length;          /* Length of -d argument string     */

opterr = 0;  /* Turn off error messages by getopt() */
while  (TRUE) {
    ch = getopt(argc, argv, options);
    if ( -1 == ch )
        break;
    switch ( ch ) {
    case 'd':                        /* Has required argument */
        d_option = TRUE;
        d_arg_length = strlen(optarg);
        d_arg = ❶ malloc(d_arg_length * sizeof (char));
        if ( NULL == d_arg )
            fatal_error(EXIT_FAILURE, "calloc could not allocate memory\n");
        strcpy(d_arg, optarg);
        break;
    case 'h':
        sprintf(usage_msg, "%s [-d <time adjustment>]"
               " [+\"format specification\"]", basename(argv[0]));
        usage_error(usage_msg);
    case '?' :
        fprintf(stderr,"Found invalid option %c\n", optopt);
        sprintf(usage_msg, "%s [-d <time adjustment>]"
               " [+\"format specification\"]", basename(argv[1]));
        usage_error(usage_msg);
    case ':' :
        fprintf(stderr,"Missing required argument to -d\n");
        sprintf(usage_msg, "%s [-d <time adjustment>]"
               " [+\"format specification\"]", basename(argv[0]));
        usage_error(usage_msg);
    }
}
if (optind < argc) {
    if ( argv[optind][0] == '+' ) /* Argument starts with + */
        strncpy(format_string, argv[optind]+1, MAXLEN-1);
    else {
        sprintf(usage_msg, "%s [-d <time adjustment>]"
               " [+\"format specification\"]\n", basename(argv[0]));
        usage_error(usage_msg);
    }
}
else  /* No argument - use default */
    strcpy(format_string, FORMAT);
```

```
    current_time = time(NULL);
    bdtime = localtime(&current_time);
    if (bdtime == NULL)
        fatal_error(EOVERFLOW, "localtime");
    if ( d_option ) {
        parse_time_adjustment(d_arg,  &time_adjustment );
        update_time(bdtime, &time_adjustment);
     ❷ free(d_arg); /* Allocated in option handling above */
    }

    if (0 == strftime(formatted_date, sizeof(formatted_date),
             format_string, bdtime) )
        fatal_error(BAD_FORMAT_ERROR, "Conversion to a date-time string "
                "failed or produced an empty string\n");
    printf("%s\n", formatted_date);
    return 0;
}
```

*Listing 4-11: The main program of the third version of spl_date, with some code omitted*

This listing introduces the C `malloc()` function ❶, which is part of a family of memory allocation functions including `calloc()` and `realloc()`. The `malloc()` function allocates memory from the heap and returns a pointer to the start of the newly allocated memory. Its prototype is

```
#include <stdlib.h>
void *malloc(size_t size);
```

Because it returns a `void*` result, we can assign that address to any C pointer, such as a `utlist*` or a `char*`. Although unlikely, it can fail because there's no memory left to allocate, and will return `NULL` and set `errno` to `ENOMEM` in this case. The listing also introduces `free()` ❷, which is used to free the memory space pointed to by ptr, which must have been returned by a previous call to `malloc()`, `calloc()`, or `realloc()`. Its synopsis is

```
#include <stdlib.h>
void free(void* ptr);
```

If the memory pointed to by `ptr` has been freed already, the consequences are unpredictable. We'll discuss allocating and deallocating memory more in Chapter 11. Note that the absence of a `break` after each call to `usage_error()` in the `switch` statement is justified because the function terminates the program. A few runs demonstrate the program's behavior.

```
$ spl_date3 -h
usage: spl_date3 [-d <time adjustment>] [+"format specification"]
$ spl_date3 +%a %b %d, %Y, at %R
Wed Feb 22, 2023, at 10:52
$ spl_date3 -d 1 year +%a %b %d, %Y, at %R
```

```
Thu Feb 22, 2024, at 10:52
$ spl_date3 -d 1 week 2 hours +%a %b %d, %Y, at %R
Wed Mar 01, 2023, at 12:52
$ spl_date3 -d -2 months   +4 months +%a %b %d, %Y, at %R
Sat Apr 22, 2023, at 11:52
$ spl_date3 -d  +120 minutes -2 hours  +%a %b %d, %Y, at %R
Wed Feb 22, 2023, at 10:52
```

Notice that subtracting two months and adding four months results in a net of two months later than the current time and that adding 120 minutes and subtracting two hours leaves the time unchanged.

## Working with Locales

Imagine now that the user of our `spl_date3` program is someone from another region of the world who doesn't speak English and doesn't use our representation of dates and times. As it's written so far, its output won't be in a form that they can understand. How can we change it so that it is?

This question leads into a deeper study of the internationalization of software and the concept of a locale. The POSIX.1-2017 definition of a locale given in Chapter 3, is "the definition of the subset of a user's environment that depends on language and cultural conventions," which is very abstract. We need to know how to program with them and what they are in more concrete terms.

In particular, we need to know the following:

- What exactly is a locale?
- What conventions does a locale influence?
- What kinds of information does a locale encapsulate?
- How many different locales are there, and where are they stored?
- Is there some standard, default locale?
- At the user level, what commands can we use to view and change our locale?
- How is the information associated with a locale structured?
- At the programming level, what library functions get and set locale contents so that programs can be internationalized?

The first step is to search the man pages for answers. Searching for the keyword *locale* using **`apropos locale`** results in a long list of man pages, many of which are in Section 3, Library Functions. We'd like some basic information, which would be in a Section 7 page, perhaps a Section 5 page, or in the POSIX *locale.h* header file page. We'd also like to know what user-level commands display information about locales, and these would be in Section 1. We start with the general description page in Section 7:

```
$ man 7 locale
LOCALE(7)                 Linux Programmer's Manual                 LOCALE(7)
```

```
NAME
       locale - description of multilanguage support

SYNOPSIS
       #include <locale.h>

DESCRIPTION
       A locale is a set of language and cultural rules. These cover aspects
       such as language for messages, different character sets,  lexicographic
       conventions,  and  so  on. A program needs to be able to determine its
       locale and act accordingly to be portable to different cultures.

       The header <locale.h> declares data types, functions and  macros  which
       are useful in this task.

       The functions it declares are setlocale(3) to set the current locale,
       and localeconv(3) to get information about number formatting.
--snip--
```

This page refers us to the *locale.h* header file for details about the data types, functions, and macros. It also mentions two functions, `setlocale()` and `localeconv()` that we may need in our modified program. The rest of the man page describes important, fundamental concepts, summarized below.

A locale consists of a collection of categories. *Categories* are parts of the locale that control related aspects of a user's cultural and language settings. For example, the `LC_CTYPE` category consists of data that specify character classification, case conversion, and other character attributes, such as which characters are letters, which are digits, which are punctuation, and so on.

The names that identify categories all begin with `LC_` (short for *locale category*). These names are both integer-valued macros declared in *locale.h* for use by programs and environment variables. Thus, `LC_CTYPE` is both the macro name of a category and the name of an environment variable.

POSIX.1-2017 defines six categories, all of which should be in the environment of most Unix systems that you might use. The GNU C library, starting with version *glibc 2.2*, extends the set with six more categories. Table 4-1 contains all of the categories present in the latest GNU/Linux distribution, as of this writing, with an indication of whether it is part of POSIX or a GNU extension, and a brief synopsis of what it controls.

These 12 categories cover a broad spectrum of information. A few other environment variables, shown in Table 4-2, control the locale in addition to the categories listed in Table 4-1.

The variables in Table 4-1 are not locale categories, but with the exception of `TZ`, they're used for managing locale information. For example, `LC_ALL` acts like a global locale setting, overriding the values of all locale variables; setting it to a specific locale assigns that locale to all of the categories, whether or not they were set to a specific locale.

**Table 4-1:** Locale Categories and Their Meanings

| Category | Availability | Meaning |
|---|---|---|
| LC_COLLATE | POSIX.1-2017 | Collation order (how characters are sorted) |
| LC_CTYPE | POSIX.1-2017 | Character classification and case conversion, such as which characters are in the character set and what are their classes |
| LC_MESSAGES | POSIX.1-2017 | Formats of informative and diagnostic messages and interactive responses |
| LC_MONETARY | POSIX.1-2017 | Monetary formatting, such as currency symbols and conventions |
| LC_NUMERIC | POSIX.1-2017 | Numeric, non-monetary formatting |
| LC_TIME | POSIX.1-2017 | Date and time formats |
| LC_ADDRESS | GLIBC-2.2 | Formats of locations and geography-related items, such as names of places |
| LC_IDENTIFICATION | GLIBC-2.2 | Metadata for the locale |
| LC_MEASUREMENT | GLIBC-2.2 | Measurement systems (for example, metric vs. US customary units) |
| LC_NAME | GLIBC-2.2 | Words used to address people (for example, "Frau", "Mme.") |
| LC_PAPER | GLIBC-2.2 | Dimensions of standard paper sizes (for example, US letter vs. A4) |
| LC_TELEPHONE | GLIBC-2.2 | Formats used with telephone services |

The Section 7 man page also describes how to pass locale data to the `setlocale()` function and shows the declaration of the `lconv` struct that `localeconv()` returns. Although we'll eventually need to learn about these two functions, we'll visit them later in this chapter, in "Programming with Locales". Before we explore how to manage locales at the user level, we need to understand a bit about time zones.

## *About Time Zones*

The `TZ` environment variable listed in Table 4-1 stores the time zone associated with the current user, which is not necessarily the same as the system time zone. For example, a large institution such as a university or a corporation might have a Linux server that people can log into from around the world. The server lives in its own time zone. Individual users can be in different time zones, and they can set their `TZ` variable to their own time zones, usually in their shell configuration files, such as *~/.bashrc*. This way the programs that they run will use their time, not the time of the server.

You assign a string representation of the time zone to the `TZ` variable. For example, `":America/New York"` is my time zone. To understand what values you can assign to it, you need to know how time zones are managed in Unix.

Time zone information is stored in a standardized binary format, defined by a standard named *Internet RFC 8536*. This format also includes information about Daylight Savings Time. The individual time zone files that contain this data are in the directory */usr/share/zoneinfo*. If you look at its contents, you'll see that most of the files there are directories, but some are plain files, and some are symbolic links. For example, it contains a direc-

**Table 4-2:** Environment Variables That Affect the Locale Settings

| Variable | Availability | Meaning |
|---|---|---|
| LC_ALL | POSIX.1-2017 | Represents the set of all locale categories and has special meaning when and precedence |
| LANG | POSIX.1-2017 | Determines the locale category for native language, local customs, and coded character set in the absence of the LC_ALL and other LC_* variables |
| LANGUAGE | GLIBC-2.2 | Used by the glibc function gettext in language translation |
| TZ | POSIX.1-2017 | Timezone information |
| NLSPATH | POSIX.1-2017 | A path variable (same format as PATH) used for finding message catalogs for translation to other languages |
| LOCPATH | POSIX.1-2017 | A path variable for finding locale data files |

tory named *America* and a directory named *Europe*, as well as a symbolic link named *Greenwich*. Each directory has ordinary files with the names of cities or regions. Each of these can be specified as a time zone, using the pathname starting in */usr/share/zoneinfo*. For example, under *Europe* there's a file named *Paris* and another named *Dublin*. All of the following are valid assignments:

```
TZ=":Europe/Paris"
TZ=":Europe/Dublin"
TZ=":Greenwich"
```

There is another, more complex, way to assign time zone information to the TZ variable, but I don't describe it here. See the POSIX.1-2017 standard for more details.

We won't delve into the form of the system time zone files either. Fortunately for us, the C Library does all of the work to take time zone information into account. Those functions that return times and dates, other than those that specifically ignore time zones, such as gmtime(), all behave correctly without our needing to do anything special.

### The Command-Level Interface to Locales

The search for man pages with which we started, apropos locale, returned references to two pages describing the locale command. One is in Section 1, and the other in Section 1POSIX:

```
locale (1)          - get locale-specific information
locale (1posix)     - get locale-specific information
```

The POSIX page is a specification of what the command should do; the other page describes the command implemented on the system you're using. Both are useful, but let's see what the first page tells us:

```
$ man 1 locale
LOCALE(1)                   Linux User Manual                   LOCALE(1)
```

```
NAME
       locale - get locale-specific information

SYNOPSIS
       locale [option]
       locale [option] -a
       locale [option] -m
       locale [option] name...

DESCRIPTION
       The locale command  displays information about the current locale, or
       all locales, on standard output.

       When invoked without arguments, locale displays the current locale set
       tings for each locale category (see locale(5)), based on the settings
       of the environment variables that control the locale (see  locale(7)).
       Values for variables set in the environment are printed without double
       quotes, implied values are printed with double quotes.
--snip--
```

When invoked without arguments, `locale` displays the current locale settings for each locale category.

Let's run it without arguments to see what it outputs. What you see when you run it is probably different from what I see:

```
$ locale
LANG=en_US.UTF-8
LANGUAGE=en_US
LC_CTYPE=en_US.UTF-8
LC_NUMERIC=en_US.UTF-8
LC_TIME=en_US.UTF-8
LC_COLLATE=C.UTF-8
LC_MONETARY=en_US.UTF-8
LC_MESSAGES="en_US.UTF-8"
LC_PAPER=en_US.UTF-8
LC_NAME=en_US.UTF-8
LC_ADDRESS=en_US.UTF-8
LC_TELEPHONE=en_US.UTF-8
LC_MEASUREMENT=en_US.UTF-8
LC_IDENTIFICATION=en_US.UTF-8
LC_ALL=
```

You can see that, on my system the locale is set to be `en_US.UTF-8` for all but one category, `LC_COLLATE`, which is set to `C.UTF-8`. The `LANG` is `en_US.UTF-8` as well. The `LANGUAGE` variable has the same value, but the name is the short form of it. The `LC_ALL` variable is assigned an empty string because if it were assigned a non-empty string, it would override the values for all other cate-

gories, which would prevent me, for example, from changing one category to a different value from the others.

Locale names are typically of the form

```
language[territory][.codeset][@modifier]
```

where *language* is an ISO 639 language code, *territory* is an ISO 3166 country code, *codeset* is a character set or encoding identifier such as ISO-8859-1 or UTF-8, and *modifier* is any string used to further refine the name.

In the locale name `en_US.UTF-8`, *en* is the English language, *US* is the United States, and *UTF-8* is the codeset. It has no modifier.

A *codeset* is a mapping from graphical characters to numeric values. The numeric values are called *code points*, and codesets are also sometimes called *character maps* or *character sets*. For example, `ASCII` is an early codeset that maps the set of characters commonly found on old keyboards, as well as certain other nonprinting characters, to 7-bit unsigned integers. It does not map characters with diacritical marks or non-Latin characters. The `UTF-8` codeset is a variable-length codeset that is capable of representing all Unicode code points in anywhere from one to four bytes per point.

*Unicode* is a numeric representation of the alphabets of almost all known ancient and modern languages, as diverse as Japanese, Chinese, Greek, Cyrillic, Canadian Aboriginal, and Arabic. Appendix E contains a brief history and description of Unicode with detailed examples.

The `locale` command with the `-a` option outputs a list of the available locales on your system. This is a fragment of the output on my system, for example:

```
$ locale -a
C
C.utf8
POSIX
en_AG.utf8
en_AU.utf8
--snip--
fi_FI.utf8
fr_BE.utf8
fr_FR.utf8
he_IL.utf8
it_IT.utf8
pl_PL.utf8
```

Except for the first three lines, these are the names of locales I've *generated* for my use, either temporarily to run some program under them, or permanently as a locale in which I want to work. You can ignore the fact that the `utf8` suffix is in lowercase and doesn't have the hyphen; codeset names are case-insensitive, and the hyphen is optional. The first three, `C`, `C.utf8`, and `POSIX`, are predefined locales. POSIX.1-2017 requires systems to have a `POSIX` locale and for it to be the default locale for all C programs. The `C` locale is

the same as the POSIX locale if a system conforms to POSIX.1-2017, but if not, the latter is usually more extensively defined.

From the names of the locales listed in the previous example, you might be able to infer what they are, but if you add the -v option, you'll get much more detail. Below are two examples of the details that you'd see:

```
$ locale -av
--snip--

locale: en_IE.utf8      archive: /usr/lib/locale/locale-archive
-----------------------------------------------------------------------------
    title | English locale for Ireland
   source | RAP
  address | Sankt Jørgens Alle 8, DK-1615 København V, Danmark
    email | bug-glibc-locales@gnu.org
 language | English
territory | Ireland
 revision | 1.0
     date | 2000-06-29
  codeset | UTF-8

  --snip--

  locale: pl_PL.utf8      archive: /usr/lib/locale/locale-archive
-----------------------------------------------------------------------------
    title | Polish locale for Poland
   source | RAP
  address | Sankt Jo/rgens Alle 8, DK-1615 Ko/benhavn V, Danmark
    email | bug-glibc-locales@gnu.org
 language | Polish
territory | Poland
 revision | 1.0
     date | 2000-06-29
  codeset | UTF-8
```

You can change your locale to any of the ones listed by this command by assigning the environment variables their full names. For example, if I change the LC_ALL variable to pl_PL.utf8, all of those functions and commands that are sensitive to the locale will use the Polish settings for my locale.

### TEMPORARILY CHANGING THE ENVIRONMENT

In bash, you can precede a command with one or more variable assignments. If these variables are environment variables, the change in their value will be in effect only for the execution of that individual command, because a temporary environment is created with those changes and passed to a subshell in which the command is run. To demonstrate, I'll run `date +"%c"` first and then set the time

zone variable `TZ` to be the current time in Spain and override all other category settings using the territorial locale for Spain, `es_ES.utf-8`, and run `date +``%c''` again, so you can see the difference:

```
$ date +"%c"
Mon 06 Mar 2023 01:11:45 PM EST
$ TZ=Spain LC_ALL=es_ES.utf-8 date +"%c"
lun 06 mar 2023 18:11:47
```

The day `lun` is short for *Lunes*, the Spanish word for *Monday* and `mar` is short for *marzo*, the word for *March*.

The locales listed by `locale -a` are a small subset of those that you can generate. In Linux, the file */etc/locale.gen* contains a list of locales that you can generate by uncommenting them and rerunning the `locale-gen` command, provided that you have superuser privilege. After you do that, the locale's name will be in the list displayed by `locale -a`. The */etc/locale.gen* file typically contains several hundred locale names, mostly commented out. Linux maintains a list of all supported locales in the file */usr/share/i18n/SUPPORTED*. The exact path might vary depending on the particular Linux distribution that you're using. The directory name *i18n* in this path is the abbreviation that people use for *internationalization*—that word has 18 letters starting with *i* and ending with *n*. That file usually has about as many entries as *etc/locale.gen*.

### The Structure of Locales

The information associated with each given locale category is represented in a precise, structured format, specified by POSIX.1-2017. This makes it possible to write functions that use locale data and to create new locales for different regions. Since Section 5 of the man pages generally contains descriptions of file formats and definitions of data structures used in system and library interfaces, it's a good place to look for information about the format and structure of locale data.

The locale man page in Section 5 describes the form and data of a *locale definition file*. Locale definitions are written in a markup language that resembles Extensible Markup Language (XML). They are given as input to the `localedef` command, which generates a compressed binary file with the same data. Programs that use locales read the binary data, not the locale definition files.

Different locale categories have different data, but their definition files all have the same form. Each is defined by its own set of *keywords* and associated values. The value for a keyword depends upon the keyword. To illustrate, I've included part of the actual definition file for the `LC_NUMERIC` category used by the `en_US.utf8` locale. It's the smallest locale category:

```
LC_NUMERIC

decimal_point     "<period>"
```

```
thousands_sep     "<comma>"
grouping          "3;0"
--snip--
END LC_NUMERIC
```

Every file begins with the name of the category and ends with `END` *`category-name`*. This category has three keywords: `decimal_point`, `thousands_sep`, and `grouping`. The first two values are self-explanatory. The value for `grouping` indicates that groups of three digits are separated by commas for all groups to the left of the decimal point. The first digit (3) is the size of the first group to the left of the decimal point. The second, 0, means that all groups to the left have 3 as well.

The `LC_CTYPE` category has much more extensive data. So that you can see how their definitions can vary, here's part of a typical definition file for the `en_US.utf8` locale:

```
escape_char   /
LC_CTYPE

upper   <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;/
        <N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>

lower   <a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;/
        <n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>

space   <tab>;<newline>;<vertical-tab>;<form-feed>;/
        <carriage-return>;<space>

cntrl   <alert>;<backspace>;<tab>;<newline>;<vertical-tab>;/
        <form-feed>;<carriage-return>;<NUL>;<SOH>;<STX>;/
        <ETX>;<SEL>;<RNL>;<DEL>;<GE>;<SPS>;<RPT>;<SI>;<SO>;<DLE>;<DC1>;/
        <DC2>;<DC3>;<RES>;<POC>;<CAN>;<EM>;<UBS>;<CU1>;<IFS>;/
        <IGS>;<IRS>;<ITB>;<DS>;<SOS>;<fs>;<WUS>;<BYP>;<LF>;/
        <ETB>;<ESC>;<SA>;<SM>;<CSP>;<MFA>;<ENQ>;<ACK>;/
        <SYN>;<IR>;<PP>;<TRN>;<NBS>;<EOT>;<SBS>;<IT>;<RFF>;/
        <CU3>;<DC4>;<NAK>;<SUB>

punct   <exclamation-mark>;<quotation-mark>;<number-sign>;<dollar-sign>;/
        <percent-sign>;<ampersand>;<apostrophe>;<left-parenthesis>;/
        <right-parenthesis>;<asterisk>;<plus-sign>;<comma>;/
        <hyphen-minus>;<period>;<slash>;<colon>;<semicolon>;/
        <less-than-sign>;<equals-sign>;<greater-than-sign>;/
        <question-mark>;<commercial-at>;<left-square-bracket>;/
        <backslash>;<right-square-bracket>;<circumflex>;/
        <underscore>;<grave-accent>;<left-curly-bracket>;/
        <vertical-line>;<right-curly-bracket>;<tilde>

digit   <zero>;<one>;<two>;<three>;<four>;/
```

```
        <five>;<six>;<seven>;<eight>;<nine>

xdigit  <zero>;<one>;<two>;<three>;<four>;/
        <five>;<six>;<seven>;<eight>;<nine>;/
        <A>;<B>;<C>;<D>;<E>;<F>;/
        <a>;<b>;<c>;<d>;<e>;<f>

blank   <space>;<tab>

toupper (<a>,<A>);(<b>,<B>);(<c>,<C>);(<d>,<D>);(<e>,<E>);/
        (<f>,<F>);(<g>,<G>);(<h>,<H>);(<i>,<I>);(<j>,<J>);/
        (<k>,<K>);(<l>,<L>);(<m>,<M>);(<n>,<N>);(<o>,<O>);/
        (<p>,<P>);(<q>,<Q>);(<r>,<R>);(<s>,<S>);(<t>,<T>);/
        (<u>,<U>);(<v>,<V>);(<w>,<W>);(<x>,<X>);(<y>,<Y>);(<z>,<Z>)

tolower (<A>,<a>);(<B>,<b>);(<C>,<c>);(<D>,<d>);(<E>,<e>);/
        (<F>,<f>);(<G>,<g>);(<H>,<h>);(<I>,<i>);(<J>,<j>);/
        (<K>,<k>);(<L>,<l>);(<M>,<m>);(<N>,<n>);(<O>,<o>);/
        (<P>,<p>);(<Q>,<q>);(<R>,<r>);(<S>,<s>);(<T>,<t>);/
        (<U>,<u>);(<V>,<v>);(<W>,<w>);(<X>,<x>);(<Y>,<y>);(<Z>,<z>)
--snip--
END LC_CTYPE
```

*Listing 4-12: A locale definition file for the English language in the United States*

Notice the syntax that's used for defining the keyword values in this category. The `toupper` keyword provides the data that functions would need to convert lowercase to uppercase, so it's a semicolon-separated sequence of pairs that essentially defines a function from characters to characters. In contrast, the `xdigit` keyword's value is just a list of the names of the hexadecimal digits that we use in the United States.

If you want to know what the keywords and values are for a locale category, you could read the documentation, but fortunately, the `locale -k` command will list them. Give it the name of the category, and it outputs a list:

```
$ locale -k LC_TIME
abday="Sun;Mon;Tue;Wed;Thu;Fri;Sat"
day="Sunday;Monday;Tuesday;Wednesday;Thursday;Friday;Saturday"
abmon="Jan;Feb;Mar;Apr;May;Jun;Jul;Aug;Sep;Oct;Nov;Dec"
mon="January;February;March;April;May;June;July;August;September;October;
November;December"
am_pm="AM;PM"
d_t_fmt="%a %d %b %Y %r %Z"
d_fmt="%m/%d/%Y"
t_fmt="%r"
t_fmt_ampm="%I:%M:%S %p"

--snip--
```

You can also give it a keyword. To see the format used by `date`, enter the following:

```
$ locale -ck date_fmt
LC_TIME
date_fmt="%a %b %e %r %Z %Y"
```

The `-c` option prints the locale category, in this case `LC_TIME`, on a separate line. With the `-k` *keyword* option, `locale` prints the supplied keyword and its value, in this case `date_fmt` and its value, `%a %b %e %r %Z %Y`. Consulting Table C-1 we can verify that this is what `date` prints, but we can also enter that command to double-check:

```
$ date +"%a %b %e %r %Z %Y"
Wed Jan 25 12:11:00 PM EST 2023
$ date
Wed Jan 25 12:11:01 PM EST 2023
```

You can see that `date` with no format outputs exactly the same fields as the format string "`%a %b %e %r %Z %Y`".

### *Programming with Locales*

We started this exploration of locales partly so that we could internationalize the `spl_date` program. When we read the `locale(7)` man page, it mentioned the `setlocale()` function. Let's see what its man page says about it:

```
$ man setlocale
SETLOCALE(3)               Linux Programmer's Manual              SETLOCALE(3)

NAME
       setlocale - set the current locale

SYNOPSIS
       #include <locale.h>
       char *setlocale(int category, const char *locale);

DESCRIPTION
       The setlocale() function is used to set or query the program's current
       locale.
--snip--
```

This is a library function for setting a program's locale, as well as finding out what locale is in effect for it. Its first parameter is the category name. If the second parameter is `NULL`, it doesn't change the locale, but returns the name of the locale currently assigned to the passed-in category. If the second parameter is the full name of locale, such as `"en_US.UTF-8"`, it will set the category's value to that locale; otherwise, it returns `NULL`. If the second parameter is an empty string (`""`), `setlocale()` will set the category's value to the locale

setting it finds in the current environment. The rule it uses depends on the version of Unix that you're running. In GNU/Linux, the steps for deciding which locale to assign to the category in the first parameter when the second is an empty string are:

1. If there is a non-null environment variable `LC_ALL`, the value of `LC_ALL` is used.
2. If an environment variable with the same name as the category exists and is non-null, its value is used.
3. If there is a non-null environment variable `LANG`, the value of `LANG` is used.

A program must call `setlocale()` in order for it to be internationalized. In the absence of a call to this function, the program will use the "C" locale. If it calls `setlocale(LC_ALL, "")`, it will assign to all categories the value of the locale it determined from the steps above. If the program uses library functions whose behavior is dependent on the locale, they will use the values determined by `setlocale()`. Therefore, calling `setlocale()` is the first step in internationalizing your programs. For `spl_date`, it's the only step we need to take.

### *An Internationalized Version of the showdate Program*

We insert a call to `setlocale()` into `main()` before calls to any other library functions, right before the option-handling code. The man page for `strftime()` states that the only environment variables that it uses are `TZ` and `LC_TIME`. In other words, it uses the time zone setting in `TZ`, and it uses the values of the keywords in the `LC_TIME` category to format the time, based on the format specification that we pass to it.

We don't need to do anything for our program to report the correct time for the user's time zone, because we assume that when the user set up their account, they supplied their time zone, which was stored in the `TZ` environment variable. If not, the time zone defaults to the system's time zone, which is usually stored in */etc/timezone*.

Therefore, we'll call `setlocale()`, passing `LC_TIME` as its first argument rather than `LC_ALL`. We could pass it `LC_ALL` if we thought it might influence the behavior of other functions in our program, but in this case it isn't necessary:

```
if ( setlocale(LC_TIME, "") == NULL )
    fatal_error( LOCALE_ERROR,
       "setlocale() could not set the given locale");
```

The program doesn't save the return value, but it checks whether it's `NULL`, which is returned if the locale couldn't be set. If we want to save the name of the locale for later use, we copy it into a local string variable.

The structure of the revised program, with most of it omitted to save space, is shown in Listing 4-13. The complete program is in the source code distribution for the book.

*spl_date4.c*

```
#define _GNU_SOURCE
#include <locale.h>
#include "common_hdrs.h"

#define FORMAT  "%c"              /* Default format string */
#define MAXLEN  STRING_MAX        /* Maximum size of message string */
#define BAD_FORMAT_ERROR    -1    /* Error to return if format is malformed */
#define TIME_ADJUST_ERROR   -2    /* Error to return if parsing problem  */
#define LOCALE_ERROR        -3

--snip-- (definitions of parse_time_adjustment and adjust_time omitted
          to save space)

int  main(int argc, char *argv[])
{
    --snip-- (variable declarations omitted)

    if ( setlocale(LC_TIME, "")  == NULL )
        fatal_error( LOCALE_ERROR, "setlocale() could not set given locale");

    --snip-- (option-parsing code omitted)

    if (optind < argc) {  /* use supplied format specification  */
        if ( argv[optind][0] == '+' )  /* argument starts with + */
            strncpy(format_string, argv[optind]+1, MAXLEN-1);
        else {
            sprintf(usage_msg, "%s [-d <time adjustment>]"
                   " [+\"format specification\"]\n", basename(argv[0]));
            usage_error(usage_msg);
        }
    }
    else  /* no argument - use default */
        strcpy(format_string,  FORMAT);
    current_time = time(NULL);
    if ( (bdtime = localtime(&current_time)) == NULL ) )
        fatal_error(EOVERFLOW, "localtime");

    if ( d_option ) {
        parse_time_adjustment(d_arg,  &time_adjustment );
        update_time(bdtime, &time_adjustment);
        free(d_arg); /* allocated in option handling above */
    }

    if (0 == strftime(formatted_date, sizeof(formatted_date),
```

```
            format_string, bdtime) )
        fatal_error(BAD_FORMAT_ERROR, "Conversion to a date-time "
                    "string failed or produced an empty string\n");
    printf("%s\n", formatted_date);
    return 0;
}
```

*Listing 4-13: The internationalized spl_date program, with most code omitted*

Let's see how this program behaves. We'll run it under several different locales, leaving the time zone unchanged, and with both the default format and a custom format:

```
$ LC_TIME=da_DK.utf8 spl_date4
lør 11 mar 2023 14:22:39 EST
$ LC_TIME=da_DK.utf8 spl_date4 +%A, %d %B %Y
lørdag, 11 marts 2023
$ LC_TIME=de_DE.utf8 spl_date4 +%A, %d %B %Y
Samstag, 11 März 2023
$ LC_TIME=es_ES.utf8 spl_date4 +%A, %d %B %Y
sábado, 11 marzo 2023
$ LC_TIME=fi_FI.utf8 spl_date4
la 11. maaliskuuta 2023 14.22.39
$ LC_TIME=fi_FI.utf8 spl_date4 +%A, %d %B %Y
lauantai, 11 maaliskuu 2023
$ LC_TIME=fr_FR.utf8 spl_date4
sam. 11 mars 2023 14:22:39
$ LC_TIME=ja_JP.utf8 spl_date4
2023年03月11日 14時22分39秒
$ LC_TIME=ja_JP.utf8 spl_date4 +%A, %d %B %Y
土曜日, 11 3月 2023
```

This final version of `spl_date` is able to display dates and times following the conventions of a wide range of geographic locales. In the end, enabling this feature required only a small modification to the previous program, but understanding why and how it works was the real goal. Now, we'll turn our attention to other aspects of internationalization.

### Other Ways to Internationalize Programs

The System Interfaces section of the POSIX.1-2017 standard specifies which functions in the C library should take locale information into account and which parts of the locale they should use. In general, a library implementation in a Unix system may or may not conform to these requirements. For the most part, the GNU C Library in Linux meets the standard's requirements and goes beyond them by providing some features not specified in POSIX.1-2017. Here we limit discussion to the GNU C library's internationalization features.

The underlying philosophy of the GNU C library is that the programmer should be freed as much as possible from the burden of handling internationalization. If a program sets its locale using `setlocale()` or one of a few other similar functions I haven't mentioned yet, before calling any library functions, all of the functions that are designed to use locale data will modify their behavior according to the locale's rules.

This reduces our problem to knowing which functions use locale information and which locale categories they use. Unfortunately, the documentation doesn't contain a complete list of precisely those library functions that use locale information, so I'll provide some guidance that overcomes this deficiency. Following is a list of functions that do use locale data:

| | | | | | |
|---|---|---|---|---|---|
| fprintf | islower | iswcntrl | iswupper | strcoll | toupper |
| fscanf | isprint | iswctype | iswxdigit | strerror | towlower |
| isalnum | ispunct | iswdigit | isxdigit | strfmon | towupper |
| isalpha | isspace | iswgraph | mblen | strftime | wcscoll |
| isblank | isupper | iswlower | mbstowcs | strsignal | wcstod |
| iscntrl | iswalnum | iswprint | mbtowc | strtod | wcstombs |
| isdigit | iswalpha | iswpunct | perror | strxfrm | wcsxfrm |

Most of these use data from either the `LC_CTYPE` or `LC_COLLATE` categories, but some also use `LC_NUMERIC`, `LC_TIME`, or `LC_MONETARY`. Their man pages specify which of these categories the function uses, either by naming which locale-specific environment variables it uses, or by stating that the function uses the locale in a specific way. You can search for the keyword *locale* or the pattern `LC_` in the page using the pager's search operator `/` followed by the keyword, as in **`/LC_`** to jump to the part of the page that references these terms.

If this list isn't accessible, and you can't remember which functions use the locale, you can refer to the `SEE ALSO` section of the Info page for `setlocale()`, or visit the POSIX.1-2017 website page for it, at *https://pubs.opengroup.org/onlinepubs/9699919799/functions/setlocale.html* where many of the functions are listed.

The `strcoll()` function is worth singling out. Here's its prototype:

```
int strcoll(const char *s1, const char *s2);
```

It compares two strings, `s1` and `s2`, and returns a negative integer if `s1 < s2`, zero if `s1 == s2`, and a positive integer if `s1 > s2`. Most people use `strcmp()` for comparing two strings in C. Its prototype is the same, but `strcmp()` doesn't use locale data in its comparisons, which means that sorting algorithms based on `strcmp()` won't sort according to the true ordering of characters in the user's locale. In contrast, `strcoll()` does use the locale's `LC_COLLATE` data. The following program demonstrates its use:

*strcoll_demo.c*

```
#define _GNU_SOURCE
#include "common_hdrs.h" /* Includes <locale.h>. */

int main( int argc, char* argv[])
{
    char *smallest;
    char usage_msg[256];
```

```
    int  i = 1, j;

    if ( argc < 3 ) {
        sprintf(usage_msg, "%s string string ...\n", basename(argv[0]));
        usage_error(usage_msg);
    }
    if ( NULL == setlocale(LC_COLLATE, "") )
        fatal_error( LOCALE_ERROR,
             "setlocale() could not set the given locale");
    smallest = argv[i];
    for ( j = i+1; j < argc; j++)
        if ( strcoll(smallest,argv[j]) > 0 )
            smallest = argv[j];
    printf("%s\n",smallest);
    return 0;
}
```

If we compile and run this program, setting a different temporary locale for each run, we see how it behaves:

```
$ LC_COLLATE=C strcoll_demo  Zebra lion camel ape
Zebra
$ LC_COLLATE=en_US.utf8 strcoll_demo  Zebra lion camel ape
ape
```

The C locale uses the ASCII ordering of characters, with all uppercase preceding all lowercase. In contrast, the en_US.utf8 locale sorting order is case insensitive. If, in *strcoll_demo.c*, we replaced `strcoll()` with `strcmp()` and ran this program, in both locales the output would be `Zebra`, showing that `strcmp()` doesn't use locale data.

Sometimes no library function can handle the problem you're trying to solve in a locale-sensitive way. In that case, you need to access locale data directly. The library has ways to do this. When we first searched for functions to internationalize our `spl_date` program by entering **`apropos locale`**, the output listed a few library functions that we overlooked. We'll search again but limit the search to Section 3:

```
$ apropos -s3 locale
--snip--
localeconv (3)       - get numeric formatting information
localeconv (3posix)  - return locale-specific information
--snip--
nl_langinfo (3)      - query language and locale information
nl_langinfo_l (3)    - query language and locale information
--snip--
```

(I removed lines that aren't relevant.) The `localeconv()` and `nl_langinfo()` functions can each be used for obtaining information about the values of keywords in the current locale of the calling program. If you read their man

pages, you'll learn that the difference between them is that `localeconv()` returns all of the information available in the locale in one very large data structure, (`struct iconv`), whereas `nl_langinfo()` is given a keyword from a locale category and returns the value of that particular keyword. The GNU C Library Reference Manual calls `nl_langinfo()`, *pinpoint access* to the locale.

The `localeconv()` function is more portable than `nl_langinfo()`, but it's slow because it has to gather all of the locale data, it isn't extensible, and it's not general enough, since it gives access to only `LC_MONETARY` and `LC_NUMERIC` data. In contrast, `nl_langinfo()` also provides extensive access to information from the `LC_TIME` category and limited access to `LC_MESSAGES`.

Here's a simple example that uses `nl_langinfo()` to print the days of the week in the language of the current locale:

*nl_langinfo_demo.c*

```
#define _GNU_SOURCE
#include <langinfo.h>
#include "common_hdrs.h"

int  main(int argc, char *argv[])
{
    char*  mylocale;
    if ( (mylocale = setlocale(LC_TIME, "") ) == NULL )
        fatal_error( LOCALE_ERROR,
                    "setlocale() could not set the given locale");
    printf("The current locale is %s\n", mylocale);

    /* DAY_1 is a keyword defined in langinfo.h. When passed to nl_langinfo,
       the function returns the name of the first day of the week in the
       current locale. The second day is DAY_2, and so on.  Because they are
       consecutive integers, we can increment to advance through them.      */
    for ( int dayofweek = DAY_1; dayofweek < DAY_1+7; dayofweek++ )
        printf("%s\n", nl_langinfo(dayofweek));
    return 0;
}
```

This program uses the knowledge that the keywords `DAY_1`, `DAY_2`, and so on, are integers with consecutive values of an enumeration type, defined in the header file *langinfo.h*, so that we could loop through the keywords. We compile and run the program, changing the locale to see its effect:

```
$ LC_ALL=es_ES.utf8 nl_langinfo_demo
The current locale is es_ES.utf8  Spanish in Spain
domingo
lunes
martes
miércoles
jueves
viernes
sábado
```

```
$ LC_ALL=da_DK.utf8 nl_langinfo_demo
The current locale is da_DK.utf8 Danish in Denmark
søndag
mandag
tirsdag
onsdag
torsdag
fredag
lørdag
```

The `nl_langinfo()` function can access many other keywords, which makes it possible to write your own functions that are locale-sensitive, provided that they depend only on the categories of data that it's able to retrieve.

### Locale Objects

Although it's an advanced topic, I'll briefly describe the manipulation of locales. You might at some point decide that you want to create your own custom locales. A *locale object* is an object of type `locale_t`. Locale objects can be created by two functions: `newlocale()` and `duplocale()`. These functions were added to the locale interface as multithreading became more common in software. Individual threads in a process can call these functions to create locale objects independently. In a single-threaded process, there is a single locale, but when a process has multiple threads, each can have its own locale. In Chapter 13, we'll discuss how you can use `newlocale()` to create locale objects for each thread. Although they were intended to facilitate internationalization of multithreaded programs, you can call them in a single-threaded program to create new locale objects for that program.

The synopsis for `newlocale()` is as follows:

```
#include <locale.h>
locale_t newlocale(int category_mask, const char *locale, locale_t base);
```

The first parameter (`category_mask`) is a set of the locale categories you want to modify, such as `LC_TIME`. To modify more than one, you give it a bitwise-or of category names, such as `LC_TIME|LC_NUMERIC`. The second parameter (`locale`) is the string name for the locale that you want to apply to this category, such as `es_ES.utf8`. The last parameter (`base`) is the locale object that you want to modify. If `base` is the value `(locale_t) 0`, meaning the value zero typecast to type `locale_t`, then a new locale object is created, otherwise the locale object in `base` is modified.

This function allows you to process different categories of locale data in one locale and then process other data in a different locale during a computation. The program in Listing 4-14, based on the example from the `newlocale()` man page, demonstrates how to use it. It expects two locale names on the command line. It combines the `LC_NUMERIC` settings of the first one and the `LC_TIME` settings of the second one in a new locale object. It also uses the `uselocale()` function, whose prototype is:

```
#include <locale.h>
```

```
locale_t uselocale(locale_t newloc);
```

The uselocale() function is given a locale object and makes it the locale for the calling thread, in this case the entire process, and returns the locale object in use before the call. Comments in the program are mostly omitted, to save space. The fully-documented program is in the book's source code distribution.

*newlocale_demo.c*

```
#define _XOPEN_SOURCE 700
#include "common_hdrs.h"
#define TESTNUMBER 123456789.12  /* Number to test locale.      */

int  main(int argc, char *argv[])
{
    time_t t;                   /* To store current time          */
    struct tm *tm;              /* To store broken-down time      */
    char buf[100];              /* To store formatted time string */
    char  err_msg[STRING_MAX]; /* For error messages             */
    locale_t loc, newloc;       /* Temporary locale objects       */

    if (argc < 2) {
        sprintf(err_msg, "Usage: %s locale1 [locale2]\n", argv[0]);
        usage_error( err_msg);
    }

    loc = newlocale(LC_NUMERIC_MASK, argv[1], (locale_t) 0);
    if (loc == (locale_t) 0)
        fatal_error(EXIT_FAILURE, "newlocale");

    if (argc > 2) {
        newloc = newlocale(LC_TIME_MASK, argv[2], loc);
        if (newloc == (locale_t) 0)
            fatal_error(EXIT_FAILURE, "newlocale");
        loc = newloc;
    }
    uselocale(loc);
    printf("With numeric settings of %s, number is: %'8.2f\n",
            argv[1], TESTNUMBER);
    t = time(NULL);
    if ( (tm = localtime(&t)) == NULL)
        fatal_error(EXIT_FAILURE, "localtime");

    if ( 0 == strftime(buf, sizeof(buf), "%c", tm) )
        fatal_error(EXIT_FAILURE, "strftime");
    printf("With time settings of %s, date/time is: %s\n", argv[2],  buf);

    uselocale(LC_GLOBAL_LOCALE);    /* So loc is no longer in use. */
    freelocale(loc);                /* Release storage for loc */
```

```
    return 0;
}
```

*Listing 4-14: A program that uses newlocale() to create a custom locale object*

Notice that the last step this program takes is to change the process's locale to `LC_GLOBAL_LOCALE` and then free the locale object that it created. The locale object was allocated memory by `newlocale()`. We need to free that memory. The `NOTES` section of the `newlocale()` man page indicates that our programs must free that memory using `freelocale()`, but we can't free it if it's in use, so we change locale to `LC_GLOBAL_LOCALE`, which is not a real locale object; it's a special value that can't be used as a locale. To illustrate how this program works, here's a run with the numeric settings from Spain and the date and time settings from Japan:

```
$ newlocale_demo es_ES.utf8 ja_JP.utf8
With numeric settings of es_ES.utf8, number is: 123.456.789,12
With time settings of ja_JP.utf8, date/time is: 2023年09月25日 09時55分36秒
```

The GNU C library includes functions that explicitly use locale objects as parameters. They're easily recognized because their names end in `_l` and they're documented on the same man pages as their non-`_l` counterparts. For example, `isalpha()` and `isalpha_l()` share a man page. Whereas the `isalpha()` function implicitly uses the locale set by a call to `setlocale()` or `uselocale()`, `isalpha_l()` is passed a locale object explicitly in its last parameter. This allows different threads of a process to use different locale objects. These functions were added to the POSIX standard in 2008 but not all systems support them. To use them, you need to provide a feature test macro in your programs. The man pages contain the specific macros that you need, depending on which function you want to use.

## Summary

The hands-on approach we use for learning how to write system programs is to try to write programs that are similar to existing commands, researching those commands to understand which resources they use and how they use them. Because all of the projects that we develop share a core of common code, in this chapter we showed what that code is, how it's organized, and how we'll incorporate it into our projects. We chose to start by implementing a simplified version of the `date` command because that command doesn't use system resources other than access to the system clock. Through a search of the man pages, we learned that we needed the `time()` system call and the `localtime()` and `strftime()` library functions to implement `date`. We went through a few incremental revisions of the program to demonstrate how to add optional formats, how to add the user option to display dates other than the current one, and finally, how to internationalize the program.

Learning how to internationalize the `date` program allowed us to explore the more general subject of internationalization. We explored the concept of a locale, studying the kinds of data that it encapsulates, the commands

available for viewing and manipulating them, and the programming interface to them as specified by the POSIX standards and as implemented in GNU/Linux. In particular, the GNU/Linux C library has many functions that are locale-aware and several functions for extracting information from locale objects.

## Exercises

1. Write a program that expects one or more hexadecimal numbers on the command line, and for each number, prints its value as a decimal integer, one per line. If any argument is not a valid hexadecimal number or has trailing characters that aren't hexadecimal digits, it should display '`not a valid number` for that word. It should allow a leading `0x` or `0X` but not require it. For example, if the executable is named `hex2int`, it should work like this:

```
$ hex2int   0xa  b 0xf00 foo abe
a = 10
b = 11
f00 = 3840
foo: not a valid number
abe = 2750
```

2. Write a program that sorts the words entered on the command line and prints them on the standard output, one per line. It should sort according to the collating sequence of the user's locale. A word is any sequence of characters other than whitespace. Hint: You'll need to store the words in an array.
3. Read the man page for `strtod()`, the function that parses floating-point decimals and returns their values. Based on that page, write a function named `get_longdbl()`, with the prototype

```
int get_longdbl(char *arg, int flags, long double *value,
    char *msg );
```

that stores into `value` the numeric value of its first argument. Based on the possible error value, design a set of flags to pass to this function to control what should constitute an error versus just a warning, such as whether it has trailing characters or is negative, or too large.

4. Write a program named `yearday2date` that, when given an integer argument, returns the date in the current year that it represents in the format <*Monthname*> <*dayofmonth*>, <*current year*>. January 1 is always day one of a year. For example, if you entered **`yearday2date 100`** in the year 1970, a non-leap year, it would print `March 10, 1970`. If the number is zero, it should report an error. If the number is greater than the number of days in the current year, it should calculate the

date in the future, and if it is negative, it should calculate the date in the past. For example, if you ran it in 1970, **`yearday2date -100`** should print the date in 1969 that is the 265th day of that year, `October 23, 1969`. Hint: Read the man page for `mktime()`.

5. The `locale` command without any arguments or options prints out the values of the categories in the current locale. Write a program that does this. This program will be very easy if you examine the source code in the file */usr/include/langinfo.h*. You'll see that a macro is defined there to make this easy.

# 5

# BASIC CONCEPTS OF FILE I/O

One of the most fundamental operations that system programs perform is the transfer of data to and from files. The transfer of data from a file to the memory space of a process is called *file input*, and the transfer in the opposite direction is called *file output*. Together, they're called *file I/O*. Because most system programs perform file I/O and because even the simplest of problems will require our programs to perform it as well, we study it next.

Our primary objectives in this chapter are to explain the issues and concepts related to accessing files in Unix and to demonstrate how to use the file-handling part of the kernel API. In addition, we'll explore file-handling issues that impact the performance and portability of the programs we write.

We start with an overview of the Unix I/O model, and then cover some key background concepts related to file permissions and processes in order to understand how Unix decides whether a process is allowed to access a file in a particular way. Following this, we examine in detail the elementary parts of the Unix kernel API related to file I/O. We'll implement a simplified version of the `cp` command, and we'll consider how various design decisions affect its overall performance.

## Introduction

High-level language libraries usually provide many different functions to perform I/O conveniently. In previous chapters, we saw examples of these, such as `scanf()` and `printf()` for input and output of textual data. Functions like these are often sufficient to solve some problems, but for other problems they aren't suitable, either because they don't give us enough control over how a read or write operation should be performed, or because our program needs to work with data that isn't plaintext. In addition, the cost of their convenience is usually longer running time compared to what's possible using system calls directly. If a program is not going to read or write large amounts of data, it may not matter how fast it transfers data, but it does make a big difference in performance when we want our programs to handle large datasets. Therefore, it's important to know how to use the lower-level functions for file I/O provided by the kernel.

## Universal I/O

From its inception, Unix employed a *universal* model of I/O. As Ritchie and Thompson wrote in 1974 [24], "the system calls to do I/O are designed to eliminate the differences between the various devices and styles of access." In other words, the same system calls that are used to perform I/O on disk files are also used on all other types of files, including device files. A program that can read from or write to disk files can also read from or write to devices, such as terminals, network interfaces, and peripheral devices. From a program's perspective, there's no distinction between random access devices, such as disks, and sequential access devices, such as tapes.

Unix frees you as a programmer from needing to know the details of devices such as disks, tapes, network interfaces, and terminals to write programs that transfer data to or from them. This is why Unix is said to provide device-independent I/O. You don't need to understand how the kernel performs this magic in order to write a system program, but we'll explain a bit about how it does so in Chapter 18.

## File Permissions Revisited

In Chapter 2, when we introduced the concept of file permissions in Unix, we covered just the basic ideas. Now we introduce one more significant concept that comes into play in the context of file creation, namely the file creation mask.

Every process has a *file creation mask*, which is a set of nine bits that restricts which permissions are enabled when the process creates a file. The file creation mask is commonly called a *umask*. Because this name is shorter, we'll call it a umask henceforth.

When a process calls a function that creates a file, that file is given an initial set of permissions, such as who can read it, who can write to it, and so on. Functions that create files usually have a mode argument that lets the process set those initial permissions, but the umask is applied to the

mode that the process tries to put on the file, possibly removing some permissions specified by that mode. The resulting initial permissions set on the file are what the process attempted to set on it, minus those that the umask removed.

### *Applying the Umask*

The umask is essentially an inverted mask: a 1-bit in the umask disables the corresponding permission in the created file, but a 0-bit in the umask does not disable the corresponding permission in the file if the process tries to enable it. We can view umask's nine bits as three groups of three bits each, corresponding to the permission bits in the file mode. For example, the umask

```
000010010
```

can be viewed as

```
000 010 010
```

The first group controls user permissions, the second controls group permissions, and the third controls others permissions:

```
0 0 0   0 1 0   0 1 0
r w x   r w x   r w x
user    group   others
```

Because an octal digit is equivalent to three binary digits, octal numbers are used to represent umask values. The previous umask value is octal 022, for example.

When a process calls a function to create a file with a mode of *mode*, the system applies the umask to it using the bitwise C operation (*mode* `&` `~`*umask*). In other words, the umask is inverted and bitwise-and'd to the mode. For example, suppose that `mode` is 110110010 and `umask` is octal 022, or binary 000010010. The complement of `umask` in binary is 111101101. The operation is thus

```
  110 110 010
& 111 101 101
= 110 100 000
```

Interpreting this as a permission string, it gives the user read and write permission, the group only read permission, and no permissions on the file to anyone else. The umask value 022 is a common value because it disables writing to a file by anyone other than its owner, but it doesn't limit reading. If we wanted an even more secure value that prevented reading by others for every created file, we'd set the process's umask to octal 066, or binary 000110110. The complement of 066 is 111001001, and applying it to the previous `mode`

```
   110 110 010
 & 111 001 001
 = 110 000 000
```

disables read and write by anyone except the user. Some people find it easier to apply a umask using a modified form of subtraction in which 0 − 1 = 0. The original, non-inverted umask is subtracted from the mode as in:

```
   110 110 010
 - 000 111 111
 = 110 000 000
```

When we treat its application as a form of subtraction, it's easier to remember that the umask acts like a filter that removes permissions.

### *Setting and Getting Umasks*

When you log in to a Unix system, the shell started up for you is assigned an initial, default value for the umask, often octal 022. This default value depends on the operating system and the shell. You can view your umask with the `umask` command. To see the octal value of the umask, enter:

```
$ umask
0022
```

This shows that the umask value is octal 022. The first `0` means the following digits are octal. If you want to see the permissions that are *not masked* by the umask in symbolic form, use the `-S` option:

```
$ umask -S
u=rwx,g=rx,o=rx
```

This shows that the umask doesn't mask out any permissions for the user, because `u=rwx`, and it doesn't mask out read and execute permissions for anyone else (`g=rx,o=rx`).

You can use the `umask` command to change the umask by giving it a umask value argument:**`umask 033`**:

```
$ umask
0022
$ umask 033
$ umask
0033
```

You can put a `umask` command into one of your shell configuration files if you want the shell's umask to be something other than the default value. For example, in `bash`, you could add the following lines into your *.bashrc* file if you want every interactive shell to use that umask:

```
# Set my umask to turn off group reads; others: no read, no write
```

```
umask 026
```

All shells that start up when you open a terminal window are interactive and read that file. If you put them into your *.bash_profile* file, then only the login shell will use that umask.

### Propagating Umasks

Whenever you run a process from the command line, its umask is inherited from the shell, so unless that process changes its umask, its value will be the one your shell had when you ran that command. Any processes created by that process will also have that umask, so the umask propagates downward to every process running on your behalf, unless one of these processes changes it.

A process can change its umask with the `umask()` system call. In Chapter 12, we'll go over examples of programs in which processes change their umasks.

## A Process's User IDs

File permissions exist to control which users can access files and how they can access them. The only way that a user can actually access a file is by running a command or a program, which is to say, running a process. Therefore, file permissions must determine which user processes can access files and how they can access them.

The essential idea that underlies how permissions are used is that every process is associated with at least one user ID. To be precise, on Linux, every process has four user IDs:

- A *real user ID*
- An *effective user ID*
- A *saved set-user-id*
- A *filesystem user ID*, which is Linux-specific

We'll explain them here.

On most Unix systems, the kernel uses the effective user ID when it needs to determine whether to grant a process permission to access a resource, such as memory, or to access files. On Linux, it uses the filesystem user ID to determine access to files, but the filesystem user ID is always equal to the effective user ID, so in effect, it's using the effective user ID as well.

Normally, when you run a program, the process that's created is assigned an effective user ID and a real user ID that are both equal to your user ID and thus the same. Sometimes, however, a process can have different effective and real user IDs. Usually, when they're different, the effective user ID gives the program greater privileges than the real user ID. A program can be run in such a way that the effective user ID of the running process is not that of the user who runs it, but is instead the user ID of the

owner of the program file. In the next section we explain what makes this possible.

## The setuid Bit

The highest-order bit in a file's mode is a bit named the *setuid* bit. If this bit is set, or enabled, for a file containing an executable program, then when a user who doesn't own the file runs that program, the process will have an effective user ID that is different from its real useruid. Specifically, whenever the program is run, the created process has an effective user ID equal to the user ID of the *owner of the program file*, and a real user ID equal to the user ID of the user that ran the program. You can see whether the setuid bit is enabled with the `ls -l` command. When it's set for a file, the permission string `ls -l` displays will have an `s` instead of an `x` for the user's execute permission letter.

Programs with the setuid bit enabled often have a need to temporarily change their effective user ID to the real user ID and then restore the effective user ID to what it was. The purpose of the saved set-user-id is to store the effective user ID in such programs for later retrieval. Programs that modify their effective user IDs are called *setuid programs*.

A good example to illustrate these concepts is the `passwd` command. The `passwd` command is usually contained in the */usr/bin/passwd* file. If we view the permissions on that file using `ls -l /usr/bin/passwd`, we see:

```
-rwsr-xr-x 1 root   root        59976 Nov 24 07:05 passwd
```

The `s` in the permission string indicates that the setuid bit is on. The file is owned by `root`, whose user ID is 0. When we run the `passwd` command, its effective user ID will be 0, but its real user ID will be our own user ID.

Try this experiment: open two terminal windows, and in one, run the `passwd` command without entering anything. In the second terminal, look at the `passwd` process's status by entering **`ps -o euid,ruid,ppid,pid,args -C passwd`**. This command displays the effective and real user IDs, the parent process and process IDs, and the command name for every running instance of `passwd`. You'll see that the real and effective user IDs are different:

```
$ ps -o euid,ruid,ppid,pid,args -C passwd
 EUID  RUID    PPID     PID COMMAND
    0   500   13744   14561 passwd
```

The column labeled `EUID` is the effective user ID, and the `RUID` column is the real user ID. The `RUID` column will contain your actual user ID. When you're finished, press CTRL-D twice in the terminal with the `passwd` program waiting for input. It will terminate without making any changes to your password.

The role of a process's credentials in file input and output should now be clear. A process can only access files for which it has permission to do so. This is determined by the file's permissions and the effective user ID of the running process.

## Input/Output Concepts

Before a process can access a file, it needs to establish a connection to that file in order to communicate with it. A *connection* is an object that manages and controls a process's access to the file. It contains data such as the *file offset*, also called the *file pointer*, which is the offset in the file at which the next operation takes place, various flags and mode bits that control the manner in which operations are performed, information to locate the file, and pointers to kernel functions that the process can invoke. To create this object, a process must *open* the file. In fact, the POSIX specifications call the connection object an *open file description (OFD)*, which is the term we'll use here.

In Unix, a process can open a file to access it in one of three modes:

**Read mode** Exclusively to read from it

**Write mode** Exclusively to write to it

**Read/write mode** To both read and write

These are called the *access modes* of the opened file. The access mode is one of the items stored in the OFD.

The operation that opens a file returns an identifier that serves as a reference to the newly created OFD. This identifier is called a *file descriptor*. A file descriptor is a typically small, non-negative integer. Once you've opened a file and have a file descriptor for the connection, you must pass that descriptor to all subsequent operations on that file.

Traditionally, Unix systems did not prevent multiple processes from opening the same file, and POSIX codified that behavior. POSIX-conforming systems allow multiple processes to access the same file at the same time, which is an important feature of Unix. It's why it is possible for multiple users to run the same command or change their passwords at the same time. In fact, a single process can open the same file multiple times as well. Unix systems do provide locking mechanisms so that a process can prevent other processes from opening a file while it's accessing it.

Each open operation on the file creates a distinct open file description for that file and returns to its caller a unique file descriptor for that description. A process may also have multiple file descriptors that refer to a single OFD, because Unix provides a means by which a process can duplicate the descriptor that refers to an OFD, so that the new descriptor and the original both point to the same OFD. We'll study the duplication of file descriptors in Chapter 18, and we'll examine the various data structures that the kernel uses for file I/O in greater depth in Chapter 14.

Figure 5-1 depicts a portion of the tables and data structures created to manage I/O operations on files.

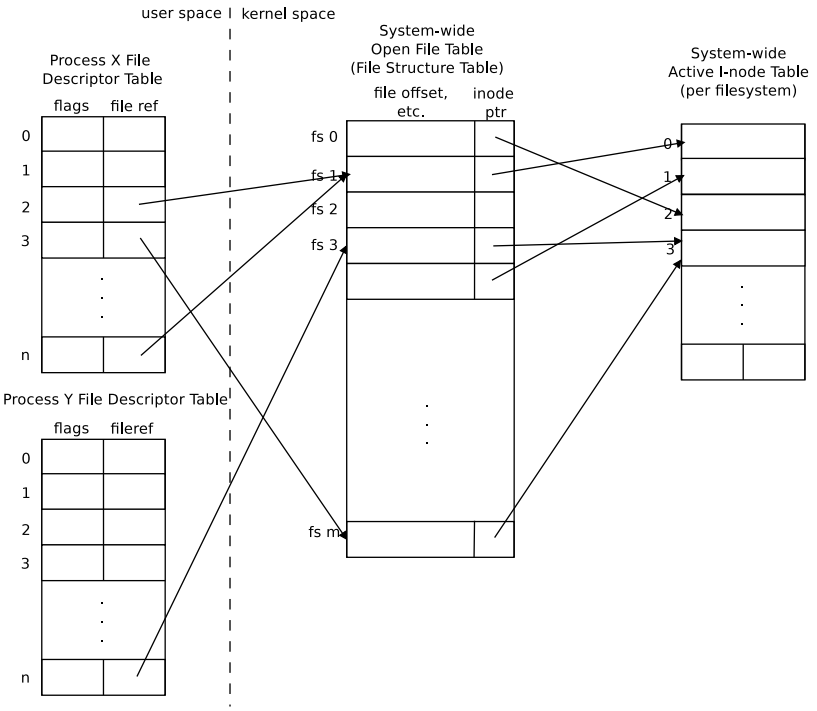



*Figure 5-1: The tables used to manage files opened by processes.*

Figure 5-1 shows the kernel's in-memory *open file table*, also known as the *file structure table*. Open file descriptors point to entries in this table. The entries in this table have many members, among which is a pointer to the inode that represents the actual file. In Figure 5-1, Process X has two open files, with OFDs at locations `1` and `m`. It duplicated a descriptor (2) so that descriptors 2 and `n` point to the same open file description at index 1 in the file structure table. Process Y and Process X each opened the file with inode 3, so they have two different open file descriptions at locations `m` and 3, respectively, pointing to it.

When a program has finished all reading and writing and no longer needs access to the file, it needs to *close* it. Closing the file breaks the connection between the program and the file. It frees the file descriptor, so that it no longer refers to the OFD. If there are no other descriptors pointing to the OFD, the resources used for the OFD are freed and the OFD is removed. (One field in the OFD is a reference count that keeps track of how many descriptors point to it.) Even more importantly, if your program doesn't close a file to which it was writing, some of the data may be lost. This can happen because, in some cases, writing to a file is not direct—the data is written to system buffers that are eventually written to the underlying hardware. Closing the descriptor is necessary to *flush* the buffers to the device, but even closing it may not be sufficient.

In summary, a process performs file I/O in three steps:

1. Open a connection to the file to read or write.
2. Perform the reads and/or writes through that connection.
3. Close the connection to the file.

## Standard File Descriptors

When a process is started from a shell, it inherits three open file descriptors, numbered 0, 1, and 2. These descriptors refer to connections that have already been opened by the time the process starts execution:

- File descriptor 0, called the *standard input*, refers to the connection from which it receives input.
- File descriptor 1, called the *standard output*, refers to the connection to which it sends output.
- File descriptor 2, called the *standard error*, refers to the connection to which it sends error messages.

For clarity, programs can use the symbolic constants `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`, defined in *unistd.h*, for the numbers 0, 1, and 2 respectively.

If the shell that creates the process is an interactive shell, meaning a shell that you're using to enter commands, the three descriptors are usually connected to the terminal device in which the shell is running. The input comes from the keyboard, and the output and error are written to the terminal screen. However, if any shell redirection operators were applied to the command that the process is executing, those descriptors may be pointing elsewhere. We'll explain how that works in Chapter 14.

When the process terminates, these descriptors are closed automatically, which is why you never have to explicitly open and close them in your programs.

## The Kernel I/O Interface

In "Creating, Removing, and Copying Files and Links" in Chapter 2, we introduced the `cp` command, Unix's command for copying files. The simplest form of that command,

```
cp file1   file2
```

copies the contents of *`file1`* to *`file2`*. If the latter file already exists, `cp` will silently overwrite its contents; otherwise, it creates a new file with that name.

Writing our own version of this command without using any library functions is a good way to learn how to use the kernel's I/O interface. We'll first research the kernel's system calls for opening, reading, writing, and closing files. We'll follow the same approach that we used when writing the `showdate` program in Chapter 4, namely, we'll search the man pages to find the system calls we need, starting with one that opens files.

### Opening Files

Since we're looking for system calls related to opening files, we restrict the apropos search to Section 2 and give it the keyword open:

```
$ apropos -s2 open
creat (2)             - open and possibly create a file
epoll_create (2)      - open an epoll file descriptor
epoll_create1 (2)     - open an epoll file descriptor
flock (2)             - apply or remove an advisory lock on an open file
mq_open (2)           - open a message queue
name_to_handle_at (2) - obtain handle for a pathname and open file via a handle
open (2)              - open and possibly create a file
open_by_handle_at (2) - obtain handle for a pathname and open file via a handle
openat (2)            - open and possibly create a file
openat2 (2)           - open and possibly create a file (extended)
perf_event_open (2)   - set up performance monitoring
pidfd_open (2)        - obtain a file descriptor that refers to a process
```

The four contenders from the returned list that warrant further inspection are creat(), open(), openat(), and openat2().

It turns out that creat(), open(), and openat() share a single man page. The man page for openat2() states that this system call extends the functionality of openat(), that it's a system call without a *glibc* wrapper and therefore has to be called using the syscall() function, and that it's a Linux-specific call, meaning that it isn't required by POSIX.1-2017. If we use it, our program would be limited to Linux systems only. For these reasons, we'll focus on the other functions and examine their man pages. The SYNOPSIS section there is:

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

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
int openat(int dirfd, const char *pathname, int flags);
int openat(int dirfd, const char *pathname, int flags, mode_t mode);
```

First we see that even though these are system calls, they're not declared in *unistd.h*. Certain system calls are declared in other headers, in this case, in *fcntl.h*. The *sys/stat.h* and *sys/types.h* headers contain type and constant declarations needed by these calls. On some Linux systems, you may not need to include the *sys/types.h* header file. Check your local documentation to be sure.

Doing a quick scan of the page without reading all of the details, we discover first that creat() is essentially a special case of open() and that openat() is also an extension of open() that gives us the ability to name the file to be

opened relative to a given directory. We can safely conclude that `open()` is the function we need to research.

### The open() System Call

There are two prototypes for `open()`, their only difference being the presence of a third argument. Some man pages present these instead as a single prototype:

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

The declarations are equivalent because the third argument is optional.

The first argument, named `pathname`, is a character string containing the pathname of the file to be opened. The second argument, named `flags`, is a bit mask. A *bit mask* is an integer whose individual bits represent distinct Boolean values that can be inspected or modified. This argument specifies the access mode for the file: reading only, writing only, or reading and writing, but other flags can be bitwise-or'd into it to control how the file is opened or created and how file operations are performed. We'll get to those shortly.

The optional third argument is also a bit mask, but it's used only when `open()` is being called to create a new file, in which case it specifies the permissions on that file.

If successful, the call to `open()` creates a new open file description and returns an integer file descriptor that refers to it and can be used in subsequent operations on the file. This descriptor is the lowest-numbered file descriptor not already in use by the process. We'll see in Chapter 14 why this fact is important. If the call isn't successful, it returns -1, and the error code is set in `errno`. Note that a process can open the same file multiple times; each call to `open()` creates a new open file descriptor.

The `flags` bit mask must include one of the following access modes, defined in *fcntl.h* and zero or more *file creation flags* and *file status flags* bitwise-or'd into the mask:

**`O_RDONLY`** Open the file for reading only.

**`O_WRONLY`** Open the file for writing only.

**`O_RDWR`** Open the file for reading and writing.

For example, to open an existing file named *infile* in the working directory for reading, we'd write:

```
int fd;
fd = open("infile", O_RDONLY);
if ( -1 == fd )
    /* Handle error */
```

If we try to open a file for reading that doesn't exist, `open()` will return `-1` and store the `EACCES` error code in `errno`.

Similarly, to open an existing file named *outfile* for writing, we'd write:

```
int fd;
fd = open("outfile", O_WRONLY);
if ( -1 == fd )
    /* Handle error */
```

This second example raises a few additional questions:

- If the file to which we want to write doesn't exist, will `open()` return an error or will it create a new file?
- If the file to which we want to write exists but isn't empty, will `open()` return an error or silently overwrite it?
- If that file exists and isn't empty and `open()` doesn't return an error, will writes to the file start at its beginning, replacing its contents, or will they be appended to the end of the current contents?

The answers to these questions depend on the values that we bitwise-or into the `flags` argument. These flags fall into two categories: file creation flags and file status flags. *File creation flags* influence the behavior of the `open()` call itself, whereas *file status flags* modify the actual file operations.

The man page has a complete list of all flags, but to understand many of them we need to know more about advanced methods of I/O, which we'll cover in Chapters 14 and 15. For now, I'll explain the file creation flags that are relevant to the preceding questions:

**`O_CREAT`** Creates the file specified by `pathname` if it does not already exist.

**`O_EXCL`** Used with `O_CREAT`, forces `open()` to fail if `pathname` already exists.

**`O_TRUNC`** If `pathname` exists and is a regular file, truncates it to zero length before writing to it.

To answer the first question, if we try to open a file for writing, using either `O_WRONLY` or `O_RDWR` with no other flags, and the file doesn't exist, `open()` will fail. If we bitwise-or the `O_CREAT` flag to `O_WRONLY`, as in

```
int fd = open("outfile", WR_ONLY|O_CREAT);
if ( -1 == fd )
    /* Handle error */
```

it will instead create a new file with the given name. In short, trying to open a new file for writing fails unless the `O_CREAT` flag is bitwise-or'd into the second argument.

If we pass `O_CREAT` and no other flags to the call but the file exists, `open()` will not fail, but the file's contents will be written starting at the beginning of the file. If we write *N* bytes to the file, the first *N* bytes of the original file will be replaced but the rest of the file will remain.

If we want the entire file to be replaced, we need to bitwise-or the `O_TRUNC` flag as well. This sets the file to zero length before the first write to it:

```
int fd = open("outfile", O_WRONLY|O_CREAT|O_TRUNC );
if ( -1 == fd )
```

```
    /* Handle error */
```

This is a typical way of opening a file for writing, but it's not the only way. You also can bitwise-or the `O_EXCL` flag to `O_CREAT` to produce a different behavior, as in

```
fd = open("outfile", O_WRONLY|O_CREAT|O_EXCL );
```

which returns an error if *outfile* exists and if it doesn't, creates it. In short, if you want the file to be replaced if it exists or create it if it doesn't, use `O_WRONLY|O_CREAT|O_TRUNC` as the second argument. If you want to prevent the file from being overwritten if it exists and create it if it doesn't, pass `O_WRONLY|O_CREAT|O_EXCL` instead.

Most of the other possible combinations of these file creation flags, used with read, write, or read-write access, will have either undefined or undesirable behavior. For example, if you try to open a file for read-write access and pass the following flags, the call will truncate the file when it's opened, leaving nothing to read initially:

```
fd = open(argv[1], O_RDWR |O_CREAT|O_TRUNC);
```

Make sure that's really what you want to do.

None of the flags just introduced will allow your program to open an existing file for writing and start writing to it at the end of the file, so that successive opens of the file for writing enlarge the file. You might want to do that if a program needs to maintain a logfile and append to it when it's running. In Chapter 6, I show you how to do this.

#### Attributes of Created Files

Let's consider the last parameter of the `open()` call, which is only needed when the call creates a file, meaning that it's called in either read-write or write mode, with the `O_CREAT` flag in the second argument. It has no effect otherwise.

When `open()` creates a file, we need to know the following:

- Which user is the owner of the file?
- What group is associated with the file?
- What are the permissions on the file?

The man page tells us that the owner is set to the effective user ID of the calling process, not the real user ID.

The answer to the group ownership question depends on the particular Unix system, but there is no simple answer. Just as a process has real and effective user IDs, it also has real and effective group IDs, with analogous meanings. In Linux, the file's group ownership is either the effective group ID of the calling process or the real group ID of the directory in which the file is created. It depends on several factors. Most of the time, it's going to be the effective group ID of the calling process, which is most likely the same as the real group ID. Check your local documentation to know for sure.

Finally, we focus on what permissions the file is given when it's created. If you don't supply a third parameter but you've passed `O_CREAT` in flags, the permission is unpredictable. You must provide that parameter. If you do, the permissions are the mode that you pass to it, modified by the process's umask. You can specify the mode either by supplying a literal number, such as the octal value 0600, or by a bitwise-or of one or more of the symbolic constants defined in the *sys/stat.h* system header file and shown in Table 5-1.

**Table 5-1:** Symbolic Constants for File Mode

| Constant | Numeric value | Permission |
|---|---|---|
| `S_IRWXU` | 00700 | User has read, write, and execute permission. |
| `S_IRUSR` | 00400 | User has read permission. |
| `S_IWUSR` | 00200 | User has write permission. |
| `S_IXUSR` | 00100 | User has execute permission. |
| `S_IRWXG` | 00070 | Group has read, write, and execute permission. |
| `S_IRGRP` | 00040 | Group has read permission. |
| `S_IWGRP` | 00020 | Group has write permission. |
| `S_IXGRP` | 00010 | Group has execute permission. |
| `S_IRWXO` | 00007 | Others have read, write, and execute permission. |
| `S_IROTH` | 00004 | Others have read permission. |
| `S_IWOTH` | 00002 | Others have write permission. |
| `S_IXOTH` | 00001 | Others have execute permission. |

For example, you can specify the `rw-r--r--` permission on the file by passing the bitwise-or `S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH` in the third parameter.

### Errors When Opening Files

The `open()` call can fail for various reasons, and the man page describes all of the associated error values. A good program would try to inform its user why it failed so they can correct the error. Some of the causes of failure follow:

- The specified file doesn't exist.
- The user doesn't have permission to open the file.
- One or more flags passed to the open call are invalid.
- The open call tried to create a file and couldn't, either because it didn't have permission in the specified directory or the file exists and `O_CREAT` and `O_EXCL` were passed.

Because a failure to open a file will most likely prevent any useful computation to continue, the easiest way to handle the errors is to display an error message and terminate. The `fatal_error()` function presented in Chapter 3 does this, so a pseudocode fragment for opening a file should be of the form:

```
int fd;  /* File descriptor to receive */
fd = open(pathname, file_opening_flags, mode);
if ( -1 == fd )
```

```
    fatal_error(errno, "name of function");
```

The program might also have to release any resources that it acquired, such as other files it might have opened successfully and modified but has not yet closed.

## Closing Files

It's best to learn the proper way to close files before we consider how to read from or write to them for a few reasons:

- Closing a file shouldn't be an afterthought, because the close operation performs important tasks, as we noted previously in this section.
- We won't be able to write any programs to demonstrate how to read from or write to files unless we know how to open and close them.
- Opening and closing are somewhat symmetric operations that act like bookends surrounding actual I/O operations, respectively acquiring file descriptors and relinquishing them, so it's natural to learn about the closing operation right after opening.

### The System Call to Close a File

We need to find the system call that closes a file. A search in section 2 of the man pages for the keyword `close` produces a single page:

```
$ apropos -s2 close
close (2)            - close a file descriptor
```

Notice that this man page's one-line description states that this call closes a *file descriptor*, not a *file*. Although we think of the open operation as opening files, we think of the close operation as closing a file descriptor. The man page for `close()` starts with:

```
SYNOPSIS
       #include <unistd.h>
       int close(int fd);

DESCRIPTION
       close() closes a file descriptor, so that it no longer refers to any
       file and may be reused.  Any record locks (see fcntl(2))  held  on  the
       file it was associated with, and owned by the process, are removed (re
       gardless of the file descriptor that was used to obtain the lock).

       If fd is the last file descriptor referring to the underlying open file
       description  (see open(2)), the resources associated with the open file
       description are freed; if the file descriptor was the last reference to
       a file which has been removed using unlink(2), the file is deleted.
```

```
RETURN VALUE
       close()  returns zero on success.  On error, -1 is returned, and errno
       is set appropriately.
```

Observe that the header file needed for `close()` is not the same as the one for `open()`. It's declared in *unistd.h*, whereas `open()` is declared in *fcntl.h*.

This function has a single argument, which is the file descriptor of the connection to be closed. The call closes that file descriptor. If a file has been opened by a process via multiple calls to `open()`, the `close()` call doesn't close the other connections; it closes only the one corresponding to `fd`.

The second paragraph of the man page `DESCRIPTION` implies that there can be multiple file descriptors referring to the *same* open file description. When we introduced the `open()` call, we pointed out that each call to it creates a new open file description and returns a descriptor that refers to it, so how is that possible? The answer lies in the fact that file descriptors can be duplicated using the `dup()` or `dup2()` system call. These calls create copies of file descriptors that point to the same open file descriptions.

### Errors When Closing Files

If `close()` is successful, its return value is 0, but if not, it returns -1. You might wonder what could possibly go wrong when closing a file and why `close()` can fail.

One reason might be that we passed it a bad file descriptor when we called `close()`, in which case it returns -1 and sets `errno` to `EBADF`. Another reason could be that the kernel, in the middle of executing code within the `close()` system call, may be given an urgent task to complete, one so urgent that it has to drop the `close()` call immediately to handle that task. In this case, the call returns −1 and sets `errno` to `EINTR`. Further still, the file may not have been on the local machine or local drive. It might be a file on a remote system that we're accessing across a network. The network connection might have gone down, in which case the close operation cannot complete its actions, and again, we'd get a -1 return value, and most systems will set `errno` to `EIO`.

Finally, if this file had been opened for writing, many other problems could cause `close()` to fail. For example, the kernel may discover in the `close()` call that it cannot complete a transfer of data to a disk file, because, for example, we've run out of disk space. On some systems, this error isn't reported until we close the file, when `errno` would be set to `ENOSPC` meaning no more disk space, or `EDQUOT`, meaning this write exceeded our quota.

The `NOTES` section of the man page provides guidance on how to handle the various errors if `close()` fails. If `close()` returns -1, we should not retry calling it, because doing so might cause other problems, particularly for programs with multiple threads. The failure value is supposed to be used only for diagnostic or remedial purposes, which means it's best to try rewriting the data to the file or writing to a new file if possible. In fact, the most recent version of the man page as of this writing states, "A careful programmer who wants to know about I/O errors may precede `close()` with a call to `fsync(2)`." The `fsync()` function forces any remaining writes to a file to be

flushed to the underlying device, so that a failure in closing isn't related to the attempted writes to it. By calling `fsync()` before calling `close()`, we'd see any I/O errors related to writing first and could handle them before closing the descriptor.

Handling the `EINTR` error in a comprehensive and portable way in POSIX-conforming systems is difficult because different implementations handle this error differently. POSIX.1-2017 doesn't specify what an implementation is supposed to do if the kernel is interrupted while executing code in the `close()` function. Therefore, some implementations guarantee that the file descriptor has been closed despite the error. Others don't close the file descriptor and require that `close()` be called again, which, as noted previously, can cause other problems. To handle closing errors in a portable way, a program would have to respond to this error in a different way depending on which implementation it's running. This in turn requires checking at runtime which kernel is running and which libraries it's using.

At this point, we'll present a simpler way to handle closing errors in which we call `fsync()` before `close()`, and if `close()` sets `errno` to `EINTR` we just exit:

```
errno = 0; /* Need to include <errno.h>. */
/* On some Unix systems, we need to check whether fd has been used
   for writing. On Linux, we don't need to do this. */
return_val = fsync(fd); /* Flush data to device. */
if ( -1 == return_val) {
    /* Error trying to flush data to device.
       Depending on application, we might need other
       actions here. */
    fatal_error(errno, "fsync() to  pathname");

/* fsync() was successful. */
errno = 0;
if ( -1 == close(fd) )
    fatal_error(errno, "closing pathname");
```

The preceding code is safe to use in Linux because it isn't an error in Linux to call `fsync(fd)` when `fd` was opened for reading only.

### Reading From Files

We need to find the system call that can read the contents of a file, not just text files but arbitrary files. In Chapter 1, we saw that in Unix, from the kernel's perspective, a non-directory file is simply a sequence of bytes without structure; we need a function to read such a file. We'll resort to our usual method for finding the right call, namely a man page search. We try the obvious search, using the exact (option `-e`) keyword `read`, limiting the search to section 2:

```
$ apropos -s2 -e read
```

```
--snip--
pread (2)         - read from or write to a file descriptor at a given offset
pread64 (2)       - read from or write to a file descriptor at a given offset
preadv (2)        - read or write data into multiple buffers
preadv2 (2)       - read or write data into multiple buffers
pwrite (2)        - read from or write to a file descriptor at a given offset
pwrite64 (2)      - read from or write to a file descriptor at a given offset
pwritev (2)       - read or write data into multiple buffers
pwritev2 (2)      - read or write data into multiple buffers
read (2)          - read from a file descriptor
readdir (2)       - read directory entry
--snip--
```

On my Linux system, the search returned more than 20 hits. This list is a fragment of those results, however, it does contain two functions that warrant further research:

```
pread (2)        - read from or write to a file descriptor at a given offset
read (2)         - read from file descriptor
```

The first, `pread()`, may not be suitable. When we read its man page, we see that it's more general than we need it to be and is really intended for multithreaded programs. We want to read the man page of the second, `read()`. If we enter

```
$ man read
READ(1POSIX)               POSIX Programmer's Manual              READ(1POSIX)
--snip--
NAME
       read - read from standard input into shell variables
--snip--
```

we get the page for a `read` command, not the one for the `read()` system call. We need to specify the section number in the `man` command:

```
$ man 2 read
--snip--
NAME
    read - read from a file descriptor

SYNOPSIS
    #include <unistd.h>
    ssize_t read(int fd, void *buf, size_t nbytes);

DESCRIPTION
read() attempts to read up to nbytes bytes from file descriptor fd into
the buffer starting at buf.
--snip--
```

Note that to use `read()`, we need to include the *unistd.h* header file. The function has three parameters: an integer file descriptor `fd`, a void pointer named `buf`, and one of type `size_t`, named `nbytes`. The man page states that it reads up to `nbytes` many bytes from the file descriptor `fd` into the buffer whose starting address is `buf`.

The bytes that are read are stored into memory locations starting at `buf`, which is declared as type `void*` so that any address can be passed to it with a type cast. It's worth pointing out that this parameter is named `buf` to emphasize that it's memory in the calling process that temporarily holds data to be transferred from the file. A program calling `read` should copy that data out of `buf` before calling `read` again. The third parameter, `nbytes`, is the number of bytes to read.

The return value is of type `ssize_t`. This is a system type similar to `size_t` except that it's a *signed integer* type, so that it can store negative numbers. The return value is either the number of bytes actually read, which can never be larger, but might be smaller, than `nbytes`, or -1, if something went wrong, in which case `errno` contains the error value.

Figure 5-2 represents the actions resulting from a call to `read(3,buf,len)`.

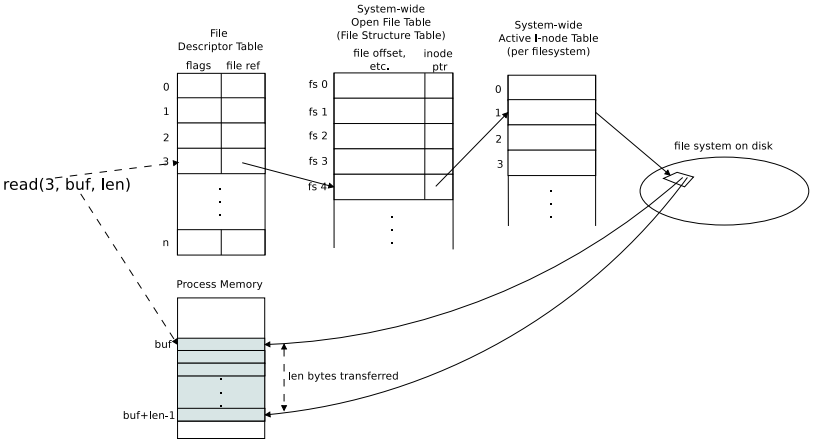



*Figure 5-2: A read of `len` bytes by a process from the file with file descriptor `3` to memory location `buf`*

The kernel uses `3` file descriptor 3 to locate the open file description in the file structure table. It uses that OFD to locate the inode for the file, which stores the address on disk of the file's data. When we first discussed open file descriptions in "Input/Output Concepts" on page 191, we noted that one of the members of an open file description is a file offset, which always points to the place in the file to perform the next operation. The reading of data starts at the file offset and in this case attempts to read `len` bytes. The data is copied into the memory locations in the process's address space starting at `buf`. The read operation advances the file offset by `len` bytes.

Let's take a look at a code fragment that puts some of these ideas together. Suppose that `fd` is a valid file descriptor that we've opened for reading, `buffer` is a `char` array of size 100, and `return_val` is a variable of type `ssize_t`.

The following code fragment shows how to repeatedly read 100 bytes of data at a time from the file associated with `fd` until there's no more data to read:

```
BOOL done = FALSE;  /* Flag to indicate no more data */
while ( ! done ) {
    return_val = read(fd, buffer, 100);
    if ( 0 > return_val )
        /* An error code was returned during reading  */
        ❶ fatal_error(errno, "error reading file...");
    else if ( 0  == return_val )
        /* The end of file was reached - stop reading */
        done = TRUE;
    else
        ❷ /* buffer[0...return_val-1] contains the bytes just read.
          Transfer this data to its final destination before
          it is overwritten by the next call to read(). */
}
```

This is the structure of a typical read loop. The loop repeatedly calls `read()` until it returns either zero or a negative value. A negative value indicates an error during reading, in which case we handle the error by calling the `fatal_error()` function ❶ (which we presented in Chapter 3) to print a message and exit the program. A zero return value just means we've read all there is to read, so we set the flag `done` to `TRUE` to break the loop. Any other value means that `read()` was successful. That case is handled in the `else` clause ❷. which in this fragment is just a comment indicating that `buffer` contains the data just read. That comment tells us we need to transfer the data before it's overwritten, which might mean writing it to a file or copying it into some data structure, for example.

There's no guarantee that the `read()` call actually read 100 bytes; it might have read fewer bytes, which is why the comment states that `buffer[0...return_val-1]` is the data just read. If, for example, only 256 bytes remained in the file at the current position of the file offset, then the next two calls to `read()` would read 100 bytes each, and the third, just 56 bytes, as depicted in Figure 5-3.

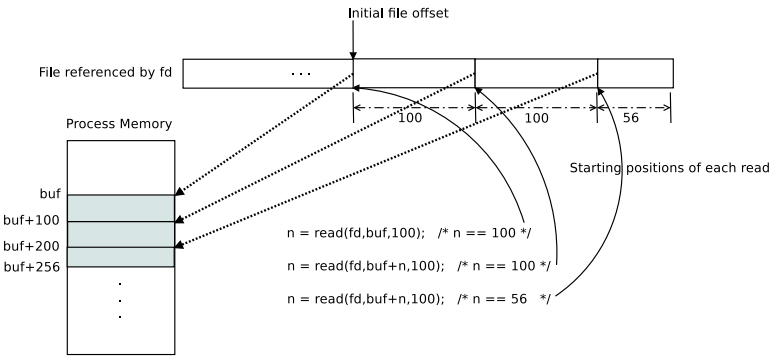



*Figure 5-3: Three successive reads with the third returning fewer bytes than requested*

You can't assume that the number of bytes requested is the same as the number received, and it's not an error when it isn't.

Remember that each successive call to `read(fd, buf, nbytes)` starts reading in the file referenced by `fd` at the byte immediately following the last byte read by the previous call, because the file offset is advanced by the read. This is why successive calls will eventually read the entire file without missing or duplicating bytes. The ability to read the entire file correctly hinges on `read()`'s advancing that file offset.

The next step is to find the system call that we can use for writing to files. Once we know that, we'll be ready to create a program to copy files.

### *Writing to Files*

To find the system call that can write to files, an obvious choice of a search in the man pages would be

```
$ apropos -s2 write
```

This search on my system returns about 20 different man pages. Depending on which distribution you're running, it might be more or less than that. We can refine the search to produce a smaller set of pages with the `-a` option to `apropos`, which lets us search for pages that match *all* of the supplied keywords, and give it both `write` and `file`, since we want to write to files in particular:

```
$ apropos -s2 -a write file
_llseek (2)        - reposition read/write file offset
llseek (2)         - reposition read/write file offset
lseek (2)          - reposition read/write file offset
pread (2)          - read from or write to a file descriptor at a given offset
pread64 (2)        - read from or write to a file descriptor at a given offset
pwrite (2)         - read from or write to a file descriptor at a given offset
pwrite64 (2)       - read from or write to a file descriptor at a given offset
write (2)          - write to a file descriptor
```

The very last hit is the one we want: the function that writes to a file descriptor. Its man page begins with

```
SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t nbytes);

DESCRIPTION
       write() writes up to nbytes bytes from the buffer starting at buf to the
       file referred to by the file descriptor fd.
--snip--
RETURN VALUE
       On success, the number of bytes written is returned.  On error,  -1  is
```

```
       returned, and errno is set to indicate the cause of the error.
--snip--
```

The `write()` system call is a symmetric counterpart to the `read()` call. It writes `nbytes` bytes starting at the address given by `buf` to the file associated with the `fd` file descriptor. The return value when it's successful is the number of bytes actually written. It'll never be greater than `nbytes`. If there's an error, it returns -1 and `errno` contains the error code. Like the buffer parameter of the `read()` call, this buffer parameter is declared as a void pointer, so that typecasts can be used to transfer any type of data.

The man page provides the details for using `write()`. A call such as

```
write(fd, buffer, num_bytes)
```

attempts to transfer `num_bytes` bytes from the memory location pointed to by `buffer` to the current position of the file offset in the file opened for writing via the `fd` file descriptor. The initial position of the file offset depends on how the file was opened. For example, if you opened the file for writing, passing the `O_WRONLY|O_CREAT|O_TRUNC` flags, the file offset will be at the start of the file. After each call to `write()`, it's incremented by the number of bytes actually written, so that the next call will write its data immediately after the data just written. Thus, no *holes* are created in the file under normal usage.

A program calling `write()` should check whether it returns -1, which indicates a write error, and handle the error appropriately. If `write()` doesn't return -1, there was no error in writing, but it's possible that the number of bytes actually written is less than the number of bytes that were supposed to be written. This *partial write* to an ordinary disk file can be caused by a variety of reasons: the file might have reached a predefined maximum size, the disk might be full, or the user's disk quota might be reached. After a partial write, your program can either just display a suitable error message and exit, or try to write the remaining data again. If it calls `write()` again after the partial write, it'll either transfer the remaining bytes or return an error, which it can then handle.

A simple way to call `write()` that doesn't try to rewrite after a partial write is of the form:

```
errno = 0;
result = write(fd, buffer, num_bytes);
if ( result == -1 )
    /* Error in writing - use errno value to print a message
       to user, exiting if appropriate. */
else if ( result < num_bytes )
    /* Some but not all data was written; display message and exit. */
else
    /* write() was successful and all data was written. */
```

It is always a good idea to check whether a partial write occurred, and at the least, inform the user that it did.

**NOTE** *A successful call to `write()` to a disk file doesn't necessarily transfer the data to the disk. In fact, on most modern Unix systems, writing to a file doesn't cause any immediate disk I/O. Instead, the data is transferred to kernel buffers, which are written to the disk at a later time. This practice generally reduces disk I/O, saves time in the kernel, and speeds up the writes. Chapter 14 covers this concept of kernel buffering.*

We're now ready to write a simple version of the Unix `cp` command, which we name `spl_cp`.

## Writing a copy Command

The simplest form of `cp` makes a copy of one file to a file with a different name, using the syntax:

```
cp source_file target_file
```

The two arguments to `cp` can be any pathnames, including those with symbolic links. To illustrate, suppose our current working directory has a symbolic link named *backups*:

```
$ ls -F
file1  backups@
```

The `-F` option to `ls` classifies the directory entries with an *append indicator*, which is one of `*`, `/`, `=`, `>`, `@`, or `|`. The `@` symbol indicates that an entry is a symbolic link, showing in this example that *backups* is a symbolic link. We can see what the target of the *backups* symbolic link is with the `readlink` command:

```
$ readlink backups
/data/backups/
```

The `readlink` command displays the full pathname to which a symbolic link refers, showing in this case that *backups* is a link to a directory, since it's displayed with a trailing slash. You can use the `realpath` command also; for this simple case they behave exactly the same.

If we issue the command

```
$ cp file1  backups/current/file1_bkup
```

`cp` makes a copy of *file1* in the directory */data/backups/current/* with the name *file1_bkup*.

Although the command seems simple enough, several questions about its behavior come to mind immediately:

- If *`target_file`* already exists, will `cp` replace it, or will it refuse to replace it and issue a warning instead?
- After a successful copy, what permissions will *`target_file`* have, and which user and group will own it?

- Can *source_file* and *target_file* refer to the same file? In other words, can cp replace a file by itself, or is that an error? Remember that the two files can be different links to the same file in Unix systems.
- If *source_file* is a symbolic link to another file, does cp copy the file referenced by the link or the link itself?
- Can cp make copies of special files and directories?

The POSIX specification of cp[17] answers all of these questions for systems that conform to POSIX requirements:

- If *target_file* exists, cp truncates the file and replaces its contents with the contents of *source_file*, an action known as *clobbering*. This is dangerous, as you cannot recover a file once you've clobbered it, so many people use the interactive option -i to cp, which prompts the user before overwriting the file:

```
$ cp -i README  README.md
cp: overwrite 'README.md'? n
$
```

  Any answer that begins with y or Y is interpreted as *yes*, and anything else is taken as a negative answer.
- The permissions and ownership given to *target_file* depend on whether or not it existed before the copy operation. If it existed before, they will remain the same as they were. If it is newly created, the mode will be the mode of *source_file* modified by the user's umask and the ownership will be the same as that of *source_file*.
- POSIX doesn't require an implementation of cp to detect whether the source and target pathnames refer to the same file, but most implementations do. Suppose, for example, that *linux_cheatsheet* and *commands* are links to the same file, which we can verify with the ls -i command, which prints the inode numbers of the files:

```
$ ls -i
6690145 commands  6690145 linux_cheatsheet
```

  Then if we enter:

```
$ cp linux_cheatsheet commands
cp: 'linux_cheatsheet' and 'commands' are the same file
```

  we see that cp looks not at the names but at the files to which they refer.
- If *source_file* is a symbolic link to another file, then cp copies the target of that link, not the link itself.
- Without options, cp does not copy directories, but it can copy special files that a user has the privilege to read.

Copying a file does not preserve any attributes other than the mode and ownership. To preserve the timestamps and other attributes when copying, we can use the `-p` (short for *preserve*) option.

A final point to remember is we cannot create a file in a directory unless we have write permission on that directory. Therefore, if `cp` needs to create the target file, we must have write permission on the target directory.

### Design of the copy Program

With all of the preceding considerations in mind, we can now outline the structure of our `spl_cp` program:

1. Check that the command line has two arguments and exit with a usage error if it doesn't.
2. Open the first argument file for reading, which we'll call the *source* file. If it cannot be opened for reading, report the error and exit. This should take care of detecting whether it's a directory or a file that we don't have permission to read.
3. Open the second argument file for writing, passing the bitwise-or of the `O_CREAT` and `O_TRUNC` flags to allow it to be created if it doesn't exist and overwritten if it does.
4. Enter a loop that performs the following sequence of instructions until either an error occurs or it has read the entire source file:
   (a) Read a chunk of data from the source file into a buffer and store the number of bytes read into a variable named `num_bytes_read`. If there is no data left, or a read error occurred, break out of this loop.
   (b) Write `num_bytes_read` many bytes of data from the buffer to the target file. If a write error occurred, or the amount of data written is less than `num_bytes_read`, report the error and break out of the loop.
5. Close the source file.
6. Close the target file.
7. Return a value indicating whether or not the program copied the file successfully.

We've already covered everything we need to know to implement that logic, so writing the program will be relatively straightforward. We just have a few decisions to make about the variables our program will need.

### Implementation of copy

The `read()` and `write()` system calls both require a buffer. We need to decide how large that buffer should be, and that decision will affect the program's performance. For now, we'll choose its size somewhat arbitrarily, and after creating the initial version of the program, we'll consider how the buffer size affects its performance. Listing 5-1 shows the complete program.

*spl_cp.c*

```
#define _GNU_SOURCE
#include "../include/common_hdrs.h"
#include "../include/common_defs.h"

#ifndef ❶BUFFER_SIZE
   #define BUFFER_SIZE      4096
#endif
#define MESSAGE_SIZE   512
#define PERMISSIONS  S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH /*rw-rw-r-- */

int main(int argc, char *argv[])
{
    int     source_fd;
    int     target_fd;
    int     num_bytes_read;
    int     num_bytes_written;
    mode_t  permissions = PERMISSIONS;
    char    buffer[BUFFER_SIZE];
    char    message[MESSAGE_SIZE];

    /* Check for correct usage.  */
    if ( argc != 3 ){
        sprintf(message,"%s source destination", basename(argv[0]));
        usage_error(message);
    }

    /* Open source file for reading */
    errno = 0;
    if ( (source_fd = open(argv[1], O_RDONLY)) == -1 ) {
        sprintf(message, "unable to open %s for reading", argv[1]);
        fatal_error(errno, message);
    }

    /* Open target file for writing */
    errno = 0;
    if ( (target_fd = open( argv[2], O_WRONLY|O_CREAT|O_TRUNC,
                            permissions) ) == -1 ) {
        sprintf(message, "unable to open %s for writing", argv[2]);
        fatal_error(errno, message);
    }
    /* Repeatedly transfer BUFFER_SIZE bytes at a time from source_fd to
       target_fd */
      errno = 0;

❷   while ( (num_bytes_read = read(source_fd , buffer, BUFFER_SIZE)) > 0 ){
          errno = 0;
          num_bytes_written = write( target_fd, buffer, num_bytes_read ) ;
```

```
        ❸if ( errno != 0 )
             fatal_error(errno, "copy");
         else
             if ( num_bytes_written  != num_bytes_read) {
                 ❹sprintf(message,"write error to %s\n", argv[2]);
                 fatal_error(-1, message);
             }
         errno = 0;
      }
      if (num_bytes_read == -1)
          fatal_error(errno, "error reading");

      /* Close files */
      errno = 0;
      if ( close(source_fd) == -1 ) {
          sprintf(message, "error closing source file %s", argv[1]);
          fatal_error(errno, message);
      }
      errno = 0;
      if (-1 == fsync(target_fd))  /* Flush data to device. */
          fatal_error(errno, "fsync");
     /* fsync() was successful. */
     errno = 0;
     if ( close(target_fd) == -1 ) {
          sprintf(message, "error closing target file %s", argv[2]);
          fatal_error(errno, "error closing target file");
      }
      return 0;
}
```

*Listing 5-1: A complete implementation of a simple file copy program*

The conditional macro ❶ that defines the buffer size allows us to change the buffer size without having to change the program itself. For example, by entering the command

```
gcc -DBUFFER_SIZE=4096 spl_cp.c  -L ../lib -lspl -o spl_cp
```

to compile and build the executable `spl_cp`, the value `4096` assigned to the symbol `BUFFER_SIZE` on the command line will override the value it's given in the source code. Recall from Chapter 3 that various functions needed by all of our programs were placed into our own static library, named *libspl.a*, in the directory *demos/lib*, whose relative pathname from this chapter's demo directory is *../lib*. That's why the compilation command needs the options `-L../lib -lspl`.

The `while` loop ❷ condition uses a common C paradigm:

```
while (returnvalue = func(...) conditional_operator expression)
```

To evaluate the condition, the function is called and its return value is assigned to *returnvalue*, which is then compared to *expression* using the given *conditional_operator*. If the comparison is true, the loop is entered, otherwise not.

The `while` loop does the main work. The loop is entered each time that the `read()` call transferred one or more bytes to the `buffer`. The loop body attempts to write those bytes to the target file descriptor. The return value of `write()` is checked to see whether the number of bytes transferred equals the number requested by the call. If not, something went wrong and the program exits❹. The program also checks whether `write()` set `errno` to a nonzero value ❸ and exits if it does.

It's time to run the program and see how it works.

### Testing of the copy Program

Let's run the `spl_cp` command to make a copy of its own source code:

```
$ ./spl_cp spl_cp.c spl_cp_backup.c
```

We can check whether *spl_cp* and *spl_cp_backup.c* are identical with the `diff -s` command, which we introduced in Chapter 2:

```
$ diff -s spl_cp.c spl_cp_backup.c
Files spl_cp.c and spl_cp_backup.c are identical
```

In this case it copied the file correctly, but more generally, how do we know whether the program is correct? *Correctness* in this context means:

1. It should make identical copies of the source files every time it's called, regardless of their size.
2. It should report incorrect usage and report errors whenever errors occur.

We can't verify the first condition because that would require running the program on every possible file. However, we can convince ourselves with high probability that the program is correct by running it on as many files as is reasonably possible, with varying sizes, from empty files to extremely large files that might possibly exhaust system resources.

How can we compare two arbitrary files to see if they're identical? We can't use the `diff` command, because it can compare only text files, but the `cmp` command can compare any two files, text or otherwise. For example, we can run `spl_cp` to make a copy of itself and then check whether the copy is identical to the original:

```
$ spl_cp spl_cp spl_cp.bkup
$ cmp spl_cp spl_cp.bkup
$
```

It will display the first position at which they differ if they're not the same or nothing if they're identical. You can use **cmp -l** to see all differences in the two files.

Thus, to convince ourselves of its correctness, we can run our spl_cp program on files of size zero, files of moderate size, and extremely large files, checking with cmp to see whether the copies are the same as the originals, or whether writing fails because there's not enough disk space to make the copies.

To start, here's a run on an empty file:

```
$ du -b /temp/emptyfile    # Show actual size of /temp/emptyfile.
0 /temp/emptyfile
$ spl_cp emptyfile emptyfile.bkup
$ du -b /temp/empty*
0 /temp/emptyfile
0 /temp/emptyfile.bkup
```

There's no need to compare them since they're both empty. A run on a file of medium size:

```
$ du -b /temp/mediumfile
57569256        /temp/mediumfile
$ spl_cp /temp/mediumfile /temp/mediumfile.bkup
$ cmp mediumfile*
$
```

Try running this program on extremely large files and you'll see that it copies them correctly. To test for write errors, I ran it to make a copy of a very large file, a guest virtual machine image file for Ushahidi Ubuntu, which was 13,129,809,920 bytes, on a disk whose capacity was only about 8GB, as follows:

```
$ ./spl_cp /data/Ushahidi-Ubuntu.vmdk  /temp/largefile
write error to /temp/largefile
```

Referring to Listing 5-1, you can see that this error message is the one that's written when errno == 0 after the call to write() and write() wrote fewer than the number of bytes it was supposed to. If we modify the program so that it calls write() one more time after this, errno would be set to ENOSPC and we'd see the message copy: No space left on device.

### *The Universality of the Copy Program*

In the beginning of this chapter I pointed out that the Unix model of I/O is universal. Now that we've written the spl_cp program, we can demonstrate concretely how it works. Open a terminal window, and in bash, navigate to the directory containing the spl_cp executable. Create a small text file in that directory named *testfile*. It doesn't matter how large or small it is, or what it contains. The file I'll work with has the following lines:

```
####         ####
        ^
       ---
```

First, copy *testfile* to the terminal:

```
$ ./spl_cp testfile /dev/tty
####         ####
        ^
       ---
$
```

The contents of *testfile* appeared on the terminal screen because the system calls work on *all* files, and */dev/tty* is a device file that represents the terminal window in which you're working. This shows that the `spl_cp` program copied the file to the terminal.

Now try it the other way around. Enter any text followed by CTRL-D after the command line and then look at the contents of *newfile* with the `cat` command:

```
$ ./spl_cp /dev/tty newfile
####         ####
        ^
       ---
CTRL-D
$ cat newfile
####         ####
        ^
       ---
```

This time, `spl_cp` read the terminal device file and wrote what it contained to a new file named *newfile*. The `read()` system call returns a zero only when it receives a CTRL-D, the keyboard character sequence that sends an end-of-file signal to it; that's why we need to enter that CTRL-D.

You can also use `spl_cp` to send what you enter in one terminal window to another. Open a second terminal window and enter **`tty`** in it. The `tty` command prints the pathname of the terminal's device file. You'll see a string such as `/dev/pts/2`, which is the device file for that terminal window. Enter the following command, substituting the pathname that `tty` printed in your window, and then enter whatever text you like after it, followed by CTRL-D:

```
$ ./copy /dev/tty /dev/pts/2
Hello there.
I'm trying to reach you.
I'm on terminal /dev/pts/1.
Bye.
CTRL-D
```

You should see whatever you typed in one window in the other. In your second window, press ENTER to get the prompt again. It's not magic; it's Unix.

## Timing Programs

The correctness of a program is the most important aspect of its quality, but not the only one. Other measures of program quality relate to how well it performs. Running time is usually the most important performance metric. For example, we'd like to know how fast our `spl_cp` program is at copying files, on average, and how long will it take to copy very large files. This raises the question of how can we measure the amount of execution time that programs take in Unix.

When we researched the man pages in Chapter 4 to investigate time in Unix (in particular, in "About Calendar Time in Unix"), we learned about real time and process times. In that chapter, we weren't interested in process times, so we didn't dwell on them, but now we need to know more about them. The Section 7 man page for `time` mentioned that the `time` command could be used to determine the amount of CPU time consumed during the execution of a program.

The `time` command's man page explains that it can be used to provide data on various system resources used by a program. By default, it displays information about resources besides running time, such as memory usage and I/O activity. To restrict its output to just running times, enter:

```
$ time -p command
```

where *command* is the command whose running time you want to measure. The `-p` option tells `time` to display the traditional POSIX output, which consists of three values, each measured in seconds up to two decimal places:

- *Elapsed clock time*, listed in the output as `real` time
- *User time*, listed as `user` time
- *System time*, listed as `sys` time

Reported `real` time is the number of seconds that elapse from when the command was invoked until it completed. Reported `user` time is the total amount of time that the process and any child processes or their descendants executing on its behalf, spent running in user mode. Reported `sys` time is the total amount of time spent on the process's behalf running within the kernel—that is, in privileged mode, including such time spent by its children as well.

Note that `time` writes its output to the standard error stream rather than the standard output stream. You can supply different options to control the format of the output as well as the kinds of resources about which you'd like `time` to report. The man page contains a detailed list of all of the resource usage that it reports. Also, shells such as `bash` typically define their own version of the `time` command, so you should always type the full pathname of

the `time` program when using it, if you want the non-`bash` version. Since `time` is usually installed in */usr/bin/*, you would enter

```
$ /usr/bin/time -p command
```

For example, I'll run our `ls_cp` program on a relatively large file, a disk image of *Chimera Linux*, a non-GNU Linux, approximately 116MB:

```
$ $ /usr/bin/time -p ./ls_cp chimera-1.9-linux_x86_64.bin  /temp/chimera
real 0.73
user 0.02
sys 0.14
```

This output shows that 0.73 seconds elapsed between when the program started and when it finished, and that it spent about 0.02 seconds running in user mode and about 0.14 seconds running in kernel mode.

Notice that the sum of user and system times is much less than the real time. The real time can never be less than their sum, but it's usually much larger. Processes often spend time waiting for I/O operations to complete. This waiting time is not part of user or system times. When a process issues a request for I/O, it's removed from the CPU until the I/O is complete. We say that the process is *blocked* when it isn't allowed to use the CPU because it doesn't have a required resource to run, which in this case is a completed I/O operation. In addition, when many processes are running, they share the CPU(s) with each other. Each ends up waiting in a queue until it acquires a CPU. This waiting time isn't part of user or system times either. These waiting times account for the difference between elapsed (real) time and the sum of user and system times. Our `ls_cp` program spent 0.73 − 0.16 = 0.57 seconds not on the CPU, either waiting for I/O or the CPU.

Although the amount of time that a process spends waiting depends heavily on what else the system is doing, the more calls it makes, the longer it will take, on average.

When we try to copy larger and larger files, we should expect the process times to become larger. To test this hypothesis, we'll need a set of large files. Also, we'll change the options to the `time` command so that it puts the output on a single line rather than three lines, and sends it to a file instead. The `-f` *`format-string`* option controls both output content and format. We'll use the format string `"\t%e \t%U \t%S"`, which reports real, user, and system times on a single, tab-separated line. We'll send the output to a file with the `-o` *`output-file`* option and ask it to append output to the file instead of overwriting it with the `-a` option.

I created a set of four files, *f1*, *f2*, *f3*, and *f4*, such that *f1* is 60MB and the others double in size successively, and I ran the following `bash` `for` loop:

```
$ for i in 1 2 3 4 ; do
    let size=$((60*(2**(i-1))))
    echo -n $size >> results
    /usr/bin/time -f "\t%e \t%U \t%S" -o results -a  ./copy f${i} f${i}.bk
done
```

The loop uses bash arithmetic operations to calculate the value of size to print and prints that value to the *results* output file, without a trailing newline character. It then copies each of the files into a new file, recording the times in the same *results* file. Table 5-2 shows the results of this experiment.

**Table 5-2:** Process Times in Seconds of the *ls_cp* Program on Four Successively Larger Files

| File Size (MB) | Real Time | User Time | System Time |
|---|---|---|---|
| 60 | 0.32 | 0.00 | 0.08 |
| 120 | 0.75 | 0.01 | 0.13 |
| 240 | 1.47 | 0.03 | 0.26 |
| 480 | 3.07 | 0.06 | 0.50 |

Notice in Table 5-2 that the real and system times increase approximately in proportion to the size of the file over this small dataset, but the user times do not. This isn't a coincidence; the system time is related to the number of system calls that the program makes, and the user time is independent of how many calls it makes. Most of the work is being done in the kernel, not in the program code.

### THE OVERHEAD OF SYSTEM CALLS

System calls in general take much more time than calls to library functions. In "System Calls" in Chapter 3, we summarized the sequence of actions that take place when a program makes a system call. Steps such as copying the arguments of the call to a place that the kernel can access them, trapping to kernel mode, and locating the code to be executed inside the kernel and jumping to it, all take time.

We can get an estimate of how much overhead a system call requires by timing a small program that does nothing other than make a large number of system calls. The `uname()` system call is a relatively small system call that retrieves information about the kernel from data stored internally. It makes few function calls itself inside the kernel and runs pretty quickly. I wrote a small program, (*spl_syscalloverhead.c*) available in the book's source code distribution, that calls `uname()` 100,000,000 times. The `time` command reported that it ran for about 42.25 seconds. As a means of comparison, I replaced the system call in that same program by a call to the GNU C library function `rand()`, which returns a random integer. The `rand()` function is a wrapper for a hidden `__random()` function, which performs some integer arithmetic using a very efficient algorithm for random number generation based on saved state information. It finished in about 1.47 seconds. Both programs were run on a host with a Linux 5.15 kernel. That program (*spl_libcalloverhead.c*) is available in the book's source code distribution as well. The difference in time is primarily due to the overhead required to execute a system call.

### Buffer Size and Running Time

In our *ls_cp.c* program, we chose a buffer size of 4096 bytes. If we increase the buffer size, the program will make fewer system calls to copy a fixed size file. For example, if our file is 4,096,000 bytes, with a buffer of size 4096 bytes, it makes 1000 `read()` and `write()` calls, but if we double the buffer size, it makes 500 calls to each. Decreasing the buffer size increases the number of calls to each. Given that these system calls have overhead, it stands to reason that our programs will run faster with larger buffer sizes. We test this hypothesis by modifying our *spl_cp.c* program so that the buffer size is a command line argument. We named the modified version of the program, *spl_cp2.c*. We can then conduct an experiment in which we run the program with varying size buffers and measure how much time it takes for each buffer size. For each buffer size, we need to run it multiple times and take the average time of the runs. This by itself won't give us a good picture of the effect of buffer size, because the kernel itself performs buffering when transferring data to and from disk devices.

When a user process calls `read()` for data from a disk file, the kernel doesn't transfer the data directly from the disk to the address space of the user process. Instead, it transfers the data from the disk to a storage area in kernel memory, and when all of the data has been transferred, it copies it into the user process's address space.

Symmetrically, when a user process calls `write()` to transfer data to a disk file, the data it sends is copied to a storage area in kernel memory, and at some future time the kernel transfers it to the disk file. This buffering scheme is depicted in Figure 5-4. The kernel's storage area for these disk I/O operations is called its *buffer cache*.

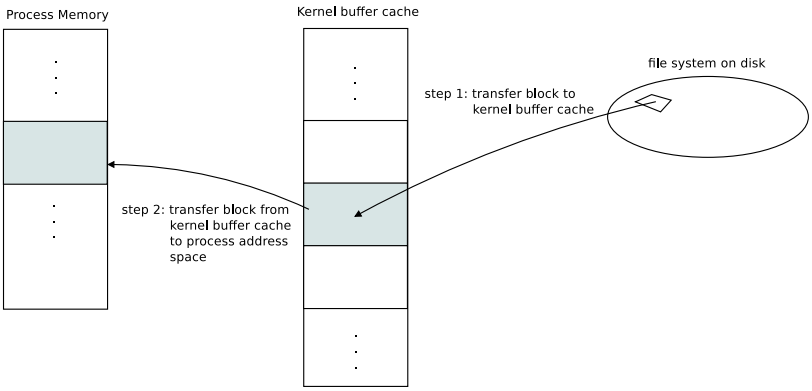



*Figure 5-4: The transfer of data during a `read()` call from the filesystem on disk to the kernel's buffer cache, and then to the process's address space*

The kernel is designed to use this buffer cache to improve overall performance. For one, the `read()` and `write()` calls don't have to wait for the slow disk operations to complete. They return as soon as data is transferred in memory.

Second, on a read request by a process, the kernel searches its buffer cache to see whether the disk data being requested is there. If a buffer is found with that data, it doesn't have to access the disk. Instead, the data is read directly from memory without any physical I/O. Write requests are slightly more complex but similar. In both cases, the average effect is that the kernel spends less time involved in actual physical I/O. In Chapter 14, we'll explain this buffering scheme in greater depth as well as how Linux in particular uses it.

The reason that this system buffering of disk data is relevant is that if our program repeatedly copies the same file to the disk, the Linux kernel will not do the same work each time. Once it reads the input file, it won't have to access the disk each time because it will have all of its data in its buffer cache, provided the machine has enough memory, and once it writes the file once, it won't have to write it again. To prevent this behavior, we'll *unmount* and *remount* the filesystem on which the file resides between runs, which has the effect of emptying the cache data. Filesystem mounting is covered in Chapter 8.

The following `bash` script was used to test the effect of buffer size on the performance of the *spl_cp2.c* program:

```
#!/bin/bash
umount /temp
printf "%s\t%s\t%s\t%s\n" Size Elapsed User System >> $1
for i in 1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536
do
    for j in 1 2 3 4 5 ; do
        mount /temp
        echo -n "$i"  >> resultfile
        /usr/bin/time -f "\t%e \t%U \t%S" -o resultfile -a \
                        ./spl_cp2  /temp/src  /temp/cpy  $i
        umount /temp
    done
done
mount /temp
```

The file and the copy were stored on a filesystem that had no other activity and could be mounted and unmounted easily. The script creates a file in tabular form, which can be imported into a spreadsheet program for further analysis. Table 5-3 displays the results of the experiment.

**Table 5-3:** Effect of buffer size on running time of the copy.c program

| Buffer Size(B) | Real Time | User Time | System Time |
|---|---|---|---|
| 1 | 214.242 | 58.556 | 155.608 |
| 2 | 107.516 | 29.388 | 78.020 |
| 4 | 53.874 | 14.784 | 38.870 |
| 8 | 27.254 | 7.298 | 19.556 |
| 16 | 14.002 | 3.700 | 9.772 |
| 32 | 7.328 | 1.792 | 4.992 |
| 64 | 3.986 | 0.920 | 2.562 |
| 128 | 2.272 | 0.428 | 1.338 |
| 256 | 1.470 | 0.216 | 0.718 |
| 512 | 1.010 | 0.110 | 0.396 |
| 1,024 | 0.816 | 0.046 | 0.252 |
| 2,048 | 0.686 | 0.030 | 0.166 |
| 4,096 | 0.640 | 0.012 | 0.134 |
| 8,192 | 0.636 | 0.006 | 0.112 |
| 16,384 | 0.642 | 0 | 0.108 |
| 32,768 | 0.586 | 0 | 0.106 |
| 65,536 | 0.608 | 0 | 0.108 |

Notice that, for the small buffer sizes, doubling the buffer size roughly halves the system time, but as the buffer size gets larger and larger, the decrease in system time diminishes, and eventually, for the last few sizes, it shows no change. Consider the fact that no matter how many calls to `read()` the program makes, by the time the program finishes, the entire file has to be transferred from the kernel's buffer cache to the program's memory area. The buffer size affects how many transfers are needed, but it doesn't change the total number of bytes to be transferred. In short, the time to transfer the data cannot be diminished by making fewer system calls. The total overhead of the calls becomes smaller as the buffer size grows, but not the total transfer time. The same principle applies to the write operations. It's a law of diminishing returns; the gain in performance obtained with larger buffer sizes is limited by the time it takes to do the transfer operations.

The experiment's results suggest that, for this particular filesystem and host computer, the total time used by the program doesn't change much for buffer sizes larger than 4096. It's certainly clear that buffers smaller than 512 bytes aren't good choices if we're trying to make our program run quickly. In general, the larger the buffer size, the less system call overhead our `spl_cp` program has. On the other hand, if we make the buffer so large that it's larger than the file to be copied, our program will incur needless overhead in the kernel when it tries to allocate unnecessary memory in its buffer cache.

The `cp` command in GNU/Linux is implemented in the GNU *Coreutils* library. In the most recent stable release (9.3) of that library as of this writing, for ordinary files the choice of buffer size is determined at runtime by the `io_blksize()` function, which chooses an appropriate block size for I/O transfers. It defaults to 128KB ($2^{17}$ bytes), much larger than our choice of 4096 bytes!

Finally, although our experiment tells us something about system call overhead, it doesn't measure the effect of the buffer size on actual disk I/O transfer times. We'll explain file I/O in more detail in Chapter 14.

## Summary

Unix employs a simple model of I/O that rests on four pillars — the `open()`, `close()`, `read()`, and `write()` system calls. To transfer data to or from a file, a program opens it, performs the transfer using either `read()` or `write()`, and closes it. This model is universal, in that these same four system calls can be applied to all non-directory files, including device files. A program does not need to call specialized functions to perform I/O on devices.

Every process inherits a file creation mask, also called a umask, that partly determines the permissions assigned to any files that it creates. Users can define the umask given to all of their processes in their `bash` start-up files.

In Unix, a process has four user IDs: a real user ID, an effective user ID, a saved user ID, and, specifically in Linux, a filesystem user ID that serve as its credentials. When a process tries to access a resource such as a file, the kernel makes sure that it has the proper credentials for the type of access it is attempting. Linux kernels use the filesystem user ID to do so and other Unix kernels use the effective user ID. The kernel only grants permission for a resource if the type of access requested is allowed for its user ID. This is how the file permission is used.

When a process opens a file, the kernel creates a data structure called an open file description that represents its connection to that file, and it gives that process an integer file descriptor associated with that description. Files may be opened by multiple processes, and even by the same process multiple times, and each open operation results in a new open file description. All operations on an open file, such as reading and writing, must be given its file descriptor. Every process started in an interactive shell is given three file descriptors, numbered 0, 1, and 2, that refer to the standard input, the standard output, and the standard error. Standard input is connected to the terminal device keyboard and the other two, to its screen.

Processes can open files for reading, writing, or both. The `open()` call allows a process to control other aspects of its connection to the file, such as what to do if a file to be written already exists, and what permissions to assign to newly created files. The `read()` and `write()` system calls each have three parameters: the file descriptor, the memory address of a buffer, and a number of bytes to transfer to or from that buffer, called the buffer size. The choice of buffer size is up to the programmer. In this chapter, we demonstrated how to use the file I/O system calls by implementing an elementary file copying program that can make a copy of a file specified on the command line with a new or existing filename.

Unix provides commands that we can use for measuring the amount of time that our program takes, so that we have a way to improve its performance. The `time` command is one of them. It can report on three different

times associated with a process: the system time, which is time spent in kernel mode, user time, which is time spent in user mode, and real or elapsed time, which is the time elapsed from when the process began to run and when it terminated.

The running time of a process that performs file I/O is greatly influenced by the size of the buffer used for the transfers. In general, larger buffer sizes result in shorter running times, up to a limit, but when we really want to find optimal values, we need to take into consideration other factors, such as the size of the file, the size of the blocks used by the filesystem and kernel for transfers.

## Exercises

1. For each umask below, write the permissions given to the file after the specified command.
   (a) The umask is 024 and the new file is created by the `touch` command.
   (b) The umask is 023 and the file is created by `gcc` when you compile a program successfully.
   (c) The umask is 066. A file named *foo* has permissions rwxrwxrwx. A new file is created with the command **`cp foo foo.copy`**.
2. Rewrite the *spl_cp.c* program so that the buffer size is a command line option of the form "`-B` *`bufsize`*" and it uses a default value of 4096 if the option is not present.
3. Implement a command named `transcript` so that when a user enters **`transcript myfile`**, everything that they type on the keyboard will be displayed both on standard output and in the file named *myfile*. Name the program file *transcript.c*. Hint: The *spl_cp.c* program can be modified to do this.
4. The `cat` command can be used to concatenate files. For example, **`cat f1 f2 f3`** concatenates files *f1*, *f2*, and *f3* and displays their concatenation on standard output. Implement this command. Assume that the total number of files allowed on the command line is ten. Remember that the program does not need to open of close standard output.

# 6

# SOME ADVANCED CONCEPTS OF FILE I/O

In the preceding chapter, we learned the fundamental concepts of file I/O as well as how to use the basic system calls related to it. Here, we'll add a few more advanced tools to our repertoire so we can create more sophisticated programs. First, we'll consider ways to control the position of the file offset, which is the component of an open file description that stores the position of the next byte to read or write in a file. Being able to control the file offset will give us the means to solve problems we currently can't solve that require reading from nonconsecutive parts of files. We'll apply this new knowledge to develop a few programs for displaying various types of information about login records.

All Unix systems record and maintain information about who has logged in, when they did so, and more, in order to answer questions such as who is currently logged in, who has logged in within some past length of time, and

when was the last time that one or more users logged in. We'll examine the files and data structures that store this information as well as the programming interfaces to them.

When we do the background research to write these programs, we'll discover that there are parts of the kernel API that simplify access to particular system databases, and we'll explore what they do and how we can use them. We'll then create a few programs that manipulate some of that data. Finally, we'll discuss a few performance issues and design choices in the programs we developed.

## Controlling the Position of I/O Operations

In Chapter 5, the read operation that we use in the `spl_cp` program is oblivious to the structure of the files it copies; it doesn't matter whether the transfers are aligned with any structural elements in the file. Figure 6-1 illustrates this idea.

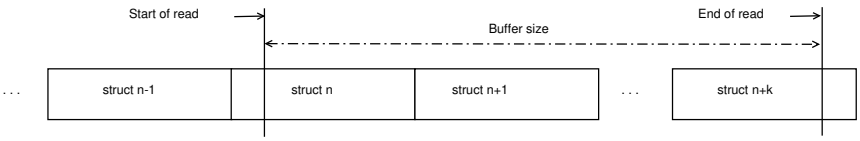



*Figure 6-1: The positions at the start and end of a read operation with respect to structures in a file*

The diagram depicts a portion of a file that consists of a sequence of C `structs` of uniform size. The buffer size in the system call `read(fd,buf, bufsize)` of that program is not chosen to align with any internal structure that the file might have. If the buffer size is not an exact multiple of the size of these structures, the beginning and ending positions of read operations can fall in the middle of these C `structs`. For the `spl_cp` program, this doesn't matter—it ultimately copies all the bytes from one file to another preserving their sequence—but other programs might need to align the starting points of read operations to the starts of these structures.

Programs that need to align their reads and writes with particular offsets in a file need the ability to move the file offset to the position at which they need to perform their next I/O operation.

### The lseek() System Call

When a file is first opened, the file offset is set to the start of the file, unless the `O_APPEND` flag was passed to the `open()` call. We haven't yet discussed the significance of the `O_APPEND` flag, but we'll do so in Chapters 11 and 14. If `read()` is called to read *N* bytes from the file, the file offset is automatically advanced *N* bytes. Similarly, if a write operation writes *N* bytes, the file offset is advanced *N* bytes. We don't control this.

When a program explicitly moves the file offset, it's called *seeking*. Unix kernels provide a system call named `lseek()`, which changes the current file offset's position. Its man page begins with:

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

       off_t lseek(int fd, off_t offset, int whence);

DESCRIPTION
       lseek() repositions the file offset of the open file description
       associated with the file descriptor fd to the argument offset
       according to the directive whence as follows:

       SEEK_SET
              The file offset is set to offset bytes.

       SEEK_CUR
              The file  offset is set to its current location plus offset
              bytes.

       SEEK_END
              The file offset is set to the size  of the file  plus offset
              bytes.
--snip--
```

The `lseek()` system call has three parameters: a file descriptor (`fd`), a distance in bytes (`offset`), and an integer flag (`whence`), which can take on one of the following macro constants: `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`. The `offset` value, which can be given any integer value including negative numbers, is the number of bytes to move the file offset. A positive value moves it forward and a negative value moves it backward. The value of `whence` determines the starting position from which it is to be moved:

**`SEEK_SET`** The file offset moves `offset` bytes relative to the start of the file.

**`SEEK_CUR`** The file offset moves `offset` bytes relative to the current value of the file offset.

**`SEEK_END`** The file offset moves `offset` bytes relative to the end of the file.

Figure 6-2 illustrates how the file offset is adjusted based on the different parameter values.

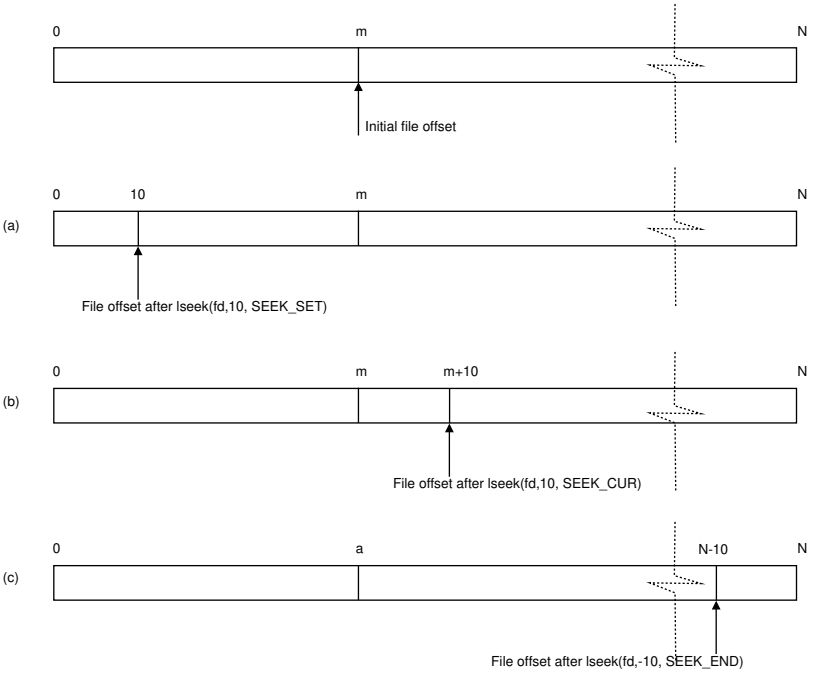



*Figure 6-2: The effect of the whence parameter on the movement of the file offset: in (a) it's moved using whence=SEEK_SET, in (b) using whence=SEEK_CUR, and in (c) using whence=SEEK_END.*

If the resulting value of the file offset would be negative, it's an error, and it isn't moved. Instead, `lseek()` returns -1 and sets `errno` to `EINVAL`. There are several other ways it can fail, and in all cases it returns −1 and sets `errno` to an error value. The man page lists all of the possible errors.

If `lseek()` is successful, its return value is the resulting position of the offset as measured in bytes from the beginning of the file. This return value is useful for two purposes. One is that we can get the current position of the offset with the call:

```
off_t current_pos = lseek(fd,0, SEEK_CUR);
```

This call doesn't move the file offset, and `current_pos` is its current position.

We can also use `lseek()` to get the size of a file by seeking to the end and getting the return value:

```
off_t size = lseek(fd,0, SEEK_END);
```

Since the file offset is now at the end of the file, the return value is the size of the file in bytes, which we store in `size`.

Other examples of using `lseek()` are:

```
lseek(fd, 20, SEEK_SET)    /* Byte 20 of the file */
lseek(fd, -1, SEEK_END)    /* The last byte of the file */
lseek(fd, -1, SEEK_CUR)    /* The byte before the current offset */
```

```
lseek(fd, 0,  SEEK_SET)    /* The first byte in the file */
```

In all of the preceding examples, we didn't try to move the file offset past the end of the file. Let's look at what happens when we do.

### File Holes

Although we can't move the file offset to a position preceding the start of the file, we can move it to a position *after* the end of the file! For example, when the value of `offset` is positive and `whence` is `SEEK_END`, the file offset is moved beyond the end of the file. Data can be written to this position, and this in effect creates a gap in the file between the original end of the file and the start of the data just written. This gap is called a *file hole*. If we then call `lseek(fd,0,SEEK_END)`, the file offset is advanced to the new end of the file.

The `read()` system call doesn't return an error when the file offset is inside a file hole. Instead, it treats the hole as a sequence of null bytes (bytes whose value is zero). To be precise, if the file offset is inside a file hole, the call `read(fd, buffer, count)` fills `buffer` with a null byte for every byte in the hole that it reads. Thus, if all `count` bytes are within the hole and the buffer is at least `count` bytes long, `buffer[0...count-1]` will be filled with zeros. If `count` is large enough that the file offset plus `count` contains data after the hole, then that data will be stored into `buffer` following the zeros. Figure 6-3 illustrates a file hole.

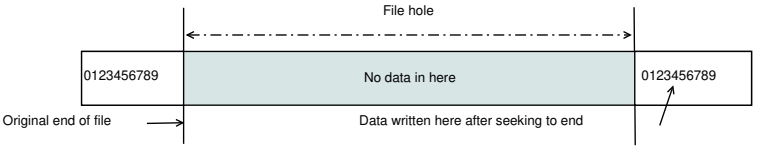



*Figure 6-3: A file hole created by seeking past the end of a file and writing data there*

Figure 6-3 depicts a situation in which a process opened a file consisting of the 10 consecutive characters `0123456789`, after which it performed a seek that moved the file offset 1,000,000 bytes past the end and then wrote the same sequence of characters (`0123456789`) at the new file offset. The file size then became $10 + 1,000,000 + 10 = 1,000,020$ bytes, even though the file has a hole of 1,000,000 bytes within it. Listing 6-1 is a program that creates such a file.

*makefilehole.c*

```
#define MESSAGE_SIZE   512
#define BUFFER_SIZE     10

int main(int argc, char *argv[])
{
    int     fd;
    char    buffer[BUFFER_SIZE];
    char    message[MESSAGE_SIZE];

    if ( 2 > argc ) {
```

```
        sprintf(message, " %s <file-to-create>\n", basename(argv[0]));
        usage_error(message);
    }

    /* Create a new file named file_with_hole in the pwd. */
    if ((fd = open(argv[1], O_WRONLY|O_CREAT|O_EXCL, 0644)) < 0) {
        sprintf(message, "unable to open file %s for writing", argv[1]);
        fatal_error(errno, message);
    }

    strncpy(buffer, "0123456789",BUFFER_SIZE); /* Fill buffer with a string.*/

    /* Write the string at the beginning of the file. */
    if (write(fd, buffer, BUFFER_SIZE) != BUFFER_SIZE)
        fatal_error(errno, "write");

    /* Seek 1,000,000 bytes past the end of the file. */
    if (lseek(fd, 1000000, SEEK_END) == -1)
        fatal_error(errno, "lseek");

    /* Write the small string at the new file offset. */
    if (write(fd, buffer, BUFFER_SIZE) != BUFFER_SIZE)
        fatal_error(errno, "write");

    if ( close(fd) == -1 ) {
        sprintf(message, "error closing file %s\n", argv[1]);
        fatal_error(errno, message);
    }
    exit(EXIT_SUCCESS);
}
```

*Listing 6-1: A program that creates a file with a hole*

We run it to create a file named *file_with_hole* and inspect its size with the command `ls -l file_with_hole`:

```
$ makefilehole file_with_hole
$ ls -l file_with_hole
-rw-r--r-- 1 stewart stewart 1000020 Jun  3 11:57 file_with_hole
```

However, the file doesn't actually contain 1,000,020 bytes. We can see its actual disk allocation with a few different commands. One way is to use `ls -s --block-size=1`. The `-s` option of `ls` shows the number of blocks allocated to the file, and the `--block-size=1` option specifies that the `-s` option should use units of one byte. The command

```
$ ls -s --block-size=1 file_with_hole
8192 file_with_hole
```

shows that this file actually has 8192 bytes. Since disk blocks on the device where the file resides are 4096 bytes (4KB) each, the file is allocated two 4096-byte blocks. We'll explain why it has these two blocks shortly.

A second method of seeing its actual disk allocation is to use the `du` command, which displays disk usage for the files specified as its arguments. It also accepts a `--block-size=1` option to show block size units of one byte:

```
$ du --block-size=1 file_with_hole
8192 file_with_hole
```

Even though the file appears to have a size of 1,000,020 bytes when we use `ls -l`, it's allocated only two disk blocks of 4KB each. Files are allocated storage in fixed-size blocks. A write of $N \leq 4096$ bytes at the start of the file requires one 4KB block. Any remaining bytes of that block are filled with nulls. Since we wrote the second string 1,000,010 bytes from the start of the file, the filesystem allocated a second block for the file. The start of that block must be a multiple of 4096 bytes from the start of the file. The largest multiple of 4096 less than 1,000,010 is $\lfloor 1000010/4096 \rfloor \times 4096$, which is 999,424. Thus the second block starts at byte offset 999,424 in the file. Since 1000010 – 999424 = 586, the start of that second string in the second block is 586 bytes after the start of that block. All bytes preceding that string in that block are filled with zeros, and all bytes after it are filled with zeros. There are no other blocks. The following figure illustrates where the string starts in the file and its relationship to the start of the block.

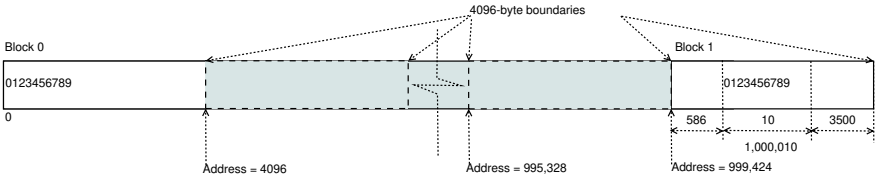



*Figure 6-4: A not-to-scale diagram of the disk blocks allocated to the file with holes that was depicted in Figure 6-3*

The `od` command can show us the actual file contents. We can give it two options: `-a` displays any bytes containing characters as the characters themselves rather than their numeric codes, and `-Ad` displays addresses in decimal instead of the default radix, which is octal:

```
$ od -a -Ad file_with_hole
0000000   0   1   2   3   4   5   6   7   8   9 nul nul nul nul nul nul
0000016 nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
*
1000000 nul nul nul nul nul nul nul nul nul nul   0   1   2   3   4   5
1000016   6   7   8   9
1000020
```

The * notation in the third row indicates that there is a hole. In all other rows, the first column is the decimal address of the first byte in that row.

The remaining fields in a row are the values of the bytes at the 16 successive addresses starting at that address. The `nul` string means that the byte is zero-filled. If the low-order seven bits of a byte represent a character, the character is displayed. That's why we see the actual characters in the output. Even though they look like numbers, they are just characters.

The next time that the output of `ls -l` suggests that a file is of a large size, remember that the actual amount of disk space used by the file might be less. In general, files that appear to be a large size but really use only a small fraction of that size are called *sparse files*.

## Displaying Last Login Information

One problem for which we need to control the position of the file offset is the retrieval and update of login records. Unix systems, like most operating systems, keep track of logins and logouts. They record the times that a user logs in and logs out, as well as other information associated with those events. Ordinary users have permission to read this data; we don't need to be a system administrator or have superuser privilege to see it. The data is structured, usually stored as C structures in binary form in disk files. In order to read, update, or write new records located in particular positions in a file, a program needs to move the file offset to those positions.

We're going to implement a command that reads data from this type of file so that we can get experience in managing the file offset. In particular, we'd like to know which commands print data associated with previous logins on our system, such as the last time that a user logged in or out. We expect that the man page descriptions of such commands will include words such as *login*, *logout*, or *logged*, as well as the word *last*. Although most commands are in Section 1 (Commands) of the man pages, commands used for system administration may also be found in Section 8 (System Management Commands). We'll use `apropos` with the `-a` option to search for all pages containing both *log* and *last* in these two sections:

```
$ apropos -s1,8 -a log last
last (1)             - show a listing of last logged in users
lastb (1)            - show a listing of last logged in users
lastlog (8)          - reports the most recent login of all users or of
                       a given user
pam_lastlog (8)      - PAM module to display date of last login
                       and perform inactive account lock out
```

The output lists three commands of interest: `last`, `lastb`, and `lastlog`. The first two have the same description, and their man pages are worth examining. In fact, they have a shared man page, which states that the `last` command searches back through a file whose typical pathname is */var/log/wtmp* and displays a list of users who have logged in and possibly logged out since the date the file was created. The `lastb` command is similar, but it reports only on bad login attempts. We'll revisit the `last` command later in this chapter.

**THE PATHS.H FILE**

Although the man page tells us that the *wtmp* file is in */var/log/*, it may not be in that location on all systems. The locations of common system files and directories vary from one Unix distribution to another. To allow applications to be written in a portable way, when a Unix system is installed, the system header file */usr/include/paths.h* is populated with macros for the actual pathnames of common system files. For example, it defines `_PATH_WTMP` as a macro name for the pathname of the *wtmp* file:

```
#define _PATH_WTMP "/var/log/wtmp"
```

and `_PATH_MAN` as the pathname to the directory storing the compressed man pages:

```
#define _PATH_MAN "/usr/share/man"
```

Whenever possible, instead of hard-coding actual pathnames in a program, it's better to use the macros instead. This way, if the program is run on a machine in which the system file is in a different location than the one on which we developed the code, it will still locate the file.

If you'd like to use the macros from */usr/include/paths.h*, you can modify the header file *sys_hdrs.h* introduced in Chapter 3 to include that header as well.

### The lastlog Command

The `lastlog` command displays a list of the most recent logins of all users who have ever logged in, or, if a username is given to it, the most recent login of that user. Sample output of the command looks like this:

```
$ lastlog
--snip--
sam          pts/2    192.168.1.112    Wed Apr 19 22:01:56 -0400 2023
lightdm                                    **Never logged in**
nm-openvpn                                 **Never logged in**
brit         pts/1    192.168.1.165    Tue Mar 21 11:20:50 -0400 2023
sshd                                       **Never logged in**
--snip--
```

This command displays the username for every user of the system who has ever logged in, the terminal line on which the login occurred, the host IP address or hostname, whether it was a remote login, and the time and the date of the login. The time is given in the locale's time format, because if I run it changing the locale's `LC_TIME` category to the Spanish language:

```
$ LC_TIME=es_ES.utf8 lastlog
--snip--
sam          pts/2    192.168.1.112    mié abr 19 22:01:56 -0400 2023
```

```
--snip--
```

it displays the date and time in Spanish.

We're going to try to implement the `lastlog` command. Its man page starts with:

```
LASTLOG(8)                System Management Commands                LASTLOG(8)

NAME
       lastlog - reports the most recent login of all users or of a given user

SYNOPSIS
       lastlog [options]

DESCRIPTION
       lastlog formats and prints the contents of the last login log
       /var/log/lastlog file. The login-name, port, and last login time will
       be printed. The default (no flags) causes lastlog entries to be
       printed, sorted by their order in /etc/passwd.

OPTIONS
--snip--
```

After the `OPTIONS` section, there's more detailed information about the command and its database file, which the page states is *var/log/lastlog*. The *paths.h* header file has a macro for this file's pathname:

```
#define _PATH_LASTLOG "/var/log/lastlog"
```

which we'll use when we implement it.

### The lastlog File

The *lastlog file* is a database that contains information about each user's last login. The `lastlog` command accesses records from this file. The page tells us that the database file is sparse, which implies that the file holes in it account for most of its size. We can infer something about the organization of the file from the following remark in the `CAVEATS` section of the page:

```
CAVEATS
       Large gaps in UID numbers will cause the lastlog program to run
       longer with no output to the screen (i.e. if in lastlog database
       there is no entries for users with UID between 170 and 800
       lastlog will appear to hang as it processes entries with UIDs
       171-799).
```

We now know that

- There are no entries for users who have never logged in.

- The `lastlog` command produces no output and appears to hang when it reaches a sequence of user IDs of users who have no entries.
- The file can be sparse.
- The file appears to be large if there are users with high user IDs.

Since the file's size increases as user IDs get higher, and since the command takes time even though it produces no output when there's no record for a user, and since it's sparse, we can infer that the file is like a large array of records for all possible users and such that the location in the file of a user's record is proportional to the user's user ID, as shown in Figure 6-5.

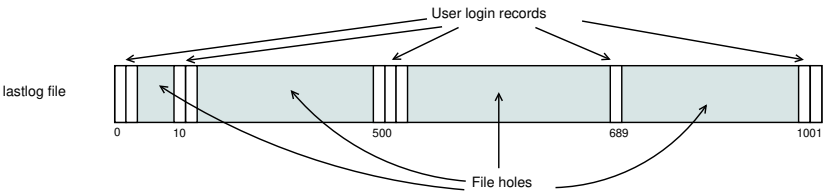



*Figure 6-5: The structure of the* lastlog *file*

Figure 6-5 shows five clusters of user login records, and the rest of the file has no data. In other words, it has file holes. The `lastlog` command spends time in these holes only to discover that the associated users have never logged in.

## The lastlog Structure

The man page doesn't show us each record's structure; for that, we need to do more research. To learn about the form and content of this structure, we check whether there's a man page for the file itself, perhaps in the `FILES` section of the man pages, but there isn't. We then check whether there's a header file that defines the structures. That file would most likely be named *lastlog.h* and be located in */usr/include*. Sure enough, that file exists and has just four lines:

```
$ more /usr/include/lastlog.h
/* This header file is used in 4.3BSD to define `struct lastlog',
   which we define in <bits/utmp.h>.  */

#include <utmp.h>
```

The comment states that the definition of `struct lastlog` is in *bits/utmp.h*. If you try to find a directory named *bits* on your system, you'll discover many of them, and they may not even contain an entry for the *lastlog.h* file.

This leads to a more general question. When we write an include directive in a program such as

```
#include <utmp.h>
```

and there are multiple files named *utmp.h* in the filesystem, which of them does the compiler use?

The compiler searches through a sequence of directories for included files in a well-defined order. Technically, it's not the compiler, but the compiler's preprocessor, `cpp` on GNU/Linux, that defines and uses this sequence. To display the preprocessor's search sequence on a GNU/Linux system, we can run the following command:

```
$ cpp -v /dev/null -o /dev/null
```

The `-v` is a `gcc` option that tells `cpp` to produce *verbose* output. The first argument is the program to preprocess, in this case `/dev/null`, which is an empty file. The `-o /dev/null` tells it to throw away the processed code, which is nonexistent. The command runs the preprocessor on an empty program in verbose mode, outputting its diagnostic report on the standard error stream (the screen) and throwing away the standard output. You'll see many output lines of messages, but there'll be a line starting with `#include <...> search starts here:`. The lines following that line show the search path for included header files. On my system, I see the following:

```
--snip--
#include <...> search starts here:
 /usr/lib/gcc/x86_64-linux-gnu/11/include
 /usr/local/include
 /usr/include/x86_64-linux-gnu
 /usr/include
End of search list.
```

To find the correct *bits/utmp.h* file, we look in each directory in turn, starting with the first. On my system, the first (and only) occurrence of the *bits/utmp.h* file is */usr/include/x86_64-linux-gnu/bits/utmp.h*.

Let's look at that file:

```
--snip--
#ifndef _UTMP_H
# error "Never include <bits/utmp.h> directly; use <utmp.h> instead."
#endif

#include <paths.h>
#include <sys/time.h>
#include <sys/types.h>
#include <bits/wordsize.h>

#define UT_LINESIZE 32
#define UT_NAMESIZE 32
#define UT_HOSTSIZE 256

/* The structure describing an entry in the database of
   previous logins  */
```

```
struct lastlog
  {
❶ #if __WORDSIZE_TIME64_COMPAT32
    int32_t ll_time;
#else
    __time_t ll_time;
#endif
    char ll_line[UT_LINESIZE];
    char ll_host[UT_HOSTSIZE];
  };
--snip--
```

The warning at the top is that a program should never include *bits/utmp.h* directly. Instead it should include *utmp.h*, because that file includes *bits/utmp.h*.

The `lastlog` structure has three members. The first is named `ll_time` and is either of type `time_t` or `int32_t`. The value of the `__WORDSIZE_TIME64_COMPAT32` ❶ macro determines whether the `ll_time` member of the `lastlog` structure is declared as `time_t` or `int32_t`.

How and where is this macro defined? The included file `<bits/wordsize.h>` is most likely where it's defined. Its full path would be */usr/include/x86_64-linux-gnu/bits/wordsize.h*. There, we see yet another conditional macro:

```
#ifdef __x86_64__
# define __WORDSIZE_TIME64_COMPAT32 1
/* Both x86-64 and x32 use the 64-bit system call interface.  */
# define __SYSCALL_WORDSIZE  64
#else
# define __WORDSIZE_TIME64_COMPAT32 0
#endif
```

The macro `__x86_64__` is set to true when the compiler is installed on a 64-bit architecture that is running in *32-bit compatibility mode*. This means that the machine can run applications that consist of instructions for a 32-bit architecture even though it's a 64-bit machine. On such machines, `__WORDSIZE_TIME64_COMPAT32` is set to true (1), otherwise it's set to false (0).

The comment preceding the conditional macro explains the need for it. It ensures that, if the machine is a 64-bit machine that allows 32-bit applications to run, such as an x86-64 or a ppc64, all applications will see `int32_t` as the type of `ll_time`, which is four bytes, and otherwise, they'll all see its type as `__time_t`, which is eight bytes on a 64-bit machine and four bytes on a 32-bit machine. This implies that, when we use the `sizeof()` function in C to obtain the number of bytes in the `struct lastlog` on our system, as in `sizeof(struct lastlog)`, our program will have the correct size, independent of the architecture, implying that we can read the structure safely with a call of the form `read(fd, ll_buffer, sizeof(struct lastlog))`.

**READING STRUCTURES FROM A FILE**

You can read an arbitrary C `struct` such as `struct lastlog` from a file with file descriptor `fd`, into a local variable, say `ll_record` of type `struct lastlog`, using the following code:

```
size_t lastlog_struct_size   = sizeof(struct lastlog);
size_t num_bytes_read = read(fd, &ll_record, lastlog_struct_size);
if ( num_bytes_read < 0 )
    /* read() error - handle it */
else if ( num_bytes_read == lastlog_struct_size) {
    /* Success - can access members with code such as
       printf("%s\n", ll_record.ll_line) */
else
    /* It was an incomplete read.  */
```

You would typically set `errno` to zero before the call in order to determine after the call what, if any, error occurred.

The other two members, `ll_line` and `ll_host`, are character arrays. In the header file just shown, these are of size 32 and 256 respectively. There is no member that stores a user ID because the records are indexed by the user ID, and therefore, the address of a record in the file implicitly gives us the user ID. In other words, letting `N = sizeof(struct lastlog)` be the size of the `lastlog` structure in bytes, if a structure starts at byte address $m \times N$, then it represents the last login of the user whose user ID is $m$.

The `lastlog` structure doesn't have a username member either, so we need to find out how to obtain the username of a user whose user ID is given.

## Usernames, User IDs, and the passwd File

If we enter **`apropos -s2,3 user`** to find either a system call or library function that returns the username associated with a given user ID, we'll get over 100 man pages that match the word *user*. We can inspect the output line by line for a candidate that might work, or we can reduce its size by piping it through the `grep` *filter* command. A filter command reads its input from the standard input, unless it's given a filename argument, in which case it reads from the file, and outputs a modification of its input on the standard output. The `grep` filter expects a string argument. The string can be a pattern, which is best enclosed in single quotes. If it's not a pattern, `grep` searches for that string exactly in every line of its standard input. By default, it outputs only those lines that have a match, so `grep name` will output only those lines containing the word `name`. It acts like a sieve, throwing away lines that don't match. (See Appendix D for more information on the `grep` filter.) Here's the result:

```
$ apropos -s2,3 user | grep name   # Find all entries containing "name"
```

```
attr_list (3)        - list the names of the user attributes of a filesystem ...
attr_listf (3)       - list the names of the user attributes of a filesystem ...
cuserid (3)          - get username
getlogin (3)         - get username
getlogin_r (3)       - get username
getpwnam (3posix)    - search user database for a name
getseuserbyname (3)  - get SELinux username and level for a given Linux username
User::grent (3perl)  - by-name interface to Perl's built-in getgr*() functions
User::pwent (3perl)  - by-name interface to Perl's built-in getpw*() functions
```

The one result that stands out is `getpwnam` in Section 3POSIX of the man pages, since it searches a user database for a name. The three functions whose description is `get username` return only the username of the calling process, which isn't what we want.

The POSIX man page for `getpwnam` is a specification of what the function should do. The relevant fragments of it are:

```
GETPWNAM(3POSIX)           POSIX Programmer's Manual          GETPWNAM(3POSIX)

PROLOG
       This manual page is part of the POSIX Programmer's Manual.  The Linux
       implementation of this interface may differ (consult the corresponding
       Linux manual page for details of Linux behavior), or the interface may
       not be implemented on Linux.

NAME
       getpwnam, getpwnam_r — search user database for a name

SYNOPSIS
       #include <pwd.h>

       struct passwd *getpwnam(const char *name);
       int getpwnam_r(const char *name, struct passwd *pwd, char *buffer,
           size_t bufsize, struct passwd **result);

DESCRIPTION
       The getpwnam() function shall search the user  database  for  an  entry
       with a matching name.
--snip--
```

This seems to be the opposite of what we want. It takes a `name` argument and searches for a matching user record in a database. We want a function that searches that database when it's given a user ID. However, the page gives us a clue: the function returns a pointer to a `struct passwd`. We should investigate this structure. Also, we should scroll down to the `SEE ALSO` section of the page to see whether it mentions other functions that might be more like what we're after. There, it mentions the function `getpwuid()`. Let's look at its man page:

```
GETPWNAM(3)                Linux Programmer's Manual                GETPWNAM(3)

NAME
       getpwnam, getpwnam_r, getpwuid, getpwuid_r - get password file entry

SYNOPSIS
       #include <sys/types.h>
       #include <pwd.h>

       struct passwd *getpwnam(const char *name);
       struct passwd *getpwuid(uid_t uid);
       int getpwnam_r(const char *name, struct passwd *pwd,
                      char *buf, size_t buflen, struct passwd **result);
       int getpwuid_r(uid_t uid, struct passwd *pwd,
                      char *buf, size_t buflen, struct passwd **result);
--snip--
       The getpwuid() function returns a pointer to a structure containing the
       broken-out  fields  of the record in the password database that matches
       the user ID uid.

       The passwd structure is defined in <pwd.h> as follows:

           struct passwd {
               char   *pw_name;       /* username */
               char   *pw_passwd;     /* user password */
               uid_t   pw_uid;        /* user ID */
               gid_t   pw_gid;        /* group ID */
               char   *pw_gecos;      /* user information */
               char   *pw_dir;        /* home directory */
               char   *pw_shell;      /* shell program */
           };

       See passwd(5) for more information about these fields.
--snip--
```

We've found the function we need. The `getpwuid()` function is given a user ID and returns a pointer to a `passwd` structure for the user with that user ID. The `passwd` structure is summarized on this man page, which also refers us to the *pwd.h* header file for its declaration. Furthermore, it tells us that there's a man page in Section 5 named `passwd`. Since Section 5 contains file formats, this is a description of the password file, named *passwd* in Unix (missing the *or* in *password*.)

This man page provides enough information for us to use the `getpwuid()` function, but it's a better idea to learn more about the `passwd` structure and the *passwd* file before continuing.

### The Password Database

In Unix systems, all users have an entry in the password database, whose pathname is always */etc/passwd*. That file is a plaintext file, unlike the *lastlog* file, and it's usually world-readable, so you can view its contents with any of the commands for viewing files, such as **`less /etc/passwd`**. Each line in the file represents a single user account, and in Linux it contains seven fields separated by colons. POSIX.1.2017 requires an implementation to have only five of these fields; the actual password and user information fields are optional. This is what a typical entry looks like:

```
linus:x:501:600:Linus Torvalds:/home/linus:/bin/bash
```

The first field is the username (`linus`). The second is the actual encrypted password, unless it is marked with an `x` to indicate that the actual password is stored elsewhere. The third field is the user ID (`501`), and the fourth is the group ID (`600`). The fifth is traditionally called the *gecos* or *comment* field, and it can contain anything that the system administrator chooses to put there, but it's often used for the user's actual name. The next two fields are the absolute pathname of the user's home directory and their startup shell.

The `passwd` structure and all functions that work with it are declared in */usr/include/pwd.h*. The man page for *pwd.h* lists all of the relevant functions, as shown in Listing 6-2.

```
void           endpwent(void);
struct passwd *getpwent(void);
struct passwd *getpwnam(const char *);
int            getpwnam_r(const char *, struct passwd *, char *,
                     size_t, struct passwd **);
struct passwd *getpwuid(uid_t);
int            getpwuid_r(uid_t, struct passwd *, char *,
                     size_t, struct passwd **);
void           setpwent(void);
```

*Listing 6-2: Functions that set and get password database information*

We're going to explore several of these functions shortly, but first let's see how we can use the `getpwuid()` function.

The argument of `getpwuid()` is of type `uid_t`, which is an integer type. The return value is a pointer to a `passwd` structure. The function may allocate that structure in static memory, which means it can be overwritten by a subsequent call to any of the functions that return a pointer to a `passwd` structure. Therefore, if we want to access the structure's members at a later time, we have to copy the structure to a local variable in the program. If not, we just declare a local pointer variable.

Listing 6-3 illustrates how we can use `getpwuid()` to print the user ID and username of the user running the program.

*getpwuid_demo.c*

```
#include "common_hdrs.h"
#include <pwd.h>
```

```
int main( int argc, char* argv[])
{
    uid_t userid;
    struct passwd  *psswd_struct; /* To save pointer returned by getpwuid() */

    /* Get the real user ID associated with the process,
       which is the same as that of the user who runs this command.         */
    userid = getuid();

    /* To get the user name, we retrieve the password structure
       from the real user ID  using the following function.                 */
    psswd_struct = getpwuid(userid);

    /* Print out the user ID with the name, in the same format as
       the  id command.                                                     */
    printf( "uid=%d(%s)\n", userid, psswd_struct->pw_name);
    return 0;
}
```

*Listing 6-3: A program that displays the user's username and user ID*

We compile and build this program using gcc and run it as follows:

```
$ gcc getpwuid_demo.c -o getpwuid_demo
$ getpwuid_demo
uid=500(stewart)
```

If you do the same on your machine, you'll see your username and user ID.

### Accessing All User Entries

The functions listed in Listing 6-2 have two separate man pages: `getpwnam()`, `getpwnam_r()`, `getpwuid()`, and `getpwuid_r()` are explained in one page, and `setpwent()`, `getpwent()`, and `endpwent()` are explained in the second. These last three functions can be used to iterate through all user entries in the *passwd* file. Their man page explains how this works:

- The `getpwent()` function returns a pointer to a record from the password database. The first time it's called, it returns the first entry; thereafter, it returns successive entries. It returns `NULL` when there are no more entries or if there's an error.
- The `setpwent()` rewinds the password database, so that the next call to `getpwent()` returns the first entry.
- The `endpwent()` function closes the password database.

The man page notes that all three have feature test macro requirements. To use any of them we need to expose their declarations with the appropriate `#define` macros prior to including the header files. Either we define `_XOPEN_SOURCE` with a value at least 500, or on systems with a version of *glibc*

greater than 2.19, we can define `_DEFAULT_SOURCE`, or on systems with older versions of *glibc*, one of `_BSD_SOURCE` or `_SVID_SOURCE`.

It also notes that neither `setpwent()` nor `endpwent()` returns a value, and that both always succeed, so we don't need to check for errors after calling them, however, because `getpwent()` can fail, we do have to check for errors after that call.

From their descriptions, it follows that we can first initialize the iterator with a call to `setpwent()`, and then call `getpwent()` in a `while` loop, terminating the loop when its return value is `NULL`. The program in Listing 6-4 demonstrates this logic.

*showallusers.c*

```
#define _GNU_SOURCE
#define _XOPEN_SOURCE  500
#include "common_hdrs.h"
#include <pwd.h>

int main( int argc, char* argv[])
{
    struct passwd  *psswd_struct;      /* Stores returned record. */

    setpwent();                        /* Initialize the iterator. */
    errno = 0; /* Set errno to 0 to detect error from getpwent(). */

    /* Repeatedly call getpwent() until it returns NULL.          */
    while ( (psswd_struct  = getpwent()) != NULL) {
        /* Print the pw_name member of the struct.                */
        printf("%s\n", psswd_struct->pw_name);
     ❶ errno = 0;
    }
    if ( errno != 0 ) {
     ❷ fatal_error(errno, "getpwent");
    endpwent();                    /* Close the passwd database. */
    return 0;
}
```

*Listing 6-4: A program that displays all usernames in the password database*

First note that the program calls `setlocale()` to print usernames in a locale-sensitive way, just in case it's run on a host computer where some usernames use character sets other than US English. (We discussed locales in Chapter 4.) Also, the program defines `_XOPEN_SOURCE` with a value of 500 to enable the features described in the man page. Finally, note that we repeatedly reset `errno` to zero ❶ in order to check its value ❷ after calling `getpwent()`.

We build the executable with

```
$ gcc -Wall -g  showallusers.c -L ../lib -lspl -o showallusers
```

The following shows a portion of its output, with most lines deleted for brevity:

```
$ showallusers
root
daemon
bin
--snip--
stewart
--snip--
ssd
getpwent: No such file or directory
```

The error message at the very end of the output comes from the program's call to `fatal_error()` ❶.

In Chapter 3, "Handling System Call Errors," we mentioned that the `errno -l` command lists all error messages with their symbolic names, numeric values, and associated message strings. We can run that command, or we can use the `-s` option to search specifically for this message:

```
$ errno -s "No such file or directory"
ENOENT 2 No such file or directory
```

The man page for `getpwent()` did not explicitly list this as a return value, which is why we didn't check for it. It occurs only at the end of the file. If we want to suppress it, we can modify the preceding `if` conditional to be `(errno != 0 && errno != ENOENT)`. Aside from this, it seems that the program's logic can serve as the basis for our implementation of a `lastlog` command.

## Developing a lastlog Program

We'll design and implement a simple version of the `lastlog` command. Initially, it won't accept any command line options. It will print the last login times of all users of the system.

### *Design Considerations*

In the section "Displaying Last Login Information" on page 230, we saw that the *lastlog* file is sparse with large gaps between user records. Figure 6-5 illustrated a hypothetical *lastlog* file, with just a few user records. Because user IDs assigned to users aren't necessarily consecutive numbers, we just can't write a loop such as

```
for ( u = lowest_userid; u < highest_userid; u++)
    process lastlog structure for user u
```

because a user may not exist for a particular value of the index variable `u`, and if we try to process a non-existent record, we'll be in a file hole. In short, this loop processes every valid record in the file, but it also tries to access non-existent ones, so it isn't correct.

We can, however, use the logic from the *showallusers.c* program in Listing 6-4 to iterate over all users and get their user IDs. Specifically, we could use a loop of the following form:

```
setpwent();
while ( (psswd_struct  = getpwent()) != NULL) {
    u = psswd_struct->pw_uid;
    process lastlog structure for user u
}
endpwent();
```

With this loop, the only records we would access would be those of actual users, but it also tries to access login records that may not exist. Some users may never have logged in and therefore would not have a record in the *lastlog* file. Our algorithm must detect this case.

Another lesser problem with this approach is that, if the *passwd* file has not been managed properly, the entries in the file may not be in numerically increasing order of user ID. In this case, the successive reads would bounce back and forth in the *lastlog* file unless we saved the user IDs into an array and sorted them before reading the file. To demonstrate, the following is a very small fragment of the sequence of user IDs in the *passwd* file of a system I log into frequently:

```
...14609,5463,13933,14978,15535,14100,15230,14203,14921,15434,15050,
14567,15414,14431,15508,6187,14903,14010...
```

This list is very unordered, and using these user IDs in the order `getpwent()` returns them would cause a lot of long movements of the file offset in the *lastlog* file. Figure 6-6 illustrates the sequence of reads in a hypothetical machine whose password database is very out of order.

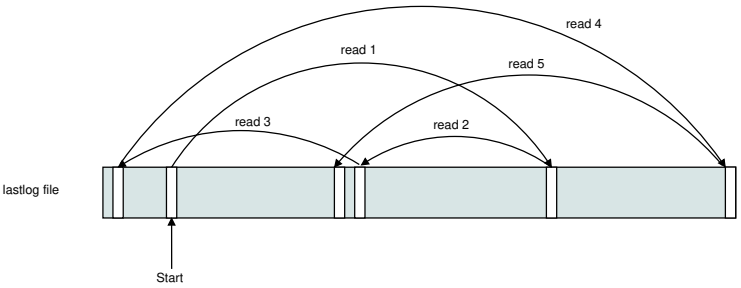



*Figure 6-6: The order of reads in the* lastlog *file using the* passwd *structures returned by successive calls to* getpwent() *on a hypothetical host*

If the *passwd* file spans many disk blocks, the out-of-order reads would be slower than if they were closer together. The problem of out-of-order reads is a performance issue. The question is whether the time required to store all of the user IDs and sort them before reading any records from the file is greater than the time spent seeking because the records delivered by

`getpwent()` are not in order? For this program, we won't sort the records. For a production version, we'd need to study this question further.

We have one other problem. Suppose the highest user ID of all users who have logged in is 1000. This implies that the *lastlog* file's highest entry starts at address 1000×`sizeof(struct lastlog)`. Suppose too that a user in the password database whose user ID is 1200 has never logged in. When the preceding loop tries to read the record for that user, which should be at location 1200×`sizeof(struct lastlog)` in the file, the program will fail with some type of read error, since that address is beyond the end of the file. We need a way to determine whether the record we'd like to process is for a user whose user ID is higher than the highest one in the file. We can solve this problem if we can obtain the size of the *lastlog* file before we start reading. If the file is of size *M* and isn't corrupted, meaning that it contains only complete `lastlog` structures, the highest user ID is (*M*/`sizeof(struct lastlog)`) -1.

We can use `lseek()` to get the size of a file by seeking to the end and saving the returned value of the file offset in a variable:

```
off_t size = lseek(ll_fd,0, SEEK_END);
```

We could also use the `stat()` system call to get the size of the file, but for this program, we'll use `lseek()`.

Finally, we need to consider the matter of how time is represented. The `lastlog` structure's time representation is platform-dependent, as we discovered in "The lastlog Structure" on page 233. The `ll_time` member might be one of two different data types, dependent on the macro condition:

```
#if __WORDSIZE_TIME64_COMPAT32
    int32_t ll_time;
#else
    __time_t ll_time;
#endif
```

The `localtime()` function, which we'll call to convert time to broken-down time, expects a value of type `time_t*`. We can't safely typecast `int32_t*` as an argument to `localtime()`. For example, if `ll_entry` is a `lastlog` struct, writing `localtime((time_t*)&(ll_entry.ll_time))` will fail, because all we'd be doing is casting the pointer type, but the underlying types might be different. Instead, we'd have to assign the entry's time value to a variable of type `time_t`, as in

```
time_t ll_time = ll_entry.ll_time;
```

and pass that variable's address to `localtime()`. Since we need to do this only if `__WORDSIZE_TIME64_COMPAT32` is defined, the code will need to be conditionally compiled based on the value of that macro.

### Program Logic

Let's sketch out the program's logic:

1. Open the *lastlog* file for reading and handle errors.
2. Get the size of the *lastlog* file.
3. Use the size to determine the largest user ID in the *lastlog* file.
4. Enable localization with a call to `setlocale()` so that dates and times are locale-sensitive.
5. Print a header row for the output.
6. For each entry in the password database,
   (a) Get the user ID of this entry.
   (b) Store the username associated with this user ID from the `psswd_struct->pw_name` member.
   (c) Check whether the user ID is no greater than the highest user ID in the file. If it's greater, treat this as the case of a user who never logged in by printing a message that the user never logged in.
   (d) If the user ID is within the bounds, seek to the start of the record in the *lastlog* file for that user ID.
   (e) Read the `lastlog` structure into a temporary variable, `ll_entry`.
   (f) If `ll_entry.ll_time` is `0`, the user never logged in. Print a message that the user never logged in and skip to the next record.
   (g) Either convert the login time stored in `ll_entry.ll_time` to a `time_t` type to pass to `localtime()`, or if it is already of type `time_t`, pass it directly.
   (h) Use the broken-down time returned by `localtime()` in a call to `strftime()` to get a date/time string `lastlog_time` with the default format.
   (i) Print a line on standard output with the user's name, the `ll_entry.ll_line`, the `ll_entry.ll_host`, and the login time, `lastlog_time`.

Implementing most of the preceding steps is straightforward, so our next task is to refine them, after which we can write the program. We'll start with the first step and continue in sequence.

### *Writing the Program*

We know how to open the *lastlog* file and handle the potential error: we'll use the `_PATH_LASTLOG` macro defined in *paths.h* as the pathname of the file, and let `ll_fd` store the file descriptor returned by the call to open the file:

```
errno = 0;
if ( (ll_fd = open(_PATH_LASTLOG, O_RDONLY)) == -1 )
    fatal_error(errno, "while opening "  _PATH_LASTLOG);
```

We'll next get the size of a file with the method we described:

```
off_t ll_file_size = lseek(ll_fd,0, SEEK_END);
```

Then we'll get the largest user ID in the file as follows:

```
size_t ll_struct_size = sizeof(struct lastlog);
int highest_uid = ll_file_size/ll_struct_size - 1;
```

Given a user ID (`uid`), to seek to the start of the `lastlog` structure for that user ID, we simply multiply the user ID by the size of the structure:

```
offset = lseek(ll_fd, ll_struct_size*uid, SEEK_SET);
```

All of the pieces are now in place and we can write the complete program, which appears in Listing 6-5. To save space, the listing does not contain thorough documentation. A fully-documented version is available in the book's source code distribution.

*spl_lastlog.c*

```
#define _GNU_SOURCE
#include "common_hdrs.h"
#include <lastlog.h>               /* For lastlog structure definition       */
#include <paths.h>                 /* For definition of _PATH_LASTLOG        */
#include <pwd.h>                   /* For password file iterators            */

#define MESSAGE_SIZE   512
#define FORMAT        "%c"         /* Default format string                  */
#define READ_ERROR      -2         /* Error reading from lastlog file        */
#define LOCALE_ERROR    -3         /* Error calling setlocale()              */

/* Prints a line for a username who has never logged in. */
void print_never_logged_in(char* uname)
{
    printf("%-16s %-8.8s %-16s **Never logged in**\n", uname, " ", " ");
}

int main(int argc, char *argv[])
{
    struct lastlog ll_entry;        /* To store lastlog record read from file */
    struct passwd  *psswd_struct;   /* passwd structure from password file    */
    int            ll_fd;           /* File descriptor of lastlog file        */
    off_t          ll_file_size;    /* Size of lastlog file, in bytes         */
    size_t         ll_struct_size;  /* Size in bytes of lastlog structure     */
    size_t         num_bytes;       /* Number of bytes read in read()         */
    uid_t          uid;             /* user ID of current search
    char           *username;       /* Username of current search             */
    int            highest_uid;     /* Highest user ID in lastlog file        */
    char           lastlog_time[64];/* Localized date/time string             */
    time_t         ll_time;         /* Lastlog time converted to time_t       */
    struct tm      *bdtime;         /* Broken-down time                       */

    errno = 0;
    if ( (ll_fd = open(_PATH_LASTLOG, O_RDONLY)) == -1 )
```

```
            fatal_error(errno, "while opening "  _PATH_LASTLOG);

        ll_file_size = lseek(ll_fd,0, SEEK_END); /* Get size of  lastlog file.   */
        ll_struct_size = sizeof(struct lastlog); /* Get size of lastlog struct.  */

        highest_uid = ll_file_size/ll_struct_size - 1;
        if ( (setlocale(LC_ALL, "")  == NULL )
            fatal_error( LOCALE_ERROR, "setlocale() could not set the given locale");

        setpwent();  /* Initialize the passwd file iterator. */
        printf("Username         Port     From             Last Login\n");

        while ( (psswd_struct  = getpwent()) != NULL) {
            uid = psswd_struct->pw_uid;
            username = psswd_struct->pw_name;
            if ( uid > highest_uid )
                print_never_logged_in(username);
            else {
                if (lseek(ll_fd, uid * ll_struct_size, SEEK_SET) == -1)
                    fatal_error(errno, "lseek");
                errno = 0;
                if ((num_bytes = read(ll_fd, &ll_entry, ll_struct_size)) <= 0) {
                    if ( 0 != errno ) /* A read error occurred */
                        fatal_error(errno, "read");
                    else { /* Not a read error - shouldn't happen but continue */
                        error_mssge(-1,"could not read the entry, skipping");
                        continue;
                    }
                }
                else if ( num_bytes != ll_struct_size)
                    fatal_error(READ_ERROR, "incomplete read of lastlog struct");

                if (0 == ll_entry.ll_time)  /* No entry for this user */
                    print_never_logged_in(username);
                else {
                    /* Convert the lastlog time into broken-down time. */
#if __WORDSIZE_TIME64_COMPAT32
                    ll_time = ll_entry.ll_time;
                    bdtime = localtime(&ll_time);
#else
                    bdtime = localtime(&(ll_entry.ll_time));
#endif
                    if (bdtime == NULL) /* The only possible error is EOVERFLOW. */
                        fatal_error(EOVERFLOW, "localtime");

                    if (0 == strftime(lastlog_time, sizeof(lastlog_time),
                             FORMAT, bdtime) )
```

```
                    fatal_error(-1, "Conversion to a date-time string failed "
                                " or produced an empty string\n");
                printf("%-16s %-8.8s %-16s %s\n", username, ll_entry.ll_line,
                       ll_entry.ll_host, lastlog_time);
            }
        }
    }
    close(ll_fd);
    exit(EXIT_SUCCESS);
}
```

*Listing 6-5: A complete program that prints the last login times for all users with entries in the password database*

We'll build the executable and run it to see whether its output matches that of the actual `lastlog` command that we displayed in Listing 6. I'll display the same portions of its output as we displayed there:

```
$ spl_lastlog
--snip--
sam              pts/2    192.168.1.105    Wed 19 Apr 2023 10:01:56 PM EDT
lightdm                                    **Never logged in**
nm-openvpn                                 **Never logged in**
brit             pts/2    192.168.1.105    Tue 21 Mar 2023 11:20:50 AM EDT
sshd                                       **Never logged in**
--snip--
```

We can manually compare the output of the actual command and our version of it to see how it differs, running them in separate terminal windows side by side, or we can pipe `lastlog`'s output to the `diff` command, which will output the differences for us:

```
$ lastlog | diff - <(spl_lastlog)
41c41
< sam              pts/2    192.168.1.105    Wed Apr 19 22:01:56 -0400 2023
---
> sam              pts/2    192.168.1.105    Wed 19 Apr 2023 10:01:56 PM EDT
45c45
< brit             pts/2    192.168.1.105    Tue Mar 21 11:20:50 -0400 2023
---
> brit             pts/2    192.168.1.105    Tue 21 Mar 2023 11:20:50 AM EDT
```

Here, `diff` is comparing its standard input stream, coming from `lastlog`, to the `<(spl_lastlog)` pseudo file created by running the `spl_lastlog` command. The `bash` notation `<(`*`command list`*`)` treats the output of *`command-list`* as a filename. The hyphen tells `diff` that its first file argument is the standard input stream.

The only differences we find are in the date/time format. We passed the `%c` format to `strftime()` whereas the Linux implementation passes the format string `"%a %b %e %H:%M:%S %z %Y"`. We can verify this by reading the

source code file *lastlog.c*, found in the *shadow-utils* package (*https://github.com/shadow-maint/shadow*). Other implementations use still other formats. The differences aren't very important.

With our implementation, we can also change the locale and see that `showlastlog` displays the login time in the locale's format:

```
$ LC_TIME=es.ES.utf8 spl_lastlog
--snip--
sam              pts/2    192.168.1.105    mié 19 abr 2023 22:01:56
lightdm                                    **Never logged in**
nm-openvpn                                 **Never logged in**
brit             pts/2    192.168.1.105    mar 21 mar 2023 11:20:50
sshd                                       **Never logged in**
--snip--
```

We don't need to do much testing of this program to see that its output is correct. We also don't need to be concerned too much about its performance, unless it's used on systems with a really large number of users. I leave it as an exercise to determine whether pre-sorting the user IDs will improve performance.

Wrapping up, we chose to implement the `lastlog` command as a way to learn more about how to read from arbitrary positions in a file by seeking to and performing reads at those locations. In the process, we discovered that Unix provides an API for accessing entries in the *passwd* file. That API in effect provides us with an iterator, `getpwent()`, that returns successive password records.

These represent two different paradigms for file access. To obtain data from one file, *lastlog*, we have to explicitly move the file offset with `lseek()` and call `read()` to retrieve the file's data. We call this *explicit seeking and reading*. For the other file, *passwd*, we didn't need to move the file offset or read explicitly, but instead used functions from a small API associated with that file to retrieve data. We call this method *API-based reading*. The differences between them are:

- With explicit seeking and reading, we can control more precisely how and when the transfers are made, than we can by using API-based reading.
- With explicit seeking and reading, the burden is on us to make sure that we haven't made mistakes in accessing the file, and if we don't design the code well, it may perform poorly.
- By using API-based reading, we simplify our programming task considerably because we use code implemented by the library's programmers.
- If the API changes, we might have to make changes to our program, which would not be the case if we were to explicitly read from the file.

Our implementation of `spl_lastlog` was a hybrid approach.

In the remainder of this chapter, we're going to explore other system databases stored in world-readable files that have APIs for accessing their records.

## Developing a last Command

In the section "Displaying Last Login Information" on pages 230 and following, in our search for commands that could display last login times, we came across a command named `last`. There we learned that it extracts information from a file named *wtmp*. We're now going to develop an implementation of that command, along the way learning about the *wtmp* file, the structure of its data, and the various issues related to login records in general. Let's start by reading the man page for `last`, which begins with:

```
LAST(1)                          User Commands                          LAST(1)

NAME
       last, lastb - show a listing of last logged in users

SYNOPSIS
       last [options] [username...] [tty...]
       lastb [options] [username...] [tty...]

DESCRIPTION
       last searches back through the /var/log/wtmp file (or the file
       designated by the -f option) and displays a list of all users logged in
       (and out) since that file was created.
--snip--
```

The page also describes a `lastb` command, but when we read further, we discover that `lastb` only displays what it calls *bad login attempts*, meaning those that failed, and only users with superuser privilege can run it.

It's important to observe that `last` searches *backward* in the *wtmp* file, not forward. The most recent entries are listed first. If we run `last` without any options on a computer that has multiple active user accounts, we'll see output such as:

```
--snip--
l.fishburne pts/1        69.114.124.124   Tue Feb 28 19:19   still logged in
v.gallo     pts/21       104.162.60.115   Tue Feb 28 18:57 - 21:26  (02:29)
s.okonedo   pts/16       173.52.89.136    Tue Feb 28 18:53   gone - no logout
❶ m.sheen     pts/14       172.58.231.252   Tue Feb 28 18:29 - 02:53  (08:24)
csguest     tty3         tty3             Tue Feb 28 18:27 - 19:53  (01:36)
s.buscemi   pts/14       146.95.73.100    Tue Feb 28 18:26 - 18:27  (00:01)
b.pepper    pts/2        24.90.66.208     Tue Feb 28 18:22 - 21:23  (03:00)
❷ m.farrow    pts/9        151.202.41.80    Tue Feb 28 18:11 - 19:40  (1+01:29)
reboot      system boot  5.15.0-57-generi Mon Feb 27 13:56   still running
--snip--
```

```
wtmp begins Sun Jan 1 08:11:55 2023
```

I replaced the actual usernames with fictitious ones to protect the privacy of the actual users. The final line of output is the date of the first entry recorded in the database used by `last`.

If we run `last -x`, we get more system-related events, such as

```
runlevel    (to lvl 5)   5.15.0-71-generi Sat May 13 13:06 - 13:16  (00:10)
reboot      system boot  5.15.0-71-generi Sat May 13 13:06 - 13:16  (00:10)
shutdown    system down  5.15.0-71-generi Sat May 13 10:52 - 13:06  (02:13)
```

Neither the man page nor the Info page describes the individual fields of output, so let's go through what those fields contain. The default output is a sequence of lines, each containing information about either a user login or some type of system activity.

The first (leftmost) field is either a username or a description of a system event, such as `reboot`. For user logins, the next field is an indication of how the user was connected to the system. This can be through a pseudo-terminal (for example, `pts/21`) or through the desktop environment (for example, `tty3`). For user logins, this field is also called the *line* (which is what we called it when we developed the `spl_lastlog` program.) For system events, this field is a description of the event, such as `system boot`. A `runlevel` event is a change in the runlevel of the system. *Runlevels* essentially define the services available to ordinary users. The lowest user runlevel gives the user a terminal interface and no desktop, for example.

The next field indicates from where the user logged in, either the remote host's internet address or sometimes its fully qualified internet name, or for system events, the name of the kernel.

The next fields are, for user logins, the times at which the user logged in and then logged out followed by the total time of that session in the format (*`hours:minutes`*). The width of a line would be too long if `last` displayed both the start time and end time of a login session in the full, date/time format, such as

```
... Tue Feb 28 18:29:22 2023 - Wed Mar 01 15:53:31 2023 ...
```

Instead it omits the seconds in both times and displays only the hour and minute of the ending time:

```
... Tue Feb 28 18:29 2023 - 15:53  ...
```

The nuance is that a login session can end on a day after the one in which it began at a time possibly earlier in the day than the login time, so that the end time is a smaller time value than the start time, such as `Tue Feb 28 18:29 - 02:53` ❶. That second number by itself doesn't tell us that the time was the following day. It could be two, three, or more days later. The session length tells us this information. It can include a count of days, and will look like `(1+01:29)` ❷, meaning one day, one hour, and 29 minutes.

We need to look at the session length to know on which day the end time occurred.

If the session is still active, instead of a time value, the field contains the text, `still logged in`. On rare occasions, and the previous output is such an occasion, instead of the `still logged in` message, the `last` command reports `gone - no logout`. This is the result of `last`'s detecting corrupted information about a user, such as a change during a session in the user's username or user ID, or a user who was not logged out automatically when the system was rebooted or shut down. For system events, this field might contain status information such as `still running` or `crash`.

It is from this field that you can see that the more recent entries precede the older ones in the output.

Unix has a few other, similar commands for listing information about logins. One is the `who` command. Its man page states that it displays information about users who are currently logged in. It's useful on systems in which most people log in remotely and you're interested in knowing whether certain users are logged in. If we run `who`, we see output such as:

```
c.deneuve    pts/2        2023-02-01 00:14 (151.202.41.80)
s.aghdashloo pts/1        2023-02-01 00:20 (73.48.77.155)
n.hawthorne  pts/1        2023-02-01 00:46 (69.114.124.124)
k.russell    pts/1        2023-02-01 09:39 (165.155.132.86)
c.rains      pts/2        2023-02-01 10:10 (146.111.116.2)
```

Here, each line represents a currently active login session. The first column is the username; the second is the device special file of the user's terminal; the third is the time at which that user logged in on that terminal; and the last is the source of the login, either the hostname if it's known, or its internet address. For example, `k.russell` was logged in on terminal line `pts/1`, the session started at 9:39 on Feb. 1, 2023, and the login was initiated from a device with internet address `165.155.132.86`. Unlike `last`, `who` does not show anything about past logins.

### Login Records

To learn more about login records, we first look at the `NOTES`, `FILES`, and `SEE ALSO` sections of `last`'s man page. For `last`, `NOTES` are mostly for system administrators and we can ignore them. The `FILES` section mentions two files: */var/log/wtmp* and */var/log/wtmpb*. The `SEE ALSO` section suggests reading the `wtmp(5)` man page. We'll read that page to learn about the files and data structures related to the command, hoping that they'll contain what we need to write the program. The `wtmp(5)` man page begins with:

```
UTMP(5)                  Linux Programmer's Manual                 UTMP(5)

NAME
       utmp, wtmp - login records
```

```
SYNOPSIS
       #include <utmp.h>

DESCRIPTION
       The  utmp file allows one to discover information about who is
       currently using the system.  There may be more users currently
       using the system, because not all programs use utmp logging.

       Warning:  utmp must not be writable by the user class "other",
       because many system programs (foolishly) depend on its integrity.
       You risk faked system logfiles and  modifications of system files
       if you leave utmp writable to any user other than the owner and
       group owner of the file.

       The  file  is  a  sequence  of  utmp structures, declared as follows in
       <utmp.h> ( note that this is only one of several definitions around;
       details depend on the version of libc):
--snip--
```

We observe first that `utmp` and `wtmp` share a man page and that both *wtmp* and *utmp* are files containing sequences of structures. The warning on the page is directed at system administrators—they should not set the *utmp* file's permissions too weakly because it makes the system vulnerable to attack. The page then describes the content and format of the *utmp* file, which stores information about logins in a sequence of `utmp` structures. The *wtmp* file is also a sequence of `utmp` structures, but they're processed in a slightly different way, which we'll cover shortly.

The `utmp` structure is declared in the *utmp.h* header file. The parenthetical note at the end is important; it tells us that this man page describes just one possible definition of the `utmp` structure, which may be different from that used by other Unix systems. If we design a program based on this definition, it may not run on other Unix systems. Whenever we see this type of warning, we should read the `CONFORMING TO` section of the man page. There, we learn that POSIX.1 does not specify a `utmp` structure, Instead, it defines a `utmpx` structure, whose declaration is exposed by including the *utmpx.h* header file. We need to read about the differences and make a decision. If our program is designed to use the `utmpx` structure, it will be portable to POSIX-compliant systems, but if it based on the `utmp` structure, it may not be.

### HISTORICAL BACKGROUND ON UTMP AND WTMP

The *utmp* and *wtmp* files have been a part of many Unix systems since its beginnings. There was initially a single definition of a `utmp` structure and a collection of functions for accessing and modifying it, in essence a mini-API, which we'll call the *traditional `utmp` API*. This original API had several disadvantages, and so as Unix evolved and diverged, different systems created alternative definitions, and System V and its derivatives introduced a `utmpx`

structure and API. The *x* was added to the name of the structure as well as files containing them so that there were two parallel systems of files and data definitions. Eventually, POSIX made the `utmpx` API part of the standard. This standard included functions for accessing and modifying the structure, which we'll call the *utmpx API*. Linux resolved the discrepancies by providing both the traditional `utmp` and the `utmpx` APIs for accessing the contents of these files. On Linux systems, these two APIs return exactly the same information.

We'll start by exploring the `utmp` structure, because that was the first one historically and also because the `utmpx` structure was derived from it. The remainder of this discussion is specific to Linux.

### The utmp Structure

The man page description of the `utmp` structure begins with the following macros:

```
#define EMPTY          0   /* Record does not contain valid info
                              (formerly known as UT_UNKNOWN on Linux) */
#define RUN_LVL        1   /* Change in system run-level  */
#define BOOT_TIME      2   /* Time of system boot  */
#define NEW_TIME       3   /* Time after system clock changed.  */
#define OLD_TIME       4   /* Time before system clock changed.  */
#define INIT_PROCESS   5   /* Process spawned by init(1)  */
#define LOGIN_PROCESS  6   /* Session leader of user login  */
#define USER_PROCESS   7   /* Normal process  */
#define DEAD_PROCESS   8   /* Terminated process  */
#define ACCOUNTING     9   /* Not implemented  */

#define UT_LINESIZE    32
#define UT_NAMESIZE    32
#define UT_HOSTSIZE    256
```

The first 10 are the possible values of the `ut_type` member of the structure, which defines the type of entry it represents, because Unix systems typically record events besides logins in the file. (We'll explain this shortly in "Logins, Logouts and the utmp and wtmp Files" on page 257. The next three are macros for the sizes, in bytes, of three members of the structure that are strings.

After these, we see the type of the `ut_exit` member of the structure:

```
struct exit_status {            /* Type for ut_exit, below     */
    short int e_termination;    /* Process termination status  */
    short int e_exit;           /* Process exit status         */
};
```

Finally we see the declaration of the `utmp` struct itself:

```
struct utmp {
    short ut_type;              /* Type of record               */
```

```
    pid_t ut_pid;               /* PID of login process         */
    char  ut_line[UT_LINESIZE]; /* Device name of tty - "/dev/" */
    char  ut_id[4];             /* Terminal name suffix         */
    char  ut_user[UT_NAMESIZE]; /* Username                     */
    char  ut_host[UT_HOSTSIZE]; /* Hostname for remote login, or
                                   kernel version for run-level
                                   messages                     */
    struct exit_status ut_exit; /* Exit status of a process
                                   marked as DEAD_PROCESS       */

/* The ut_session and ut_tv fields must be the same size when
   compiled 32- and 64-bit. This allows data files and shared
   memory to be shared between 32- and 64-bit applications.     */
❶ #if __WORDSIZE == 64 && defined __WORDSIZE_COMPAT32
    int32_t ut_session;         /* Session ID (getsid(2)),
                                   used for windowing           */
    struct {
        int32_t tv_sec;         /* Seconds                      */
        int32_t tv_usec;        /* Microseconds                 */
    } ut_tv;                    /* Time entry was made          */
#else
    long   ut_session;          /* Session ID                   */
    struct timeval ut_tv;       /* Time entry was made          */
#endif
    int32_t ut_addr_v6[4];      /* Internet address of remote
                                   host; IPv4 address uses
                                   just ut_addr_v6[0]           */
    char __unused[20];          /* Reserved for future use.     */
};
```

Before digging into the details of this data structure, let's get a sense of the content and purpose of its important members:

**`ut_type`** Indicates the type of the entry—whether it's a login entry, a boot entry, a shutdown entry, and so on.

**`ut_pid`** Stores the process ID of the process that created the entry, which, for login entries, is the user's `login` process.

**`ut_line`** Stores the name of the terminal device of the login, such as *pts/1*, which is called the *line*.

**`ut_id`** A string that's unique to each entry, serving as an identifier for that entry.

**`ut_tv`** Records the time that the record was created (see section"Logins, Logouts and the utmp and wtmp Files" on page 257 for more details).

**`ut_user`** For logins, this contains the username, and for other types of entries, it stores other identifying information.

**ut_host** Stores the name of the remote host from which the connection was made.

Some of the details warrant more explanation. The conditional macro ❶ preceding the declaration of the ut_tv member has the same meaning as the one we saw in "The lastlog Structure" on page 233 in the definition of the struct lastlog. The preceding comment explains why it's there, and a comment in the NOTES section explains further. It ensures that, if the machine is a 64-bit machine that allows 32-bit applications to run, ut_session is four bytes (int32_t) and ut_tv is eight bytes (two int32_t members) for all applications. If it's a 32-bit machine, all applications see these same sizes for these members. If it's a 64-bit host not allowing 32-bit applications to run, these members are larger, since the struct timeval will be 16 bytes, and a long is eight bytes. We'll have to use a feature test macro, as we did in Listing 6-5 in order to access the ut_session and ut_tv members in our final program.

The man page then describes how the entries in the utmp file are created and updated by various processes when you log in and log out, which we'll discuss in "Logins, Logouts and the utmp and wtmp Files" on page 257. It also reiterates the warning:

```
The file format is machine dependent, so it is recommended that it
be processed only on the machine architecture where it was created.
```

The fact that this warning appears twice on the page is not to be overlooked. It implies that we should expect our program to run correctly only on the architecture on which we compile it.

The man page doesn't list any functions specifically tied to the utmp structure definition, but in the SEE ALSO section, it does reference various library functions such as getutent() and getutmp() from Section 3 of the man pages . The page for getutent() has a warning that this function, as well as all others sharing its page, are obsolete in non-Linux systems and that POSIX.1 instead defines a corresponding set of functions with an x in their names, such as getutxent() instead of getutent(). These functions are just aliases for their counterparts without the x, so we're not going to look at the functions in the Linux utmp API. Instead, we'll study and use the Linux utmpx API in order to make our programs more portable.

### *The utmpx API*

Let's find the POSIX.1 specification of the utmpx API with this man page search:

```
$ apropos utmpx
getutmp (3)          - copy utmp structure to utmpx, and vice versa
getutmpx (3)         - copy utmp structure to utmpx, and vice versa
sessreg (1)          - manage utmpx/wtmpx entries for non-init clients
utmpx (5)            - login records
utmpx.h (7posix)     - user accounting database definitions
utmpxname (3)        - access utmp file entries
```

The utmpx.h man page is a POSIX specification of the API. It shows that the POSIX.1 utmpx structure has only six members:

```
char            ut_user[]  /* User login name */
char            ut_id[]    /* Unspecified initialization process identifier  */
char            ut_line[]  /* Device name */
pid_t           ut_pid     /* Process ID */
short           ut_type    /* Type of entry */
struct timeval  ut_tv      /* Time entry was made */
```

Even though POSIX.1 requires fewer members, the Linux utmp man page tells us that "Linux defines the utmpx structure to be the same as the utmp structure." In Linux both structures contain the 10 members defined in the man page. If our programs reference the non-POSIX members, they may not run correctly on non-Linux systems.

The functions listed in the POSIX.1 utmpx.h man page are the following:

```
void          endutxent(void);
struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *);
struct utmpx *getutxline(const struct utmpx *);
struct utmpx *pututxline(const struct utmpx *);
void          setutxent(void);
```

They're declared in the *utmpx.h* header file, which our programs will need to include.

### Logins, Logouts and the utmp and wtmp Files

Both the *utmp* and *wtmp* files are updated when a user logs in and logs out, but they serve different purposes and are processed in different ways. In order to write any program that uses their data, we need to understand how both files are processed.

The *utmp* file stores information about who is currently logged in, but it's also used to record events such as boots, reboots, and changes in the operating system's runlevel. In contrast, the *wtmp* file is an audit file that records not just current logins, but also logouts, as well as boots, reboots, and the same other events as *utmp* does.

When a user logs in, a record for that login is created in both the *utmp* and *wtmp* files. The contents of that record depend on how the user logged in: directly from the machine's GUI desktop, such as the GNOME Desktop Manager (GDM) on a Linux machine, remotely via a network protocol such as SSH, through an XTERM window, and so on. Following is a superficial description of the sequence of actions that take place when a user logs in.

- When a Unix system is booted, after the kernel performs all initializations, enables interrupts, and all other start-up actions, it creates the first user level process, whose process id is 1. This process was traditionally named init, but in Linux it's now named systemd. We'll

call it `init` here, since it is still referred to in the documentation by this name. This `init` process is the ancestor of all user-level processes in a UNIX system: all processes ever created are directly or indirectly created by it. It monitors the activities of all user processes and also manages what takes place when the computer is shut down.

- The `init` process uses information about available terminal devices on the system, such as consoles, modems, network ports, and so on, to create, for each device, a process that will listen for activity on that device. These devices are called *lines*. Some of the listening processes have names like `getty`, `mingetty`, and so on. The name *getty* is short for *get tty*.

> **TTYS**
>
> The term *tty* is short for *Teletype*. A *Teletype* is the precursor to the modern computer terminal. Teletype machines came into existence as early as 1906, but it wasn't until around 1930 that their design stabilized. Teletype machines were essentially typewriters that converted the typed characters into electronic codes that could be transmitted across electrical wires. Modern computer terminals inherit many of their characteristics from Teletype machines.

- Each `getty` process configures the terminal device, displays a prompt such as `login:`, and waits for the user to enter a username and password. Simplifying the rest of what takes place, once the login is authenticated, an entry is created in the *utmp* and *wtmp* files for that login.

Some systems use other means of authenticating logins, such as *Pluggable Authentication Modules (PAM)*, for this purpose. PAM is a library of dynamically configurable authentication routines that can be selected at runtime to do various authentication tasks, not just logins. When a system uses PAM, different software creates the login entries.

Similarly, the handling of network logins is different. These are usually derived from the BSD network login mechanism. Network logins don't use physical terminals, so there's no way to know in advance how many terminals must be initialized. In addition, the connection between the terminal and the computer is a network service, such as SSH or SFTP.

With network logins, `init` creates a process that will listen for the incoming network requests for logins. For example, if the system supports logging in through SSH, then `init` creates a process named `sshd`, the SSH daemon, which in turn creates a new process for each remote login. These new processes will in turn create a pseudo-terminal driver, which then spawns the `login` process that does everything described previously, including creating the *utmp* and *wtmp* entries, but again with slightly different content.

To summarize, there are different paths to the creation of these login records in the two files, and these paths as well as the contents of the record depend on the login method. Regardless of how the login takes place, for

each login, the ut_type of the record is set to USER_PROCESS in both the *utmp* and the *wtmp* files, and the ut_user member is set to the user's username.

When a user logs out, changes are made to both the *utmp* and *wtmp* files, and those changes depend on which processes handled the login entries. The changes made to the *utmp* file are different from the changes made to the *wtmp* file. In the *utmp* file, the login record of the user who is logging out is essentially erased. However, in the *wtmp* file, the process that updates the file appends a new record to it and doesn't modify the user's login record. The content of that record also depends upon on which form of login took place initially, whether through the console, over a network, and so on.

It would be a lot easier to understand if we could display the contents of these binary files. If we had a program that could display their raw contents converted to human-readable form, we'd also be able to debug our implementation of last when we start testing it. In addition, it's a good warm-up exercise for us to write this program. Since both files are sequences of utmpx structures, we'll design the program, which we'll name spl_utmpdump, so that by default it displays the *utmp* file but with the optional command line argument wtmp, displays the *wtmp* file, as in

```
$ spl_utmpdump         # Display the utmp file.
```

or

```
$ spl_utmpdump  wtmp   # Display the wtmp file.
```

The second form will always display the current *wtmp* file. This approach won't allow us to display older *wtmp* files such as */var/log/wtmp.1*. Printing the raw contents of a file in a human-readable format is commonly called *dumping* the file.

### A Program to Show the utmp and wtmp Files

To start, since we'll be writing a few programs that require the *utmpx* header file, we'll append the line

```
#include <utmpx.h>
```

to the *sys_hdrs.h* header file that we defined in Chapter 4. We'll develop the program from the bottom up, starting with the functions that print various pieces of information.

First, we'll write a function, print_ut_type(), partially shown in Listing 6-6 for converting the integer ut_type field to a string such as "USER PROCESS" for the symbolic constant that the integer represents.

*print_ut_type()*

```
void print_ut_type( int t)
{
    switch (t)  {
    case RUN_LVL:       printf("RUN_LVL       "); break;
    case BOOT_TIME:     printf("BOOT_TIME     "); break;
```

```
    --snip--
    case DEAD_PROCESS:  printf("DEAD_PROCESS  "); break;
    case ACCOUNTING:    printf("ACCOUNTING    "); break;
    }
}
```

*Listing 6-6: A function that prints the string associated with each ut_type value*

We make the width of the field large enough to display the longest strings and pad the shorter ones with spaces on the right.

Next, we write a relatively simple function, `print_one_rec()`, that will display the fields of a single `utmpx` structure. The only challenges are in formatting field widths and using the feature test macro that ensures that it compiles correctly for 32-bit and 64-bit architectures, with and without 32-bit application support. We'll pick a field width of nine characters for the username. Many systems allow much longer usernames, so we may want to change this. Listing 6-7 contains the complete function.

*print_one_rec()*

```
void print_one_rec( struct utmpx *utbufp )
{
    struct tm * bdtime;
    char   timestring[64];

    print_rec_type(utbufp->ut_type);
    printf("%-6d ",   utbufp->ut_pid);       /* Process id */
    printf("%-8.8s ", utbufp->ut_user);      /* User name  */
    printf("%-8.8s ", utbufp->ut_id);        /* utmp id    */
    printf("%-8.8s ", utbufp->ut_line);      /* Line       */
❶ #ifdef SHOW_EXIT
    printf("%-3d ",   utbufp->ut_exit.e_exit);
    printf("%-3d ",   utbufp->ut_exit.e_termination);
#endif
    if ( utbufp->ut_host[0] != '\0' )
        printf(" %-18s", utbufp->ut_host);   /* Host    */
    else
        printf(" %-18s", " ");

❷ #if __WORDSIZE_TIME64_COMPAT32
    time_t utmp_time =  utbufp->ut_tv.tv_sec;
    bdtime = localtime(&utmp_time);
#else
    bdtime = localtime(&(utbufp->ut_tv.tv_sec));
#endif
    if (bdtime == NULL)
        fatal_error(EOVERFLOW, "localtime");

   if (0 == strftime(timestring, sizeof(timestring),"%c", bdtime) )
        fatal_error(-1, "Conversion to a date-time string failed "
                    " or produced an empty string\n");
```

```
    printf("%s\n", timestring);
}
```

*Listing 6-7: A function to print a single utmpx record*

The code uses the `printf()` format specifier `%-8.8s` for the username field. In `%-8.8s`, the `-` means *left justify* and the `8.8s` means use exactly eight characters; if a string is smaller, it's padded on the left; if larger, it's truncated on the right. We also conditionally compile ❶ the code that displays the `ut_exit` status, to reduce the width of the output when it's too long to display. If we compile this program using

```
$ gcc -DSHOW_EXIT spl_utmpdump.c ...
```

the exit status fields will be part of the output.

If instead we compile with

```
$ gcc  spl_utmpdump.c ...
```

they'll be omitted.

The feature text macro ❷ is just like the one we used in the `spl_lastlog` program. If the program is compiled on a 64-bit machine that can run 32-bit applications, it converts the time to a `time_t` value through assignment to a variable of type `time_t`, and then passes this variable's address to `localtime()`. Otherwise, it passes the address of the `struct timeval`'s `tv_sec` member directly. Also, notice that we continue to use the `strftime()` function for converting broken-down time to a string representation in a locale-sensitive way.

Next is a little function that prints the header row for the output, which is also conditionally compiled to include or exclude the exit status heading:

*print_header_row()*

```
/* print_header_row  prints a heading for the output.                    */
void print_header_row( )
{
    printf("%-14s%-7s%-9s%-9s%-9s","TYPE","PID","USER","ID","LINE");
#ifdef SHOW_EXIT
    printf("%-9s", "STATUS");
#else
    printf(" ");
#endif
    printf("%-18s%-16s\n","HOST","TIME");
}
```

Field widths are hard-coded into it. If they need to be tweaked later, it would be better to pass them into the function as parameters, using the `printf()` feature that allows field widths to be passed as arguments, as in

```
printf("%-*s", size, str) /* str will be printed left-justified
                             in a field of width size */
```

We're ready to design the main program, which will use the `utmpx` API for reading the `utmpx` records rather than the kernel's `read()` system call. Its outline is:

1. Check whether the user passed `wtmp` as an argument.
2. If so, call `utmpxname(WTMPX_FILE)` to open the *wtmp* file.
3. Otherwise, call `utmpxname(UTMPX_FILE)` to open the *utmp* file.
4. Print a header row using `print_header_row()`.
5. Initialize reading from the file with `setutxent()`, and set `errno` to zero.
6. While `utmpx_entry = getutxent()` is successful call `print_one_rec(utmpx_entry)`.
7. Check whether the loop exited because the end of the file was reached or because `errno` was set, and handle the error in this case.
8. Call `endutxent()` to close the file.

Listing 6-8 contains the complete program with the previously-defined helper functions omitted for to save space.

*spl_utmpdump.c*

```
#define _GNU_SOURCE
#include "common_hdrs.h"

void print_header_row( )       /*  Not shown for brevity */
{ --snip-- }

void print_rec_type( int t)    /*  Not shown for brevity */
{ --snip-- }

void print_one_rec( struct utmpx *utbufp ) /*  Not shown for brevity */
{ --snip-- }

int main(int argc, char* argv[])
{
    struct utmpx *utmp_entry;  /* For returned pointer from getutxent */

    if ( (argc > 1) && (strcmp(argv[1],"wtmp") == 0) ) {
        if ( -1 == utmpxname(WTMPX_FILE) )
            fatal_error(errno, "utmpname()");
    }
    else if ( -1 == utmpxname(UTMPX_FILE) )
        fatal_error(errno, "utmpname()");
    print_header_row();
    setutxent();
    errno = 0;
    while( (utmp_entry = getutxent()) != NULL )
        print_one_rec( utmp_entry );
    if ( 0 != errno )
        fatal_error(errno, "getutxent()");
```

```
    endutxent();
    return 0;
}
```

*Listing 6-8: A program that dumps the* utmp/wtmp *file in a human-readable format*

We'll build the `showutmp` executable defining `SHOW_EXIT`:

```
$ gcc -DSHOW_EXIT spl_utmpdump.c -I../include  -L ../lib -lspl -o spl_utmpdump
```

and run it on our *wtmp* file to see what we can observe from the output.

In the first run, we discover that both fields of the exit status of all records are zeros. To reduce the width of this output, we rebuild `showutmp` without defining `SHOW_EXIT` and run it again, which results in the output that follows.

```
$ ./spl_utmpdump  wtmp
TYPE           PID    USER     ID       LINE     HOST              TIME
RUN_LVL        53     runlevel ~~       ~        5.15.0-71-generic Sat Feb 25 08:11:55 2023
INIT_PROCESS   1993             tty1     tty1                       Sat Feb 25 08:11:55 2023
LOGIN_PROCESS  1993   LOGIN    tty1LOGI tty1                       Sat Feb 25 08:11:55 2023
USER_PROCESS   2297   stewart  :0       tty7     :0                Sat Feb 25 08:13:03 2023
USER_PROCESS   6988   stewart  ts/0stew pts/0    :0                Sat Feb 25 08:33:37 2023
DEAD_PROCESS   6965   stewart  ts/0stew pts/0                       Sat Feb 25 10:50:08 2023
DEAD_PROCESS   0      stewart  :0       tty7     :0                Sat Feb 25 10:52:10 2023
RUN_LVL        0      shutdown ~~       ~        5.15.0-71-generic Sat Feb 25 10:52:21 2023
BOOT_TIME      0      reboot   ~~       ~        5.15.0-71-generic Sat Feb 25 13:06:09 2023
RUN_LVL        53     runlevel ~~       ~        5.15.0-71-generic Sat Feb 25 13:06:32 2023
INIT_PROCESS   1967             tty1     tty1                       Sat Feb 25 13:06:32 2023
LOGIN_PROCESS  1967   LOGIN    tty1LOGI tty1                       Sat Feb 25 13:06:32 2023
USER_PROCESS   2327   stewart  :0       tty7     :0                Sat Feb 25 13:08:27 2023
DEAD_PROCESS   0      stewart  :0       tty7     :0                Sat Feb 25 13:16:30 2023
USER_PROCESS   7210   jl.trint ts/1mero pts/1    24.46.119.86      Wed Mar  1 09:16:43 2023
USER_PROCESS   7342   r.griffi ts/2njia pts/2    146.95.38.217     Wed Mar  1 10:13:57 2023
USER_PROCESS   7348   b.pepper ts/4meli pts/4    24.90.66.208      Wed Mar  1 10:16:08 2023
USER_PROCESS   7367   m.grace  ts/5harm pts/5    148.74.161.63     Wed Mar  1 10:16:21 2023
USER_PROCESS   7389   m.richar ts/6weig pts/6    67.245.64.80      Wed Mar  1 10:16:42 2023
DEAD_PROCESS   0                       pts/5                       Wed Mar  1 10:22:28 2023
DEAD_PROCESS   0                       pts/6                       Wed Mar  1 10:46:29 2023
--snip--
```

Notice that the program prints all of the fields correctly, in suitable column widths. Also, observe that the file stores records other than user login and logout events. Our focus here isn't on those records, but on the `USER_PROCESS` and `DEAD_PROCESS` records.

In the very first `USER_PROCESS` record, the line is `tty7` and the `ID` is `:0`. This corresponds to a login on the computer's desktop. GDM assigns lines of the form `ttyx`, where `x` is a number, to these logins. The `ID` is assigned the value `:0`, which refers to the computer's actual screen display. When I logged out on that day, a logout entry for that same line was written to the file:

```
DEAD_PROCESS  0      stewart  :0       tty7    :0               Sat Feb 25 10:52:10 2023
```

The username field was not erased. The same is true for the login on `pts/0`: when I logged out, in that logout record:

```
USER_PROCESS  6988   stewart  ts/0stew pts/0   :0               Sat Feb 25 08:33:37 2023
DEAD_PROCESS  6965   stewart  ts/0stew pts/0                    Sat Feb 25 10:50:08 2023
```

On the other hand, the logout records for all of the other logins on lines whose names are of the form `pts/x` have had their usernames erased. This difference is exactly what we discussed in "Logins, Logouts and the utmp and wtmp Files on page 257." The man page for `utmp` and `wtmp` only states that "The wtmp file records all logins and logouts. Its format is exactly like utmp except that a null username indicates a logout on the associated terminal." The converse is not true. The entry for a logout on a terminal does not necessarily have a null username.

### *Analysis of the wtmp File*

Our next goal is to use our observations about the *wtmp* file to develop an algorithm for the `last` command. To start, let's ignore the effect of system events on what it displays, concentrating exclusively on user logins and logouts. The command has to find, for each user login record, the matching user logout record.

We'll consider an abstract version of the file, in which we remove all information not relevant to this problem, including the `PID`, `ID`, `HOST`, `TIME`, and the exit status fields of each record. We'll also remove the records that are not either user logins or logouts. We're left with a file such as:

```
USER_PROCESS  stewart          tty7
USER_PROCESS  stewart          pts/0
DEAD_PROCESS  stewart          pts/0
DEAD_PROCESS  stewart          tty7
USER_PROCESS  stewart          tty7
DEAD_PROCESS  stewart          tty7
USER_PROCESS  jl.trint         pts/1
USER_PROCESS  r.griffi         pts/2
USER_PROCESS  b.pepper         pts/4
USER_PROCESS  m.grace          pts/5
USER_PROCESS  m.richar         pts/6
DEAD_PROCESS                   pts/5
DEAD_PROCESS                   pts/6
DEAD_PROCESS                   pts/4
DEAD_PROCESS                   pts/1
DEAD_PROCESS                   pts/2
```

For every `USER_PROCESS` entry, if that user has logged out, there's a unique `DEAD_PROCESS` entry with the same terminal line. Terminal lines are unique—no two users can be logged in at the same time on the same line—and this

implies that if we know the line on which a user logged in, we can find the logout record by searching for a `DEAD_PROCESS` record with the same line that occurs most recently in time after that login.

We can think of this problem in an even more abstract way. Suppose we represent a user login on a line with a unique left bracket. Since the keyboard doesn't have an unlimited supply of different types of left brackets, I'll use a notation consisting of a left square bracket symbol subscripted with a unique integer, such as $[_1$ or $[_2$, to represent a unique left bracket type. The matching logout record will be a right square bracket indexed with same number, $]_1$ or $]_2$ respectively. A sequence of logins and logouts can then be viewed as a string of these brackets with the following constraints:

- For every right bracket $]_x$, there is a matching left bracket $[_x$ to the left of it in the string.
- There are no two occurrences of a left bracket $[_x$ without an intervening occurrence of its matching right bracket $]_x$.
- There are no two occurrences of a right bracket $]_x$ without an intervening occurrence of its matching left bracket $[_x$.
- For every left bracket $[_x$, the leftmost matching right bracket $]_x$ that occurs to the right of that left bracket is the matching right bracket for that left bracket. If no such right bracket exists, it implies that the user whose login is represented by that left bracket has not yet logged out.

If we use the number following the `pts/` as the subscript of the bracket, and $a$ for the subscript of the `tty7` line, then our data would be represented by the string

$$[_a \quad [_0 \quad ]_0 \quad ]_a \quad [_a \quad ]_a \quad [_1 \quad [_2 \quad [_4 \quad [_5 \quad [_6 \quad ]_5 \quad ]_6 \quad ]_4 \quad ]_1 \quad ]_2$$

This abstraction of the *wtmp* data can help us to solve the problem of finding logouts that match logins. It's like searching a string of brackets.

The complication is that the `last` command searches backward through the file, printing the most recent login sessions first. In essence, it travels back in time as it processes the *wtmp* file, because at any given step, the record it has just read has a timestamp that is older than all of the ones it's read before it. By reading the entries in reverse order, it sees newer entries before older ones. Our algorithm needs to do the same, making it a bit harder to understand.

As we process this string in right-to-left order, in effect, we're going back in time. When we see a right square bracket, for example, we know that it should be the matching bracket for some left square bracket that we've yet to see, representing an event that took place earlier in time. We don't know when we'll see it, meaning how far to the left it'll be. Therefore, we have to squirrel this bracket away in a safe place, such as in a linked list, and move on.

We'll solve this problem by creating an initially empty, doubly-linked list, which makes deletions easier. For the sake of precision, let's name it

saved_ut_recs. The idea will be to search the string in right-to-left order, starting from the rightmost entry, marching backward in time toward the beginning. When we find a right bracket, we push it onto the front of saved_ut_recs. When we find a left bracket, we search the saved_ut_recs list starting at its first node for the first occurrence of its matching right bracket. If we find it, we record the login and logout times that it represents and delete the right bracket from the list. If we don't find it, the user whose logout it represents is still logged in.

Notice that we can't use a stack to solve this problem because the brackets can be interleaved, as our sample data showed. We'd have to pop items off of the stack until we found the correct one, and then push back the ones that we removed.

We haven't yet taken into consideration the effect of the records related to runlevel changes, boots, reboots, and shutdowns. When the machine is shut down or rebooted, under normal circumstances, any logged-in users are logged out automatically. The result is that struct utmpx entries of type DEAD_PROCESS are appended to the file for all of these logouts. Therefore, we need to identify shutdown and reboot records.

Shutdown records are those that satisfy the following equalities:

```
ut_type == RUN_LVL
ut_user == "shutdown"
ut_line == "~"
```

Boots and reboots are those that satisfy these equalities:

```
ut_type == BOOT_TIME
ut_user == "reboot"
ut_line == "~"
```

Although there are other types of runlevel change records, we'll ignore them because they don't have an effect on user logins or logouts.

If, as we're reading records backward in the file, we come across a record that represents either a boot, reboot, or shutdown, if there are any remaining DEAD_PROCESS records in the saved_ut_recs list, they cannot be matching logouts for logins that we have yet to find, because any logins that we haven't found yet must have occurred prior to this boot/reboot/shutdown, which means those users would have been logged out prior to that event, not after it. Therefore, when we find such a record, we'll erase all entries from the saved_ut_recs list. That's their only effect on the behavior of our version of last, which we'll name spl_last.

### *Designing the spl_last Program*

We begin by outlining the design of our program, which will accept a single option, -x. If this option is given on the command line, our program's output will include the system shutdown events and runlevel changes. Otherwise, it will display only user logins, logouts, and reboots, just like the actual last command. Our initial version won't attempt to be efficient; it will

read only one utmpx record at a time, even though this takes more time. After we've written this version, we'll consider some changes that would make it more efficient.

These are the initializations that the program will need to perform:

1. Set up option handling by checking whether the -x option was supplied. If so, set a flag show_sys_events to TRUE, and if not set it to FALSE.
2. Open the *wtmp* file. If this fails, exit with a suitable error message, otherwise store the returned file descriptor into fd_wtmp.
3. Set the locale by calling setlocale(LC_TIME, ""). If it fails, display an error message and exit.
4. Read the first entry from the file and save the time in that entry's ut_tv.tv_sec field into time_t start_time for displaying as the final line of output. If the read fails, display an error message and exit.The code snippet should be:

```
errno = 0;
if ( read(fd_utmp, &utmp_entry, utsize) != utsize )
    fatal_error(errno, "read");
start_time = utmp_entry.ut_tv.tv_sec ;
```

5. Create an initially empty list of saved utmpx records, saved_ut_recs.
6. Initialize a time_t variable named last_shutdown_time to 0. This variable will be updated whenever a shutdown event is found in the file. The value 0 indicates that no shutdown record has been found yet.

After the initializations, the program will enter its main processing loop, in which, for every utmpx record in the file, starting with the last (most recent) record, it will process that record. Let's assume for now that we've written a function with the following prototype:

```
int get_prev_utrec(int fd, struct utmpx *ut, int *finished );
```

which, when called the first time, retrieves the last utmpx record in the open file with descriptor fd and stores it into *ut, not modifying the *finished parameter, and when called all subsequent times, gets the record preceding the last one read, unless it has already read the first record in the file, in which case it sets *finished to TRUE.

The main loop is then of the form

```
int done = FALSE;
while ( !done ) {
    if ( get_prev_utrec(fd_utmp, &utmp_entry, &done)  ) {
         code to process the utmp_entry
    else  /* get_prev_utrec() did not read successfully */
        if ( !done )
            fatal_error(2, " read failed");
}
```

When `get_prev_utrec()` reads the first entry in the file, it returns `TRUE`, but when it tries to read again, it returns `FALSE` and sets `done` to `TRUE`.

Let's outline the steps for processing each successfully read `utmp_entry`. Remember that we're travelling back in time with each iteration of the `while` loop.

The first step is to determine the type of `utmp_entry`. For most records, we can determine the type from the `ut_type` field, but as we noted previously, shutdowns are those records for which `ut_type == RUN_LVL`, `ut_user == "shutdown"`, and `ut_line== "~"`. If these conditions are true, the program must set the `utmp_entry.ut_type` field to `SHUTDOWN_TIME`, the `utmp_entry.ut_line` to `"system down"`, and the `utmp_entry.ut_user` to `"shutdown"` before continuing.

The remaining processing is contingent on the value of `utmp_entry.ut_type`. The following list describes what the program should do for each of its possible values.

**`DEAD_PROCESS`** If `utmp_entry.ut_line` is not null, insert `utmp_entry` onto the front of the `saved_ut_recs` list; otherwise, don't insert it, since it does not correspond to any user session.

**`USER_PROCESS`** Search the `saved_ut_recs` list for a record whose `ut_line` is the same as that in `utmp_entry`. If it's found, print a line of output for this login record in which the start time is `utmp_entry.ut_tv.tv_sec` and the end time is the `ut_tv.tv_sec` member of the record found in the list. Also compute the total login time and print it. Finally, delete the saved record from the `saved_ut_recs` list. If no matching record is found, this user login does not have a matching logout. If `last_shutdown_time` > 0, print a line of output with the end time `"gone - no logout"`. If `last_shutdown_time` == 0, the user is still logged in, so print a line of output with end time `"still logged in"`. In either case, the printed start time is `utmp_entry.ut_tv.tv_sec`.

**`BOOT_TIME`** Store the boot time into a variable `last_boot_time` and erase the `saved_ut_recs` list. Print a line of output whose `ut_line` field is `"system boot"` and whose start time is `utmp_entry.ut_tv.tv_sec`. If `last_shutdown_time` == 0, the system has not been shut down since this boot entry was recorded in the file, so the printed end time of this output line should be `"still running"`. Otherwise, the end time should be the current value of `last_shutdown_time`, and the total time should be the time difference `last_shutdown_time - utmp_entry.ut_tv.tv_sec` this value.

**`SHUTDOWN_TIME`** Save the `ut_tv.tv_sec` in this record into the `last_shutdown_time` variable, because for this shutdown, we don't yet know when the most recent preceding reboot took place, and we'll need it when we find that reboot entry, so that when we print the line for that reboot entry, we'll have its end time. We also need to print the end time for this shutdown. The end time of a shutdown event is the time after the shutdown when the system is next rebooted. In other words, the duration of a shutdown event is the time during which the machine is not running. The start time in the output line for a shutdown is when it was shutdown, and its end time is when it was rebooted afterward. That reboot has already been read, and its reboot time was stored into the variable `last_boot_time`.

Therefore, we print a line of output whose start time is `utmp_entry.t_tv.tv_sec` and whose end time is `last_boot_time`, print the total time, and erase the `saved_ut_recs` list, since any saved records in that list cannot match any logins that took place earlier in time than this shutdown event.

When the main loop ends, the program has to clean up a bit and print a final line of output. Cleanup involves freeing dynamically allocated memory in the `saved_ut_recs` linked list and closing the `fd_utmp` file descriptor. The final output line that our program should print should be the same as what the real `last` command prints, which is of the form

```
wtmp begins Thu 01 Jan 1970 12:00:00 AM
```

or something similar based on the user's locale settings. In Chapter 4 we learned how to do this. The following code fragment will work:

```
struct tm bd_start_time;
char   wtmp_start_str[128];
start_time = localtime(&start_time);
if (0 == strftime(wtmp_start_str, sizeof(wtmp_start_str),
             "%a %b %d %H:%M:%S %Y", bd_start_time) )
    fatal_error(BAD_FORMAT_ERROR,
                 "Conversion to a date-time string failed or produced "
                    " an empty string\n");
printf("\nwtmp begins %s\n", wtmp_start_str);
```

An extra newline character precedes the output string so that a blank line appears before the message, just like the real `last` command's output.

#### Support Functions

We still have to implement a few more functions for our program. One of these is the function that reads the file in backward order, whose prototype we defined earlier:

```
int get_prev_utrec(int fd, struct utmpx *ut, int *finished );
```

We also need a function that can compute the total time of a single session (whether it's a login, reboot, or shutdown), convert that time to a number of seconds, minutes, hours, and so on, and format it as a string like the one printed by `last`.

Its prototype will be:

```
void format_time_diff( time_t start_time, time_t end_time, char* time_diff_str);
```

It will compute the total number of seconds in `end_time - start_time` and store its formatted string representation in `time_diff_str`.

We need a printing function as well. When processing `utmpx` records, the program needs to print a line of output for that record with a specific start and end time. We'll consolidate this printing into a function with the prototype:

```
void print_one_line(struct utmpx *ut, time_t end_time);
```

which will print a line of output representing the `utmpx` record `ut` for a session that starts at time `ut.ut_tv.tv_sec` and ends at time `end_time`.

Finally, the program needs some doubly-linked list processing support. The doubly-linked list definition will be

```
typedef struct {
    struct utmpx ut;
    struct utmplist *next;
    struct utmplist *prev;
}  utmplist;

utmplist *saved_ut_recs = NULL;  /* An initially empty list */
```

We'll use three functions for list-related actions: a function to save a `utmpx` record into the list, one that deletes one from the list, and one that erases the entire list. Their prototypes are:

```
void save_ut_to_list(struct utmpx *ut,  utlist **list);
void delete_utnode(utlist* utptr, utlist** list);
void erase_utlist(utlist **list);
```

We'll start with `get_prev_utrec()`. We can't use the `utmpx` API for reading records from the list, because it retrieves them in a forward direction, whereas our function has to get them in the opposite direction.

The first time the function is called, it has to position the file offset at the last record in the file. All other times, it has to position the file offset to the record preceding the one it read in the previous call. Since it needs to do something different in the first call from what it does in subsequent calls, it needs to *remember* which call it's in. For this purpose, we'll use a static local Boolean variable `is_first`, initially `TRUE`, to indicate whether it's in the first call.

Our function also has to be aware of when it's being called to read the first record in the file, at offset zero, so that it doesn't seek to a negative file offset, which causes `lseek()` to fail. It could do this by getting the position of the file offset before it reads, with the call

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

If `current_offset == 0`, this is the first record, so it should set `finished` to `TRUE` to indicate that it should not be called again. The disadvantage of this solution is that it makes an extra call to `lseek()` every time it's called. We can avoid this by maintaining a second static local variable, `saved_offset`, which would save the current value of the file offset prior to the read. It would decrease it in each call by the size of the `utmpx` structure.

The last problem is how to reposition the file offset to read the previous record each time. Figure 6-7 illustrates this.

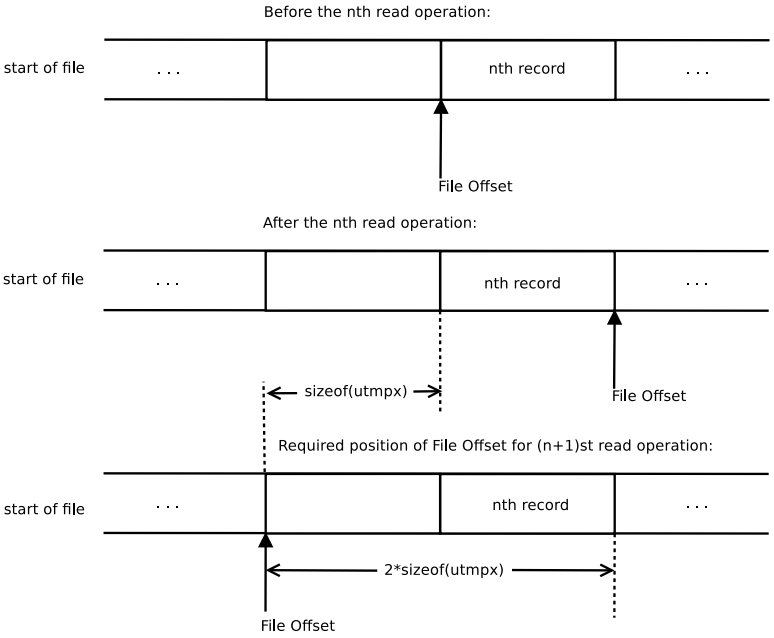



*Figure 6-7: Where the file offset has to be repositioned after a read, when reading backward in the file*

After the `read()` system call reads the *n*th record of the file, the file offset is pointing to the first byte in the file after that record. This position is `2*sizeof(struct utmpx)` bytes past where it has to be in order to read the preceding record. If we've saved the file offset into `saved_offset` before reading, then we just have to decrease `saved_offset` by `sizeof(struct utmpx)` bytes for it to be ready for the next read. All of this logic is incorporated into the function, presented in Listing 6-9.

*get_prev_utrec()*

```
int get_prev_utrec(int fd, struct utmpx *ut, BOOL *finished )
{
    static off_t  saved_offset;    /* Where this call is about to read    */
    static BOOL   is_first = TRUE;   /* Whether this is first time called */
    size_t utsize = sizeof(struct utmpx);  /* Size of utmpx struct        */
    ssize_t       nbytes_read;     /* Number of bytes read                */

    /* Check if this is the first time it is called. If so, move the file
       offset to the last record in the file and save it in saved_offset.*/
    if ( is_first ) {
        errno = 0;
        /* Move to utsize bytes before end of file. */
        saved_offset = lseek(fd, -utsize, SEEK_END);
        if ( -1 == saved_offset )  {
            error_mssge(1, "error trying to move offset to last rec of file");
            return FALSE;
        }
```

```
            is_first = FALSE; /* Turn off flag. */
        }

        *finished = FALSE;       /* Assume we're not done yet. */
        if ( saved_offset < 0 ) {
            *finished = TRUE; /* saved_offset < 0 implies we've read entire file.*/
            return FALSE;    /* Return 0 to indicate no read took place.         */
        }
        /* File offset is at the correct place to read. */
        errno = 0;
        nbytes_read = read(fd, ut, utsize);
        if ( -1 == nbytes_read ) {
            /* read() error occurred; do not exit - let main() do that. */
            error_mssge(errno, "read");
            return FALSE;
        }
        else if ( nbytes_read < utsize ) {
            /* Full utmpx struct not read; do not exit - let main() do that. */
            error_mssge(2, "less than full record read");
            return FALSE;
        }
        else { /* Successful read of utmpx record */
            saved_offset = saved_offset - utsize; /* Reposition saved_offset. */
            if ( saved_offset >= 0 ) {
                /* Seek to preceding record to set up next read. */
                errno = 0;
                if ( -1 == lseek(fd, - (2*utsize), SEEK_CUR) )
                    fatal_error(errno, "lseek()");
            }
            return TRUE;
        }
}
```

*Listing 6-9: The `get_prev_utrec()` function, which reads through the* wtmp *file backward*

The beginning of the function checks whether it's the first time it's called and positions the file offset to the last record in the file. After that, it checks whether the saved offset is negative and if so, it sets `finished = TRUE` and returns to end processing. If it makes it past this point, it reads the record, error handling if need be. If all goes well, it decrements `saved_offset` by the size of the record, and if not negative, moves the file offset to the new position, setting up the next read.

Let's turn to the next function, `format_time_diff()`. Given the starting and ending times, it computes their difference, which is in seconds. From the number of seconds, it does a bit of arithmetic to calculate the equivalent time quantity in seconds, minutes, hours, and days. If the number of days is zero, it formats it one way, and if greater than zero, another, to be consistent with how the actual `last` command behaves. It is shown in Listing 6-10.

*format_time_diff()*

```
void format_time_diff( time_t start_time, time_t end_time, char* time_diff_str)
{
    time_t secs  = end_time - start_time;
    int minutes  = (secs / 60) % 60;
    int hours    = (secs / 3600) % 24;
    int days     = secs / 86400;

    if ( days > 0 )
        sprintf(time_diff_str, "(%d+%02d:%02d)", days, hours, minutes);
    else
        sprintf(time_diff_str, "(%02d:%02d)", hours, minutes);
}
```

*Listing 6-10: The format_time_diff() function*

The third support function is `print_one_line()`, which is shown in Listing 6-11

*print_one_line()*

```
void print_one_line(struct utmpx *ut, time_t end_time)
{
    time_t     utrec_time;
    struct tm *bd_end_time;
    struct tm *bd_ut_time;
    char       formatted_login[MAXLEN];  /* Formatted login date  */
    char       formatted_logout[MAXLEN]; /* Formatted logout date */
    char       duration[MAXLEN];         /* Session length        */
    char       *start_date_fmt = "%a %b %d %H:%M";
    char       *end_date_fmt   = "%H:%M";

    utrec_time = (ut->ut_tv).tv_sec;        /* Get login time, in seconds. */

    /* If the end time is 0 or -1, print the appropriate string
       instead of a time. */
    if ( ut->ut_type == BOOT_TIME && end_time == 0 )
        sprintf(duration, "still running");
    else if ( ut->ut_type == USER_PROCESS && end_time == 0 )
        sprintf(duration, "still logged in");
    else if ( ut->ut_type == USER_PROCESS && end_time == -1 )
        sprintf(duration, "gone - no logout");
    else /* Calculate and format duration of the session. */
        format_time_diff(utrec_time, end_time, duration);

    /* Convert login time to broken-down time. */
    bd_ut_time     = localtime(&utrec_time);
    if (bd_ut_time == NULL)
        fatal_error(EOVERFLOW, "localtime");

    if (0 == strftime(formatted_login, sizeof(formatted_login),
```

```
                        start_date_fmt, bd_ut_time) )
            fatal_error(BAD_FORMAT_ERROR,
                "Conversion to a date-time string failed or produced "
                " an empty string\n");
        /* Convert end time to broken-down time. */
        bd_end_time = localtime(&end_time);
        if (bd_end_time == NULL)
            fatal_error(EOVERFLOW, "localtime");

        if (0 == strftime(formatted_logout, sizeof(formatted_logout),
                 end_date_fmt, bd_end_time) )
            fatal_error(BAD_FORMAT_ERROR,
                        "Conversion to a date-time string failed or produced "
                        " an empty string\n");
        /* Add terminating NULL to host name, otherwise it will be too long. */
        ut->ut_host[sizeof(ut->ut_host)] = '\0';

        /* Print the whole line. */
        printf("%-8.8s %-12.12s %-18s %s - %s %s\n", ut->ut_user, ut->ut_line,
                ut->ut_host, formatted_login, formatted_logout, duration);
}
```

*Listing 6-11: The `print_one_line()` function*

The function prints a single line to standard output, in the same format as the actual `last` command. The in-line comments explain its steps.

The next three listings contain the linked list functions. The first of these adds the given `utmpx` record to the given list.

*save_ut_to_list()*

```
void save_ut_to_list(struct utmpx *ut,  utlist **list)
{
    utlist* utmp_node_ptr;

    /* Allocate a new list node. */
    errno = 0;
    if ( NULL == (utmp_node_ptr = (utlist*) malloc(sizeof(utlist)) ) )
        fatal_error(errno, "malloc");

    /* Copy the utmpx record into the new node. */
    memcpy(&(utmp_node_ptr->ut), ut, sizeof(struct utmpx));

    /* Attached the node to the front of the list. */
    utmp_node_ptr->next  = *list;
    utmp_node_ptr->prev  = NULL;
    if (NULL != *list)
        (*list)->prev = utmp_node_ptr;
    (*list) = utmp_node_ptr;
}
```

The next function removes the node pointed to by p from the given list, and frees the memory allocated for that node.

*delete_utnode()*

```
void delete_utnode(u\tlist* p, utlist** list)
{
    if ( NULL != p->next )
        p->next->prev = p->prev;

    if ( NULL != p->prev )
        p->prev->next = p->next;
    else
        *list = p->next;
    free(p);
}
```

The third function is used to delete the entire list, freeing all of the memory allocated to its nodes.

*erase_utlist()*

```
void erase_utlist(utlist **list)
{
    utlist *ptr = *list;
    utlist *next;

    while ( NULL != ptr ) {
        next = ptr->next;
        free(ptr);
        ptr = next;
    }
    *list = NULL;
}
```

Notice that the parameter is a doubly-indirect pointer. We need this because the list head itself, saved_ut_recs, is modified by the call. If it weren't doubly-indirect, the call erase_utlist(saved_ut_list) would remove its nodes but on return saved_ut_recs would not be null, and the program would then have a dangerous dangling pointer.

We're ready to assemble the program. Since we've already shown the support functions, to reduce the space needed in Listing 6-12, I put stubs in place of these functions, and omitted the lengthy comments. The complete program is in the book's source code distribution.

*spl_last.c*

```
#define _GNU_SOURCE
#include "common_hdrs.h"
#define MAXLEN             128  /* Maximum size of message string  */
#define BAD_FORMAT_ERROR    -1
#define LOCALE_ERROR        -3

/* Some systems define a record type of SHUTDOWN_TIME. If it's not defined
   define it.                                                          */
```

```
#ifndef SHUTDOWN_TIME
    #define SHUTDOWN_TIME 32 /* Give it a value larger than the other types. */
#endif

typedef struct utmp_list{ /* Type of the linked list of utmpx records. */
    struct utmpx ut;
    struct utmp_list *next;
    struct utmp_list *prev;
} utlist;

int get_prev_utrec(int fd, struct utmpx *ut, int *finished );
void format_time_diff( time_t start_time, time_t end_time, char* time_diff_str);
void print_one_line(struct utmpx *ut, time_t end_time);
void save_ut_to_list(struct utmpx *ut,  utlist **list);
void delete_utnode(utlist* p, utlist** list);
void erase_utlist(utlist **list);

int main( int argc, char* argv[] )
{
    struct utmpx  utmp_entry;              /* Read info into here             */
    size_t        utsize = sizeof(struct utmpx); /* Size of utmpx record      */
    int           fd_utmp;                 /* Read from this descriptor       */
    time_t        last_boot_time;          /* Time of last boot or reboot     */
    time_t        last_shutdown_time = 0;  /* Time of last shutdown           */
    time_t        start_time;              /* When wtmp processing started    */
    struct tm    *bd_start_time;           /* Broken-down time representation */
    char          wtmp_start_str[MAXLEN];       /* String to store start time */
    utlist        *saved_ut_recs = NULL;   /* An initially empty list         */
    char          options[] = ":x";        /* getopt string                   */
    int           show_sys_events = FALSE; /* Flag to indicate -x found       */
    char          usage_msg[MAXLEN];       /* For error messages              */

    BOOL          done = FALSE;
    BOOL          found = FALSE;
    char          ch;
    utlist        *p, *next;

    if ( (fd_utmp = open(WTMPX_FILE, O_RDONLY)) == -1 )
        fatal_error(errno, "while opening "  WTMPX_FILE);

    opterr = 0;  /* Turn off error messages by getopt(). */
    while  (TRUE) {
        ch = getopt(argc, argv, options);
        if ( -1 == ch )
            break;
        switch ( ch ) {
        case 'x':
```

```
            show_sys_events = TRUE;
            break;
        case '?' :
        case ':' :
            fprintf(stderr,"Found invalid option %c\n", optopt);
            sprintf(usage_msg, "%s [ -x ]", basename(argv[0]));
            usage_error(usage_msg);
            break;
        }
    }

    if ( NULL == setlocale(LC_TIME, "")  )  /* Set the locale. */
        fatal_error( LOCALE_ERROR, "setlocale() could not set the given locale");

    /* Read the first struct in the file to get the time of the first entry. */
    errno = 0;
    if ( read(fd_utmp, &utmp_entry, utsize) != utsize )
        fatal_error(errno, "read");
    start_time = utmp_entry.ut_tv.tv_sec ;
    while ( !done ) {
        errno = 0;
        if ( get_prev_utrec(fd_utmp, &utmp_entry, &done)  ) {
        if ( (strncmp(utmp_entry.ut_line, "~", 1) == 0) &&
                 (strncmp(utmp_entry.ut_user, "shutdown", 8) == 0) ) {
                utmp_entry.ut_type = SHUTDOWN_TIME;
                sprintf(utmp_entry.ut_line, "system down");
            }
            switch (utmp_entry.ut_type) {
            case BOOT_TIME:
                strcpy(utmp_entry.ut_line, "system boot");
                print_one_line(&utmp_entry, last_shutdown_time);
                last_boot_time = utmp_entry.ut_tv.tv_sec;
                if ( saved_ut_recs != NULL )
                    erase_utlist(&saved_ut_recs);
                break;
            case RUN_LVL: /* Not handled */
                break;
            case SHUTDOWN_TIME:
                last_shutdown_time = utmp_entry.ut_tv.tv_sec;
                if ( show_sys_events )
                    print_one_line(&utmp_entry, last_boot_time);
                if ( saved_ut_recs != NULL )
                    erase_utlist(&saved_ut_recs);
                break;
            case USER_PROCESS:
                found = 0;
                p = saved_ut_recs; /* start at beginning */
```

```
                    while ( NULL != p ) {
                        next = p->next;
                        if ( 0 == (strncmp(p->ut.ut_line, utmp_entry.ut_line,
                            sizeof(utmp_entry.ut_line)) ) ) {
                               print_one_line(&utmp_entry, p->ut.ut_tv.tv_sec);
                            found = 1;
                            delete_utnode(p, &saved_ut_recs);
                        }
                        p = next;
                    }
                    if ( !found ) {
                        if ( last_shutdown_time > 0 )
                            print_one_line(&utmp_entry, (time_t) -1);
                        else
                            print_one_line(&utmp_entry, (time_t) 0);
                    }
                    break;
                case DEAD_PROCESS:
                    if ( utmp_entry.ut_line[0] == 0 )
                        /* There is no line in the entry, so skip it. */
                        continue;
                    else
                        save_ut_to_list(&utmp_entry, &saved_ut_recs);
                    break;

                case OLD_TIME:      /* Not handled */
                case NEW_TIME:      /* Not handled */
                case INIT_PROCESS:  /* Not handled */
                case LOGIN_PROCESS: /* Not handled */
                    break;
                } /* End of switch */
            }
            else /* get_prev_utrec() did not read correctly. */
                if ( !done )
                    fatal_error(2, " read failed");
        }
        erase_utlist(&saved_ut_recs);
        close(fd_utmp);

        bd_start_time = localtime(&start_time); /* Convert to broken-down time. */
        if (0 == strftime(wtmp_start_str, sizeof(wtmp_start_str),
             "%a %b %d %H:%M:%S %Y", bd_start_time) )
        fatal_error(BAD_FORMAT_ERROR, "Conversion to a date-time "
                    "string failed or produced an empty string\n");
        printf("\nwtmp begins %s\n", wtmp_start_str);
        return 0;
    }
```

*Listing 6-12: An implementation of the `last` command, with stubs for the previously defined support functions*

The `main()` function consolidates the logic we discussed previously. When it sees `DEAD_PROCESS` records, it inserts them into the list, provided that their `ut_line` field is non-empty. When it sees `USER_PROCESS` records, it searches the list from the beginning for a record whose `ut_line` matches, deletes it from the list, and prints a line on output. If it doesn't find a matching record, either the user is still logged in or was never logged out properly. It checks which occurred (`last_shutdown_time > 0`) and prints accordingly. When it sees a `SHUTDOWN_TIME` or a `BOOT_TIME` record, it erases the list of saved records after printing a line of output.

Here's a sample of the program run without options:

```
$ ./last
--snip--
o.isaac  pts/3        100.2.79.16        Mon Jul 03 22:35 - 22:49 (00:13)
w.housto pts/0        104.162.60.115     Mon Jul 03 10:45 - 23:24 (12:38)
szhang44 pts/1        104.162.60.115     Sun Jul 02 18:47 - 15:53 (5+21:05)
d.moore  pts/3        71.249.97.95       Sun Jul 02 12:15 - 12:41 (00:26)
d.moore  pts/0        104.162.60.115     Sun Jul 02 11:16 - 00:38 (13:22)
s.morton pts/0        49.43.217.131      Sun Jul 02 02:07 - 04:59 (02:52)
s.morton pts/4        49.43.217.131      Sat Jul 01 22:54 - 03:05 (04:11)
s.morton pts/3        49.43.217.131      Sat Jul 01 22:44 - 00:57 (02:12)
a.serkis pts/3        70.23.200.201      Sat Jul 01 19:05 - 19:07 (00:02)
o.isaac  pts/1        104.162.60.115     Sat Jul 01 17:05 - 18:47 (1+01:42)
d.hopper pts/0        104.162.60.115     Sat Jul 01 10:40 - 00:14 (13:33)

wtmp begins Sat Jul 01 00:27:35 2023
```

This machine is almost never rebooted since it serves as a gateway for an internal network, so there are no reboot entries, but once a month its *wtmp* file is cleared, which is why the file begins on the first of the month.

Here's a run on a different host with the `-x` option:

```
sweiss   pts/1        146.95.214.131     Sun Jul 16 12:50 - 13:02 (00:12)
sweiss   pts/1        146.95.214.131     Sun Jul 16 12:44 - 12:45 (00:01)
--snip--
n.rapace pts/0        146.95.214.131     Wed Feb 01 00:47 - 00:49 (00:02)
n.rapace pts/0        146.95.214.131     Tue Jan 31 22:12 - 23:05 (00:53)
a.george pts/0        146.95.214.131     Tue Jan 31 14:46 - 17:07 (02:21)
p.liant  pts/0        146.95.214.131     Wed Jan 25 20:15 - 20:19 (00:04)
w.beatty pts/0        146.95.214.131     Fri Jan 20 09:39 - 09:40 (00:00)
root     pts/0        146.95.78.229      Thu Jan 19 12:49 - 12:50 (00:00)
root     pts/0        146.95.78.229      Wed Jan 18 16:11 - 16:11 (00:00)
csguest  tty2         tty2               Tue Jan 10 13:42 - 13:42 (00:00)
a.viole  pts/1        146.95.214.131     Tue Jan 10 13:32 - 13:32 (00:00)
csguest  tty2         tty2               Tue Jan 10 13:30 - 13:42 (00:11)
```

```
reboot   system boot  5.15.0-57-generic  Tue Jan 10 13:30 - 19:00 still running
shutdown system down  5.15.0-57-generic  Tue Jan 10 13:30 - 13:30 (00:00)
reboot   system boot  5.15.0-57-generic  Tue Jan 10 13:27 - 13:30 (00:02)
shutdown system down  5.15.0-43-generic  Tue Jan 10 13:27 - 13:27 (00:00)
csguest  tty3         tty3               Tue Jan 10 13:22 - 13:22 (00:00)

wtmp begins Tue Jan 10 13:15:47 2023
```

This output shows that the host had a lot of system activity on a single day. It also shows that someone logged into it on a console as a guest on that same day, perhaps in order to reboot the machine.

### User Space Buffering of Input

Our implementation reads one `utmpx` record at a time, which is not efficient, as we explained in Chapter 5. The implementation of the actual `last` command reads much larger chunks of the file at a time, but as a result its logic is more complex. Because our primary objective in this chapter was to learn how to manipulate file offsets without making the problems overly complex, we didn't attempt to add user buffering to the programs.

Before we leave this chapter though, we should explore user space buffering of input. The kernel buffers its reads and writes, as we discussed earlier, but our programs can also explicitly buffer input. Suppose that, instead of reading one record at a time, our program reads many records at a time. If a record is $N$ bytes and the kernel reads 4096 bytes at a time from the disk, we can reduce the number of system calls our program makes by reading the largest number of records that fit into a 4096 byte block, or $M = \lfloor 4096/N \rfloor$ records each time. The program's performance would improve, but we'd have to solve a few new problems.

The records would have to be stored into an array in our program's local memory, and we would need functions to get and remove the next record of the array and reload the array when it was empty. Since we're reading backward in the file, we'd have to read the records in backward order from the array. In other words, if we read $M$ records at a time from the file into an array declared as `struct utmpx utrecs[M]`, we'd have to retrieve them from the array in the order `utrecs[M-1]`, `utrecs[M-2]`, `utrecs[M-3]`, ... `utrecs[0]`. The logic for processing records does not change, and the logic for reading backward is similar to that of the `get_prev_utrec()` function, except that, instead of moving the file offset with `lseek()`, we'd use an integer variable that points to the next index in the array from which to retrieve a record. When that variable reaches 0, the program would have to reload the array from the file.

Let's formalize an interface that we could use for user-buffered input from the *wtmp* file. It needs just a few functions:

```
#define NRECS   16
#define UTSIZE  (sizeof(struct utmpx))

/* Open the wtmp file specified by filename, obtaining a file descriptor,
```

```
   and if successful, allocate storage for a buffer *utbuf of size
   NRECS*UTSIZE (large enough to store NRECS utmpx structures).
   Return the file descriptor if successful, -1 on failure.*/
int init_wtmp( char *filename, struct utmpx **utbuf);

/* Return a pointer to the next utmpx structure to process from
   the utbuf buffer at index next_ut, decrementing next_ut. */
struct utmpx *get_next_utrec(struct utmpx *utbuf, int *next_ut);

/* Try to read the next NRECS utmpx structures from fd_utmp into
   the buffer utbuf starting at the beginnning of the buffer.
   Return the number actually read, or -1 if  reading failed. */
int load_buf(int fd_utmp, struct utmpx *utbuf);

/* Free all memory used by the buffer utbuf and close the file descriptor
   fd_wtmp. */
void wtmp_finalize(int fd_wtmp, struct utmpx **utbuf);
```

The program would call `init_wtmp()` to open the file and allocate an array that can hold `NRECS utmpx` structures. If this is successful, it would call `load_buf()` to read up to `NRECS` records from the file, starting `NRECS*UTSIZE` bytes before the end of the file, into the buffer. Subsequent calls to `load_buf()` would read that many bytes from the position `NRECS*UTSIZE` bytes before the previous call.

When the program reaches the beginning of the file, reloading may load fewer than `NRECS` records. It needs to check the return value of this call so that it knows where in the buffer to get the next `utmpx` record. Of course if the return value is zero, it means there's nothing left to read.

Each time the program is ready to process the next `utmpx` record, it calls `get_next_utrec()`. It starts at the highest index in the array containing a valid record, which would be the return value of `load_buf()` minus one, and works downward. The function decrements this index. When it becomes −1, the program needs to reload the buffer. When loading returns 0, the file's contents have been read completely and the program should call `wtmp_finalize()`. Implementing the complete program is left as an exercise.

## Summary

Reading and updating files in Unix may sometimes require moving the file offset around in the file. The `lseek()` system call allows us to reposition this offset so that read and write operations start at specific offsets in the file. Being able to move this offset well beyond the end of a file and write data at that position gives us the ability to create *file holes*, gaps in the file containing no data. The possibility that files may contain file holes implies that the actual disk usage of a file may be different from the size reported by commands such as `ls`.

Unix systems maintain records of user logins and logouts, as well as various system events such as boots, reboots, changes in runlevel, and shutdowns. The standard set of utilities in Unix typically contains commands

that allow us to query these types of records. These include `who`, `lastlog`, `last`, and others. The files that store these records are generally world-readable, so that any user can look up who's logged in currently, or the last time that a particular user logged in, and so on. Most of these records are in binary format and must be read using system calls such as `read()` or library functions that can read binary data. They reside in files such as *lastlog*, *utmp*, and *wtmp*. We can access data from the *utmp* and *wtmp* files by using a POSIX API that does not require making system calls to read directly.

Most data associated with users contains the user ID of the user, not the username. The *passwd* file contains an entry for every user that associates the username to a user ID, and Unix provides functions for retrieving entries from this file either by supplying a user ID, `getpwuid()`, or by supplying a username, `getpwnam()`.

In the chapter, we developed a few programs to learn how to work with system files and move the file offsets around. In particular, we implemented simple versions of the `lastlog` and `last` commands, as well as a command that dumps the contents of any file based on `utmp` records, which we named `showutmp`.

## Exercises

1. Rewrite the *spl_lastlog.c* program so that it accepts a `-u` *`user1 user2 ...`* option such that, instead of printing the last login information for all users, it prints the information for the listed users. You can limit the number of arguments to 16 for simplicity.
2. Rewrite the *spl_lastlog.c* program so that it accepts a `-t` option (`t` for *terse*) that suppresses output for users who never logged in and just displays actual logins.
3. Rewrite the *spl_utmpdump.c* program to use the kernel's `read()` system call for reading the `utmp` records.
4. Write an implementation of the `spl_last` program with options that limit the range of dates for which it will output data. Specifically, give it two options, `-s` *`start_time`* and `-e` *`end_time`*, so that it only shows events that take place *after* start time and *before* end time.
5. Write an implementation of the `last` program that uses user-buffering of input, as described in "User Space Buffering of Input" on page 280 Experiment with different size buffers. To do this, make the buffer size a command option `-b` *`nrecs`*, where *`nrecs`* is the number of records to read each time.
6. The implementation of `spl_lastlog` might be improved by storing all user IDs returned by `getpwent()`, sorting them, and then accessing the *lastlog* file. Write a version of this program based on this strategy. Then time both versions on the same input files multiple times to compare their running times. If you have access to some Linux systems with many users, run the two versions on them to see which is faster.

# 7

# OVERVIEW OF FILESYSTEMS AND FILES

In Chapter 1 we introduced the basic concepts of files, directories, and filesystems, and in Chapter 2 we described the user level interface to the filesystem, namely the basic commands for working with files and directories. Here, we dig a little deeper into the hardware and software layers on which this interface is built. We begin by examining the physical layer that underlies a typical disk-based file system. After that, we discuss the different types of filesystems supported by Linux, as well as the Linux *Virtual File System* (*VFS*). We then describe the various data structures used to implement a traditional, generic Linux filesystem, which will make it easier to develop programs that interact with the filesystem with cognizance of portability considerations in the design of these programs. This leads us to explore the programming interface to the filesys-

tem, examining some of the functions of the Unix API for retrieving both file and filesystem attributes. Finally, we put this knowledge to use in the design and implementation of a few programs that print these attributes.

## Disks and Disk Partitions

Files play a fundamental role in all Unix systems. Ordinary files can contain data and programs, directory files organize sets of files, and various types of special files allow us to interact with devices in the same way that we access regular files.

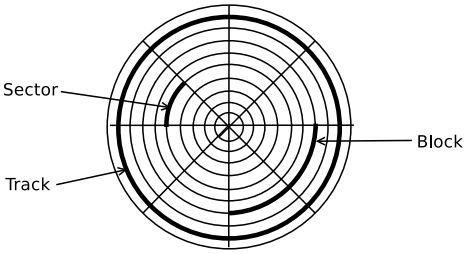
Filesystems are the framework for storing files. They organize the entire collection of files, providing both the infrastructure and an interface for accessing them. Most filesystems are *disk-based*, meaning that they reside on some type of disk storage device, such as a magnetic disk. They can also reside on other types of physical storage devices such as magnetic or optical tapes, internal memory, and solid-state devices such as flash drives. Some are memory-based. Because the most common filesystems are disk-based, we begin this chapter with a brief overview of the structure of disks and the software that manages them.

### Disk Geometry

Even though the term *hard disk* is in the singular, a hard disk typically consists of multiple disks, which we call platters. A *platter* is a circular, rigid disk that has two *surfaces*, each of which can store data magnetically. They're usually made from glass, aluminum, or ceramic. The platters rotate together at a constant fixed rotational speed around a spindle, which is connected to a motor. The rate of rotation is measured in *rotations per minute* (*RPM*).

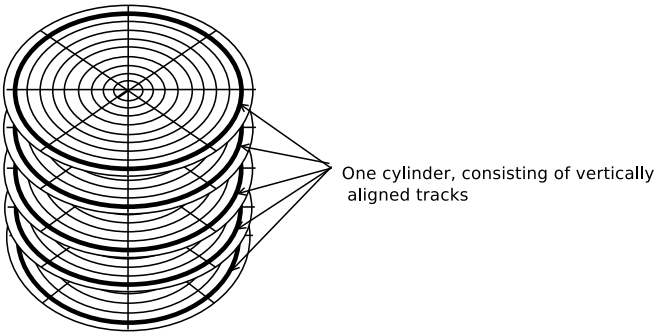
Data is encoded on each surface in concentric circles. Each concentric circle is called a *track*. The set of all tracks on all surfaces that are at the same radius from the center of the disk is called a *cylinder*. The tracks in a cylinder are therefore aligned vertically. Tracks are divided into equal length segments called *sectors*, though sometimes different tracks can have a different number of sectors. *Physical blocks* consist of one or more sectors. Physical blocks are most often 512, 1024, or 4096 bytes in size, but they can be other sizes, as long as they're multiples of 512 bytes. A block is the smallest unit of data that can be transferred to or from a disk. Sometimes groups of adjacent blocks are called *clusters*. Figure 7-1 is a schematic representation of a single surface, and Figure 7-2 depicts schematically a cylinder for a five-platter disk drive, which would have ten tracks in every cylinder.

*Figure 7-1: The structure of a typical disk surface, showing sectors, blocks and tracks*




In the figure, the thickened arcs represent sectors, blocks, and tracks. A block is shown having two adjacent sectors, and each track has eight sectors.

*Figure 7-2: A schematic representation of a cylinder for a five-platter disk*




Each surface on a disk has a disk head, which moves like the tone-arm on a phonograph. The tracks of a single cylinder can all be read with the disk head in the same position. Figure 7-3 is a photograph showing a set of disk heads for a three-platter disk drive. The disk head can both read and write data on the disk, but it needs to be moved into position to do so.

*Figure 7-3: A photograph of an opened hard disk drive, showing the disk heads; photo credit: Geni, license: CC BY-SA 3.0 via Wikimedia Commons; https://upload.wikimedia.org/wikipedia/commons/d/dc/MPC310AThard_drive.JPG*

To read data, for example, the disk head must be moved to the track containing the data. This is called *seeking*. The time it takes to move the head to the correct track is the *seek time*. Once the head is on the correct track, the disk must be rotated until the block to be read is under the head. The time that it takes to rotate the disk to this position is called the *rotational delay*. Once the head is over the needed sector, the data is transferred. The seek time and rotational delay are start-up costs of an I/O operation, part of its overhead, with seek time dominating this overhead.

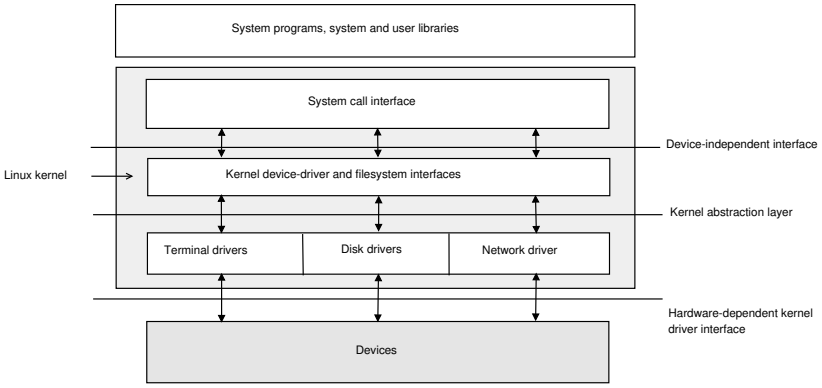
In comparison to the amount of time needed to read or write information in main memory, data transfers to or from a disk are slow because of seek time and rotational delay. The time it takes to transfer data is a function of the amount of data, but generally, it is orders of magnitude greater than the time to read memory.

## Disk Device Drivers

The kernel interacts with disks through device drivers. A *device driver* is a collection of kernel functions that make a device respond to the various system calls such as `read()`, `write()`, `lseek()`, and so on, by communicating with the device. A *disk device driver*, or a *disk driver* for short, is a particular kind of device driver that interacts with disk drives. In essence it operates a disk device's controller, causing actions such as moving the disk head and activating reading and writing on the disk.

Each different disk has a different controller and therefore, the disk drivers are specific to particular disk devices. On the other hand, no matter which type of disk the driver controls, its interface to the kernel is the same. In short, the kernel has a set of interface specifications with which each disk driver must conform. Figure 7-4 illustrates schematically the relationship between disks, disk drivers, the kernel, and the filesystem.

*Figure 7-4: The layering of interfaces from the hardware up to userspace applications*




The hardware is the lowest level of the computer system, the device drivers interact directly with it, and the kernel interacts with them. This organization is part of the concept of device independence that is characteris-

tic of Unix systems — processes and higher level parts of the kernel are freed from having to be aware of the differences in devices.

## Disk Partitioning

In the early days of Unix, a hard disk was formatted as a continuous sequence of blocks intended to contain a single filesystem. Over time, disk capacities increased and it became possible to divide a single disk into multiple, non-overlapping logical entities, each containing a distinct filesystem. These separate portions of a hard disk were called *disk partitions*, or *partitions* for short. Disk partitions are also called *logical disks*. The act of dividing the disk into partitions is called *partitioning* the disk. Figure 7-5 shows the layout of a disk that has been partitioned. The first sector of the disk contains a record of how the disk has been subdivided. This record is often called a *Master Boot Record* (*MBR*), but modern systems also call it the *Globally Unique Identifier Partition Table* (*GPT*). In the figure it is named the *Disk (Master) Record*.

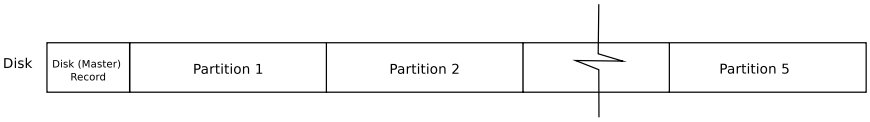



*Figure 7-5: Layout of a disk with five partitions*

Partitions can be used for purposes other than filesystems. Unix systems define a type of partition called a *swap partition*, or *swap area*, for managing memory. When the kernel needs to make room in memory for a new process, it writes the memory images of selected memory-resident processes into this swap partition. Another use of partitions is for database systems. Database management systems often use the disk in *raw mode*, meaning without a filesystem.

Some of the key benefits of partitioning a disk include the following.

**More control of file security.** The files of different user groups can be placed into different partitions, each with its own mounting options, such as whether or not it is writeable or read-only, thereby allowing different degrees of security for different user groups.

**More efficient use of the disk.** Different partitions can employ different block sizes and file size limits so that filesystems that tend to have much larger files will have different parameters than those that have smaller ones.

**More efficient operation.** When a disk is partitioned, the distances that the disk head needs to travel to perform reads and writes tend to be shorter than if it is one large disk, thereby reducing the disk access times.

**Selective backup procedures.** Backups can be performed on individual partitions, rather than entire disks, thereby making it possible to back-up different filesystems at different intervals.

**Improved failure recovery.** When disk media has failures, the damage can be restricted to a single partition rather than the entire disk, so that a smaller set of files needs to be repaired or restored.

**Reliability.** Partitioning can be used to create redundant copies of files, reducing the risk of data loss when one part of the disk is corrupted because of physical problems or malware attacks.

The biggest disadvantage of partitioning a disk is that partitions can not be increased in size. If a partition is created with too small a size, it can reach capacity quickly and cannot be made larger. In this case the entire disk needs to be repartitioned.

## Many, Many Filesystems

Neither the design nor the implementation of the Unix filesystem is part of any Unix standard. Over the years, various flavors of Unix developed their own filesystems, each of which had its own unique interface and implementation. For example, Tanenbaum[29] created the *Minix File System* when he wrote the Minix operating system in 1987, and McKusick *et al*[15] later developed the *Berkeley Fast File System* (*FFS*) for BSD2. Several Unix distributions adopted and modified the Berkeley FFS. A filesystem derived from FFS is often called a *Unix File System*, or *UFS*. The developers of Solaris, for example, created *Solaris UFS*[14], based on FFS. As of this writing there are dozens of UFS filesystems as well as many others supported by Unix distributions in general. The Wikipedia page, *https://en.wikipedia.org/wiki/Comparison_of_file_systems* contains a long list of them. Here, we'll explore those that are supported by Linux kernels.

### Filesystems Supported by Linux

Modern Unix systems such as Linux often support a wide range of different filesystems. We can learn which filesystems are supported on the machine we're using with a man page search. Entering **`apropos filesystem`** will output a long list of man pages related to filesystems, which we have to filter manually, but entering **`apropos filesystems`** results in a shorter list, among which we see the following

```
$ apropos filesystems
--snip--
filesystems (5)      - Linux filesystem types: ext, ext2, ext3, ext4, hpfs, i...
--snip--
```

The `filesystems` man page shows that the current version of Linux as of this writing supports several filesystems, some of which are listed below.

**Ext2** The high performance disk filesystem used by Linux for fixed disks as well as removable media.

**Ext3** A journaling version of the Ext2 filesystem.

**Ext4** A performance upgrade of the Ext3 filesystem.

**Minix** The original filesystem of the Minix operating system, which was the first to run under Linux.

**ISO9660** A CD-ROM filesystem type conforming to the ISO 9660 standard.

**NFS** Sun's Network File System. It also supports other network filesystems.

**tmpfs** A filesystem whose contents reside in memory.

**proc** A pseudo-filesystem which is used as an interface to kernel data structures

The first five are disk-based, but the last three filesystems in this list are not. NFS is a network file system that supports access to files on different computers across a network. The tmpfs filesystem resides entirely in memory, and the proc filesystem is not a true filesystem; it looks like one but in fact it is just a file-like interface to a set of data structures managed by the kernel.

### The Ext Filesystems

When Linus Torvalds wrote the first version of Linux, he incorporated the Minix operating system into it, mostly because it was already written and bug-free[3]. Shortly after, he and others in the Linux development community implemented a new filesystem named the *Extended Filesystem* (*Ext*), which added many new features[2]. Subsequently, the Ext2 filesystem was written by Rémy Card, Theodore Ts'o, and Stephen Tweedie specifically for Linux in 1992 and released in 1994[3]. It was widely used and was designed with provisions for future enhancements.

The next Linux filesystem was Ext3, which was developed by Stephen Tweedie and which differs from Ext2 only in that it contains journaling. *Journaling* is a way to maintain filesystem consistency in the event of hardware failures. A *journal file* records all of the actions that are supposed to be taken on the filesystem, such as creating and deleting files, changing their contents or attributes, and so on. In a journaling filesystem, this record can be used to recover the state of the filesystem without the lengthy task of examining every block on the disk. Ext2 and Ext3 are interchangeable in that one can be converted to the other while the filesystem is mounted because the difference is only in the journaling.

The Fourth Extended File System, Ext4, was released in 2008, mostly to improve performance. While Linux supports many types of filesystems, the Ext2, Ext3, and Ext4 filesystems are native to it and found on almost all Linux systems. Although there are now several different Linux filesystems, many are derived from Ext2. Since it's easier to understand filesystem concepts with a specific filesystem, the structure of the filesystem described in these notes is mostly based on the Ext2/3/4 systems. For most discussions, it doesn't matter which it is, but for others, it will matter and in those cases I'll be specific about which I mean.

## Filesystem Structure

A filesystem is not just a collection of data structures written onto a disk; it is akin to a C++ object, consisting of data structures and *methods that act upon them*. In general, while the methods of a software system are important to understand, it is often sufficient to know just its data structures to understand how that system works. Linus Torvalds advocated this principle when he was discussing his design of the `git` version control system in 2006, writing "... I'm a huge proponent of designing your code around the data, rather than the other way around" (*https://lwn.net/Articles/193245/*) to emphasize the importance of good data structures. By examining the main data structures of the filesystem, we'll get a good sense of what takes place when our programs issue requests for the kernel to read or write data. Therefore, in this section, we'll focus first and foremost on the organization and the data structures of the Ext2/3/4 filesystems, and only touch a bit on some of their methods.

### Partition Layout

In a modern Linux filesystem, the very first block in the disk partition is the *boot block*. After the boot block, the rest of the space is subdivided into a sequence of equal-size chunks called *block groups*. This is depicted in Figure 7-6, which shows the organization of the partition as well as what is contained in each block group. The figure also shows how many physical blocks are used by each part of a block group.

Earlier Unix systems grouped blocks into *cylinder groups*, which were blocks contained in one or more adjacent disk cylinders. A cylinder group is a physical concept, tied to the geometry of the disk, but a block group is a logical concept, independent of the disk geometry, because modern hard disk drives hide the geometry from the operating system. If in your readings you encounter references to cylinder groups, think of them as block groups.

The boot block contains information needed by the operating system to boot the computer. Although there's a boot block in every filesystem on a disk, the operating system only uses the very first boot block on the disk for booting under normal circumstances.

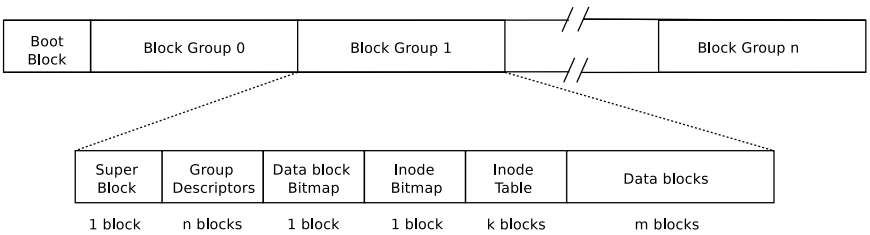



*Figure 7-6: Layout of an Ext2 partition with* n *block groups, and an exploded view of one block group*

Unix systems other than Linux use a similar decomposition of a partition into equal-size groups; the Berkeley FFS called them cylinder groups.

Shortly we'll see why it's more efficient to subdivide a partition into equal-size groups, but first let's see what components each block group contains, after which we'll go over what information these components store and how it's used.

### *Block Group Layout*

In the Ext2/3/4 filesystems, every block group contains the following data:

- A copy of the filesystem's *superblock*.
- A copy of the block group's set of *group descriptors*.
- A data block bitmap.
- An inode bitmap.
- An inode table for the files in that block group.
- The data blocks of all files in that block group.

Now let's explore each of these components.

#### The Superblock

A copy of the superblock is the first block in each block group. The superblock is a large data structure, with more than 100 members, containing parametric information about the filesystem, such as how many inodes it has, the total number of blocks, the block size, the numbers of reserved and unused blocks, timestamps of various kinds, various flags indicating whether it is read-only or locked, information about the system's mount status, and much more. The term often used to describe this type of information is *metadata* — data about data. The kernel uses the superblock in block group 0 alone. Copies are kept in the other block groups in case of a filesystem failure.

#### Group Descriptors

Every block group has its own set of group descriptors. The group descriptors store information about the group such as the address of the starting block of each other component of the block group, how many blocks in the group are in use, how many are free, and so on. For example, the data structure in Ext4 that stores group descriptors is of type `ext4_group_desc` and has a couple dozen members. Each group contains the set of group descriptors of all groups in the partition for reliability, in case of filesystem corruption.

#### Data Block Bitmap

The data block bitmap is a bitmap with one bit for every data block in that group. If the block is in use, the bit is 1, and if free, the bit is 0. The data block bitmap is allocated one block on the disk. If that block is 4096 bytes (4KB) in size, it has $8 \times 4096 = 2^{15}$ bits. In this case, the block group can have at most $2^{15}$ blocks, each of size 4096 ($2^{12}$) bytes, for a total of $2^{27}$ bytes (128MB) per block group.

### Inode Bitmap

The inode bitmap serves a similar purpose for inodes as the data block bitmap does for data blocks. It contains a bit for each inode in the inode table, which indicates whether it is in use or free. Since this bitmap is also allocated exactly one 4096-byte block, the inode bitmap can keep track of $2^{15}$ inodes.

### Inode Table

Inodes used to be stored in two separate lists, the free-list and the used-list. In modern systems, the inodes are usually in a table, and this is the case for Linux's Ext2/3/4 filesystems. The inode table stores all inodes for files whose data is in the block group. We introduced inodes in Chapter 1, "File Attributes, Permissions, and Contents." There we noted that an inode stores a file's status, the original term for its attributes. The term *file metadata* is often used to describe the contents of the inode, especially in the context of filesystems.

The structure that represents an inode is of type `struct ext4_inode` in Ext4, and similarly named for the other filesystems. It has more then 20 members. In Ext2 and Ext3, the inode is a fixed size of 128 bytes. Doing a bit of arithmetic for Ext2 and Ext3, with a 4096-byte block size, each block can store 4096/128 = 32 inodes. The superblock determines how many inodes can be in each block group. If, for example, a block group can have 256 inodes, then storage for the inode table would require 256/32 = 8 blocks.

In Ext4, the inode can be larger. Ext4 added more fields to the inode than were present in the earlier systems. For example, the `i_crtime` member, which stores the file creation time, was not in the earlier inodes, but was added to the `struct ext4_inode`. The inode itself has a member named `i_extra_isize` that indicates how much larger than 128 bytes it is.

In Chapter 1, in "Files, Directories, and the Single Directory Hierarchy", we saw that a defining characteristic of Unix file management is that file data is not stored with the file's metadata, and that all of a file's metadata is stored in the inode. This includes timestamps such as when the file was created, last modified, and last accessed. It also includes its mode, its size in bytes, how many blocks it uses, and how many links refer to it. Most importantly, it's where the pointers to all of the file's data blocks are stored. This implies that the inode must be accessed many times in order to access the file's data.

The method of storing files in Unix is flexible and efficient. Its design was visionary, because it allowed for huge files, even when there was no way to store huge files. The inode in a Unix system contains an array of (typically) 15 block pointers. A block pointer is usually four bytes long. For systems with 15 block pointers, they're used as follows.

- For regular files, the first 12 block pointers in this array are the addresses of the first 12 blocks of the file. If the block size is 4096 bytes (4KB), then a file of size at most $12 \times 4096$ bytes, or 48KB, can be accessed by one level of indirection through these pointers.

- If a regular file is larger than 48KB, then the 13th pointer contains the address of a *single-indirect block*, which is a 4096 byte block used to store block addresses. Since a block address is four bytes, there are 4096/4=1024 block addresses in this block. Since each of these 1024 blocks is 4096 bytes, the 13th pointer allows for addressing an additional 1024×4096 bytes (4MB). Therefore, using the first 12 pointers and the 13th allows for accessing files whose size is up to 48KB + 4MB.
- For still larger files, the 14th pointer is the address of a *double-indirect block* that similarly contains 1024 addresses of single-indirect blocks, each of which contains 1024 block addresses. This accommodates files with sizes up to 48KB + 4MB + (1024×1024×4) KB, which is 48KB + 4MB + 4GB.
- The 15th address is that of a *triple-indirect block*, which, needless to say, points to 1024 double-indirect blocks. Since each double-indirect block points to 1024×1024 data blocks, using this pointer, we can access 1024×1024×1024 blocks, each of size 4KB, in addition to the blocks pointed to by the other pointers. This lets us address files whose total size is in excess of 1024×1024×1024×4KB, which is 4TB.

Figure 7-7 depicts the use of these direct and indirect blocks in the inode.

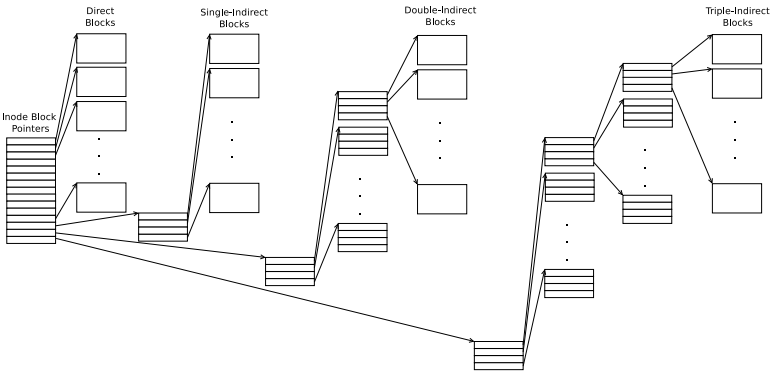



*Figure 7-7: The use of direct and indirect blocks in the inode*

For clarity, in that figure, only four addresses are shown in the indirect blocks. The number of addresses in a 4KB block would be 1024.

### Data Block Area

The last part of a block group is the set of blocks reserved for file data. Every effort is made to store all of the blocks of a file in the same block group as its inode. For large files, this is not always possible, and their data blocks may be allocated in other block groups.

### Performance Considerations

Modern filesystems do not try to store all of a file's data as a single sequence of consecutive blocks. Although accessing the data of a file would be faster if they did, the disk space would be utilized poorly, because there would be many empty gaps that would be too small for entire files. In addition, finding space to write a new file would take more time, since the filesystem would have to find free space large enough for the file. Instead, filesystems divide the file's data into blocks and store the blocks in non-contiguous locations. This results in very high disk utilization, but it introduces other problems.

For one, it takes more time to find a file's data blocks. For another, because the blocks may not be close to each other, it causes more disk seeking between accesses, increasing file access *latency*, the time needed to set up the access before the transfer of data.

If a partition were not subdivided into block groups and there were a single inode table at the beginning of the partition, then every file access would require even more disk seeking for moderately large files, because the inode needs to be accessed each time a new data block must be accessed, implying that the disk head would have to travel back and forth between the inode table and the data blocks frequently. In addition, the blocks of a file could be very far away from each other, causing more seeking.

The use of block groups mitigates these problems while still allowing the blocks of a file to be stored on disk non-contiguously. It decreases overall seek time because the inodes and bitmaps that are used to locate data blocks are either in the same cylinder as the blocks or close to that cylinder. Also, the allocation method used by the inodes in Unix allows the kernel to calculate the starting address of a block with simple arithmetic. However, the use of double-indirect and triple-indirect pointers increases the CPU time needed to access data blocks, because several pointer dereferences are needed for each block. This increase in time is offset by the use of the kernel I/O buffering described earlier, in Chapter 5.

Another issue regarding performance is the size of the block. Files are always allocated whole blocks, never pieces of a block. It's extremely rare for the size of a file to be an exact multiple of the block size. Because of this, the final block of storage is only partially filled. The unused, or wasted, space inside a block is called *internal fragmentation*. On average, the fraction of the last block that is unused is 50%. This implies that the larger the block size, the more space is wasted in that last block.

When most files are small, larger block sizes result in more wasted disk space, because small files have fewer blocks, so proportionately, the wasted space in the last block is a larger fraction of the file size. As an example, suppose that block size is 4KB. If files are 100KB in size on average, they use an average of 25 4KB blocks, of which 2KB in each last block is wasted space. Therefore, the unused space per file is 2KB/100KB , or 2% of its allocation. On the other hand, if files are much smaller, about 16KB in size, they need just four 4KB blocks each, so their wasted space is 2KB/16KB per file, or

12.5%. Wasted space also translates to wasted time, since there's more disk activity and more disk waits on average.

Larger block sizes improve performance for file systems expecting large files. Often a system administrator will choose smaller block sizes for the root file system, which tends to have smaller files, and larger ones for user data. How large are files on average? In 1993, one study that surveyed the sizes of files found on the Internet, by collecting data on over 12 million files across 1000 file systems, found that the median file size was just under 2048 bytes, with the average size being 22KB[8].

## The Kernel's Filesystem Interface

A filesystem has to provide methods that the kernel can call so that it can provide its services to user programs. Such methods include functions to create files, to read and write data, to retrieve file properties, to move the file offset, and so on. It also has to provide functions for retrieving information about disk usage and other filesystem properties. To get a better sense of the kernel's interaction with the filesystem we'll work through an explicit example, namely creating a new file and writing data into it.

### *Creating a New File*

Suppose that the current working directory is */home/snw/testing*, and that we enter the command

```
$ gcc -o myprog myprog.c
```

Assuming that the program is compiled and linked, and that we have write and execute permission for the directory */home/snw/testing*, `gcc` will create a file named *myprog* in this directory. To create this file, `gcc` must either call `open()` with the `O_CREAT` flag or call `creat()`, requesting the kernel to create it. The kernel in turn must perform a sequence of actions, which it does by making calls to lower-level filesystem methods. In the following discussion, when I say that the kernel does this or that, I really mean that the kernel calls various filesystem methods that actually perform that action.

To create a file, the kernel takes the following steps, which leave out several details, such as handling errors:

1. It checks whether the filename is valid and whether the filename doesn't exist already in the given directory.
2. It checks whether the process has permission to create a file in this directory.
3. It acquires a new inode for the file.
4. It fills in the inode with the file status.
5. It creates a directory entry in the *testing* directory with the inode number and filename *myprog*.

Each of these steps is explained in more detail below.

#### Checking Whether the Filename Exists

The kernel checks whether the filename is too long or has invalid characters and so on. It then checks whether the filename does not already exist in the given directory, */home/snw/testing*. If any checks fail, it stops here, with a suitable message. If not, it continues to the next step.

#### Checking Permissions

The kernel checks whether the file can be created in the given directory before it continues. If all goes well, it continues.

#### Creating the inode

The kernel tries to create an inode. It must get a free inode in the inode table. The inode bitmap is used for this purpose. If there are no free inodes, the kernel must report the error and stop here. In this case we'll get a message that the filesystem is full. In this step, a copy of the inode table in memory is used; the disk version of it isn't accessed.

#### Updating the inode

Assume that the kernel obtained an inode, let's say one with index 47 in the inode table. The kernel fills the inode with the owner, permissions, time of last modification, and so on. It then saves the inode number, 47, of this inode, for later use. The updates to this inode are in the memory copy of the table, not the disk-resident copy. The disk copy is updated periodically by the kernel.

#### Recording the File Name in the Directory

If all of the preceding steps were successful, then the kernel creates a new entry in the current working directory consisting of the pair (47, *myprog*), because 47 is the inode number and *myprog* is the name.

### *Writing Data to a File*

Our example command, `gcc -o myprog myprog.c`, also writes the executable code to the file, which means that the running process issues the `write()` system call to write that data to the file. Two major steps that must be performed by `write()` are

1. Allocating data blocks for the file and storing the file data into these blocks.
2. Recording the addresses of the data blocks in the inode.

We describe these steps in more detail.

To write the data to the file, the kernel must acquire the right number of free blocks. While `gcc` is running, it is generating the data to write to the file, creating it in smaller pieces at a time. Each chunk is given to the kernel through the `write()` system call. Because the kernel does output buffering, the file is being stored in kernel buffers, which are not written to disk un-

til the buffers are flushed. If the amount of data is small, all of it will fit in memory buffers and the kernel will know exactly how many disk blocks are needed for it. If the file is very large, the kernel may start allocating blocks before it knows the file's actual size. Assuming that there are enough free blocks, it will first allocate direct blocks. If the file is larger than the number of bytes that can fill all of the direct blocks, the kernel allocates single-indirect blocks as needed. If it is larger than the amount of storage they can provide, it starts allocating double indirect blocks. It continues this procedure, using triple-indirect blocks if not even all of the double-indirect blocks will suffice.

Note that the data block bitmaps are used in this step to find free blocks and that the bitmaps are modified to mark blocks as being in use as they're allocated. Note too that the locations of the data blocks are implicitly recorded in the inode by these steps because the data block pointers point to them.

## The Virtual Filesystem (VFS)

The filesystem design just described is the basis for many Unix filesystems, but in general, filesystem implementations differ from each other. Furthermore, Unix systems almost always support the ability to mount different types of filesystems onto the directory hierarchy, which implies that different parts of the hierarchy can be on devices with different filesystems. This leads to a problem that we now introduce by example.

Many Unix systems allow users to mount Microsoft's *FAT*, *FAT32*, and *NTFS* filesystems. *FAT* stands for *File Allocation Table*, and is the file system found on many Microsoft operating systems as well as on external storage devices such as USB flash memory drives. *NTFS* is Microsoft's *New Technology File System*, introduced in 1993 with Windows 3.1. In Chapter 1, we saw that a directory in Unix is a file that consists of a list of directory entries, each of which contains the name of a file and a reference to that file. In FAT and FAT32 systems, directories don't have this structure. In order to mount these systems and access the files in them, the kernel must make their directories look like the traditional Unix directories. This is just one problem.

The more general problem is that, when many different filesystems are mounted onto the directory hierarchy, the kernel can't have single implementations of the various file-related system calls, such as `read()`, `write()`, and `lseek()` because the code in those functions depends on how the filesystem is implemented.

Consider our simple implementation of the `cp` command from Chapter 5, which we named `spl_cp`. It makes calls to `open()`, `close()`, `read()`, and `write()`. Suppose that we insert a USB flash drive that has a FAT filesystem on it into a USB port on our machine. Suppose too that the drive's name is *MyDrive*. Modern machines automatically mount these flash drives by attaching them to the directory hierarchy either under */media* or under */mnt*. Assuming that it's mounted under */media* and we want to copy a file named *mywork* from that drive into our home directory, we'd enter the command

```
$ spl_cp /media/MyDrive/mywork  ~/mywork
```

and it would successfully copy the file from a FAT filesystem to the native filesystem, say Ext4. We explore why this works.

The locations and sizes of a file's data blocks vary from one system to another, making it almost impossible to have a single function that finds them without knowing the underlying filesystem implementation. As a result, the actual machine code that's executed when the file-related system calls are invoked is not bound to the system call names when the kernel is compiled.

Let's try to understand this problem in terms of a different problem with which we're more familiar, namely how pointers and virtual functions work in a programming language. When we don't know at compile time how much storage a variable will need for a running program because it depends on how much data is input, we don't declare that variable statically. Instead we declare a pointer and allocate memory to the structure at run-time. This is called *run-time* or *delayed binding* because the binding of the name of the variable to its storage location is delayed until run-time. In C++, when a class contains a virtual function, the code that's executed when that function is called is not bound to the function's name until run-time, which is another form of delayed binding. In the case of virtual functions, the solution is opaque to the programmer because the C++ run-time library handles the binding internally with a special table called a *virtual dispatch table*.

In Linux, as well as in several other Unix systems, this same idea underlies the kernel's interface to the filesystem. The designers of Ext2 created a layer of abstraction within the kernel on top of all mounted filesystem operations. This layer is called the *Virtual Filesystem* (*VFS*). The VFS defines an abstract filesystem interface and hides its implementation. At run-time, it binds the implementations of filesystem-related calls to functions that are hard-coded in each mounted filesystem, which is, in essence, a form of delayed binding. The original Linux VFS was written by Chris Provenzano and later rewritten by Linus Torvalds.

The VFS defines a set of functions that every filesystem is required to implement. This interface is made up of a set of operations associated to three kinds of objects: filesystems, inodes, and open files.

When a process issues a file-oriented system call, the kernel calls a function contained in the VFS. This function handles the structure-independent operations and then redirects the call to a function contained in the physical filesystem code, which is responsible for handling the structure-dependent operations.

For example, let's consider the `read()` system call. When a program opens a file, it gets a file descriptor for the open file. That file descriptor is a reference to a data structure that represents the file, which we learned earlier is called the open file description. One field of this structure, in Linux named `f_op`, is a pointer to a table of function addresses. The actual function that's called when `read()` is invoked is `f_op->read(...)`. If the file were on an MS-DOS filesystem, one function would be called and if it were on an Ext4 filesystem, a different function would be called.

How is the table pointed to by `f_op` initialized with the addresses of the functions? The VFS stores information about filesystem types supported by the kernel in a table that's created during the kernel configuration. When a filesystem is mounted, the kernel uses this table to populate a *mounted filesystem descriptor* with the data needed by the VFS. A mounted filesystem descriptor contains several types of data, including data common to all filesystem types, pointers to the functions provided by the actual filesystem, and private data maintained by the actual filesystem code.

The VFS supports many different types of underlying physical filesystems. In fact, in Sun's variants of Unix, from SunOS through Solaris, and in BSD and FreeBSD, the concepts of inode and inumber (inode number) have been replaced with those of *vnode* and *vnumber*, with the 'v' standing for *virtual*. Linux continues to use the term inode. A schematic representation of these levels within the *Linux Second Extended File System* (*Ext2*), based on [3], is depicted in Figure 7-8.

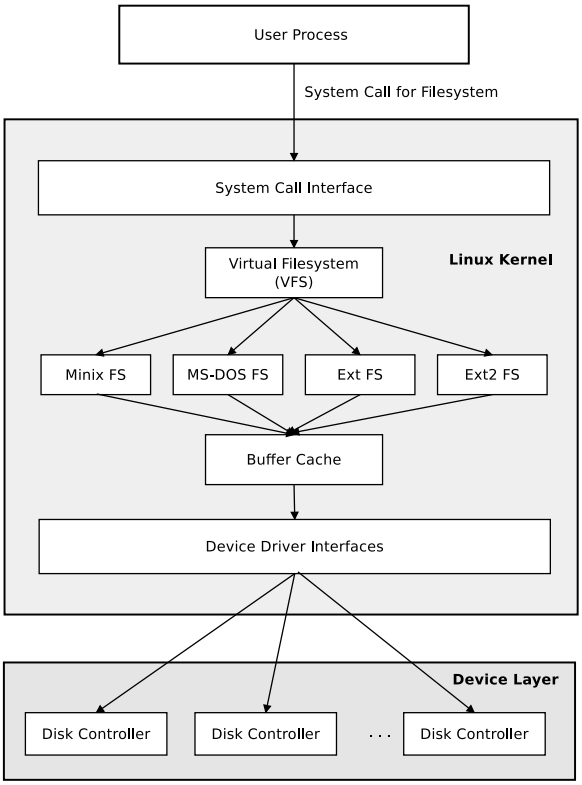



*Figure 7-8: The Linux Second Extended File System (Ext2)*

## Exploring the Filesystem API

Our next goal is to apply what we've learned about file and filesystem attributes to write a few programs that interact with the filesystem-related

parts of the kernel API. Candidate programs could be ones that display the metadata of a given filesystem, or those that display a file's metadata. If we know how to retrieve a file's metadata, we could write many different useful commands, such as one that determines whether two filenames are links to the same actual file, or whether they're owned by the same user, or which of two files was created or modified more recently.

Our ultimate goal is to write a program that accesses filesystem metadata. If we can find a command that can display this metadata, we could model our program after it. We'll discover that, because the API related to filesystems is not a part of POSIX, navigating through it is a bit murky. Nonetheless, this will be a valuable exercise in system programming.

We'll begin by searching the man pages for a command that displays filesystem metadata, in the hope that its man page will lead to other resources. It isn't enough to search only for the one-word term *filesystem*, because in the man pages it is sometimes one word and sometimes two, because different man pages are authored by different people.

In the following query, we limit the search to Section 1 and snip out matches that aren't relevant.

```
$  apropos -s1 "filesystem" "file system"
--snip--
lsattr (1)           - list file attributes on a Linux second extended file s...
--snip--
stat (1)             - display file or file system status
```

The first command, `lsattr`, isn't what we want since it just displays a list of files and their attributes in an Ext2 filesystem. The second of the two, `stat`, is one we introduced briefly in Chapter 1.

## The stat Command

Let's look at the `stat` man page in Section 1 to see if it leads us in the right direction:

```
$ man -s1 stat
STAT(1)                          User Commands                          STAT(1)

NAME
       stat - display file or file system status

SYNOPSIS
       stat [OPTION]... FILE...

DESCRIPTION
       Display file or file system status.

       Mandatory  arguments  to  long  options are mandatory for short options
       too.
```

```
       -L, --dereference
              follow links

       -f, --file-system
              display file system status instead of file status
--snip--
```

When we used the `stat` command in Chapter 1, it was to view a file's status. Notice though that it can also be used to display attributes of filesystems, by giving it the `-f` or `--file-system` option. In this case, it shows the status of the filesystem on which the given file resides. The difference in output with and without `-f` is illustrated here:

```
$ stat /etc/bash.bashrc  # show status of file /etc/bash.bashrc
  File: /etc/bash.bashrc
  Size: 2319       Blocks: 8          IO Block: 4096   regular file
Device: 10302h/66306d Inode: 10486298    Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2018-06-05 14:03:30.199540534 -0400
Modify: 2018-04-04 14:30:26.000000000 -0400
Change: 2018-06-05 14:03:30.199540534 -0400
 Birth: 2018-06-05 14:03:30.199540534 -0400
$ stat -f /etc/bash.bashrc  # show status of filesystem containing /etc/bash.bashrc
  File: "/etc/bash.bashrc"
    ID: b07bc00fdedb9bf9 Namelen: 255     Type: ext2/ext3
Block size: 4096       Fundamental block size: 4096
Blocks: Total: 58651894   Free: 28744967   Available: 25747399
Inodes: Total: 14974976   Free: 13698698
```

Without the `-f`, `stat` displays some of the file's attributes whereas with it, it displays attributes of the filesystem containing that file.

The man page further states that `stat` has a `-c` *`FORMAT`* option to control which attributes it displays as well as their formats, where *`FORMAT`* is a string similar to that used in the `date` command (see Chapter 4). The format-specifiers consist of a percent sign followed by a letter, such as `%a`, `%b`, and so on. The meanings of the various format specifiers depend on whether or not the `-f` option is present. Without it, they specify formats for file status and with it, for filesystem status. For example, for files, `%b` is the number of blocks allocated to the file, whereas for filesystems, it is the total number of blocks in the filesystem. To demonstrate, we enter the two commands

```
$ stat -c"Blocks allocated: %b" /etc
Blocks allocated: 24
$ stat -f -c"Total blocks in filesystem: %b" /etc
Total blocks in filesystem: 58651894
```

Some of the other format specifiers for filesystem status are as follows:

**%b** total data blocks in file system

**%c** total file nodes in file system

**%d** free file nodes in file system

**%f** free blocks in file system

**%i** file system ID in hex

**%S** fundamental block size (for block counts)

**%T** file system type in human readable form

We'd like to write a limited version of this command that just displays filesystem metadata, as an exercise in using the Linux filesystem API. First, we check whether the man page has enough information in it to get started. In the `SEE ALL` section of the page it mentions three system calls: `stat()`, `statfs()`, and `statx()`. The man pages for `stat()` and `statx()` tell us that they display status of files, not filesystems. The third, `statfs()`, gets filesystem statistics and might be what we want, but out of curiosity, we'll look at the `stat()` man page in Section 2, because learning about `stat()` might give us some insights into writing our program, which we'll return to later.

## The *stat* System Call

Because there's both a command and a system call named `stat`, to view the `stat()` system call's man page we need to specify Section 2 when issuing the `man` command:

```
$ man -s2 stat
STAT(2)                    Linux Programmer's Manual                    STAT(2)

NAME
       stat, fstat, lstat, fstatat - get file status

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <unistd.h>

       int stat(const char *pathname, struct stat *statbuf);
       int fstat(int fd, struct stat *statbuf);
       int lstat(const char *pathname, struct stat *statbuf);

       #include <fcntl.h>           /* Definition of AT_* constants */
       #include <sys/stat.h>
       int fstatat(int dirfd, const char *pathname, struct stat *statbuf,
                   int flags);
--snip--
DESCRIPTION
       These  functions return information about a file, in the buffer pointed
       to by statbuf.  No permissions are required on the file itself,  but in
```

```
       the  case of stat(), fstatat(), and lstat() execute (search) permission
       is required on all of the directories in  pathname  that  lead  to  the
       file.
--snip--
```

The page describes several related functions, the first three of which return information about a given file in their second argument, which is the address of a `stat` structure. The differences among the first three are only in how the file is specified:

- The `stat()` function expects a pathname for the file, and if the file we name is a symbolic link, it gives us information about that link's target. For example, if *mylink* is a soft link to *target*, `stat()` returns information about *target*.
- The `lstat()` function also expects a pathname for the file, but if it's given a symbolic link, it returns information about the link itself, not its target. Using the same example, it would return information about *mylink*.
- The `fstat()` call is given a file descriptor instead of a pathname.

The fourth system call listed there, `fstatat()`, is a more general function, designed so that it can behave like any of the other three. We won't investigate it here.

**NOTE** *It might be confusing that there's a command named `stat`, a system call named `stat`, and a data structure named `stat`! We'll be very clear about which we mean. This type of overloading of names also occurs in other parts of the API.*

None of these functions can be used to access attributes of a filesystem per se, but studying them will help us to understand how to write programs that access this type of data, because as we'll see shortly, metadata is often stored in a form that requires some type of parsing or unpacking, and these will require similar preprocessing.

The `DESCRIPTION` section of the man page tells us that we need execute permission on the path to the file if we call `stat()` but not if we call `fstat()`. If we call `fstat()`, we need a file descriptor for the file, which we can get by opening the file, whereas we don't need to open a file to call `stat()`. We'll decide later which to use, but first let's read about the `stat` structure.

### The stat Structure

The man page has the definition of the `stat` structure:

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number */
    mode_t    st_mode;        /* File type and mode */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
```

```
    gid_t     st_gid;         /* Group ID of owner */
    dev_t     st_rdev;        /* Device ID (if special file) */
    off_t     st_size;        /* Total size, in bytes */
    blksize_t st_blksize;     /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
       precision for the following timestamp fields.
       For the details before Linux 2.6, see NOTES. */

    struct timespec st_atim;  /* Time of last access */
    struct timespec st_mtim;  /* Time of last modification */
    struct timespec st_ctim;  /* Time of last status change */

#define st_atime st_atim.tv_sec      /* Backward compatibility */
#define st_mtime st_mtim.tv_sec
#define st_ctime st_ctim.tv_sec
};
```

This structure has fields to store the most consequential members of a file's inode such as the ID of the device on which it resides, its inode number, the file mode and type, various data related to its size and allocation, timestamps, and so on. The macros `st_atime`, `st_mtime`, and `st_ctime` are defined for older programs that were written before the kernel started supporting nanosecond time resolution.

The data types of all of these fields are not native C types, meaning that they're not part of the C programming language. They are *system data types*, which means that they're defined in header files in the system. The advantage of defining data structures with system data types is portability. The number of bytes in an integer type in C varies from one operating system to another. A `long int` might be four bytes on one machine, eight on another. If a program declares a variable to be of type `long int` and it needs to store a value of type `uid_t`, it may not have enough bytes on some machines. In contrast, the system data types are defined internally as `typedefs` of native C types, so that if a value is of type `uid_t` and a program declares a variable of type `uid_t` to receive that value, it is guaranteed to have the same number of bytes.

If we wanted to write a program to extract and print the data from the fields of the `stat` structure, we'd have to know more about these types. Because this is most likely going to be true for our planned `spl_statfs` program, we need to learn what these types are and how we can work with them. This man page has a lot of information, especially in the `DESCRIPTION` and `NOTES` sections; almost everything we need to know is there.

For the `st_dev` field, the page suggests reading about the `major()` and `minor()` functions:

```
st_dev  This  field  describes  the  device  on which this file resides.
        (The major(3) and minor(3) macros may be useful to decompose the
```

```
       device ID in this field.)
```

and for the `st_mode`, it suggests reading the man page `inode(7)`:

```
st_mode This  field  contains  the file type and mode.  See inode(7) for
       further information.
```

For all remaining fields, it refers us to the `inode(7)` man page. Section 7 pages are always very informational:

```
$ man -s7 inode
INODE(7)                  Linux Programmer's Manual                 INODE(7)

NAME
      inode - file inode information

DESCRIPTION
      Each file has an inode containing metadata about the file.  An applica
      tion can retrieve this metadata using stat(2) (or related calls), which
      returns a stat structure, or statx(2), which returns a statx structure.
--snip--
```

Following this brief description is a list of the inode members available for access through these system calls, with detailed descriptions of each. Some members of an inode are for internal use and not exposed in the kernel API, so they aren't mentioned in this page. The page also shows us how we can extract the file type and permission bits in the `st_mode` member, which we'll return to shortly, and describes feature test macros and conformance to standards.

How can we learn more about the other system data types appearing in the structure? In previous chapters we encountered types such as `off_t`, `size_t`, and `time_t`, which are also system data types. If we want to learn more about them, we could see if there's a man page that describes or explains more about them. A reasonable search would be `apropos "system data type"`:

```
$ apropos "system data type"
FILE (3)             - overview of system data types
aiocb (3)            - overview of system data types
clock_t (3)          - overview of system data types
clockid_t (3)        - overview of system data types
dev_t (3)            - overview of system data types
div_t (3)            - overview of system data types
--snip--
system_data_types (7) - overview of system data types
--snip--
```

We get a very long list of matches but they're all for the same page in Section 7. That page has a long list of almost all system types, with entries such as the following:

```
dev_t
        Include: <sys/types.h>.  Alternatively, <sys/stat.h>.

        Used for device IDs.  According to POSIX, it shall be an integer
        type.  For further details of this type, see makedev(3).

        Conforming to: POSIX.1-2001 and later.

        See also: mknod(2), stat(2)
--snip--
```

For each listed type, it tells us which header file has its declaration, what it's used for, what kind of type it is, such as whether it's an integer type or a structure of some kind, what the various standards say about its size, and which interface functions use it. In general, when we want to learn what a specific type is, this page is our starting point.

## The File Mode

The file mode is stored in the `st_mode` member of the `stat` structure. POSIX.1-2017 specifies the purpose of each of its sixteen bits. The highest-order four bits are called the *file type* bits. The low-order twelve bits are called the *file mode bits*. Among these twelve bits, the low-order nine bits are the *permission bits*. The three bits above them are the *special bits*. Figure 7-9 illustrates the meanings of the bits.

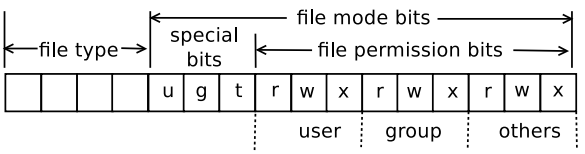



*Figure 7-9: The file mode bits in the st_mode member*

The file type bits define which of the seven possible file types the file is, such as whether it's a regular file, a directory, a symbolic link, or one of the special files. The special bits alter permissions in a few different ways. The highest order special bit is the *setuid bit*, which we introduced in Chapter 1 and explained in detail in "The setuid Bit" in Chapter 5. The next two bits are the *setgid bit* and the *sticky bit*. We'll explain their significance and use in "The setgid Bit" and "The sticky Bit" following. The next nine bits are the permission bits, grouped into three sets of three bits each. Each set has a read, write, and execute bit. The three sets of bits are respectively the permissions associated with the user, the group, and others. A 1-bit means the permission or property is on, and a 0-bit, that it is off.

POSIX.1-2017 standardizes various macros that facilitate extracting the values of these bits. Some of the macros are masks and others are macro functions for querying the values. The `<sys/stat.h>` header file contains all of

the macro definitions, and the inode man page describes them. The single-bit masks for the file mode component of the st_mode are:

```
S_ISUID    0004000    /* setuid bit                        */
S_ISGID    0002000    /* setgid bit (see below)            */
S_ISVTX    0001000    /* sticky bit (see below)            */
S_IRWXU    00700      /* mask for file owner permissions */
S_IRUSR    00400      /* owner has read permission         */
S_IWUSR    00200      /* owner has write permission        */
S_IXUSR    00100      /* owner has execute permission      */
S_IRWXG    00070      /* mask for group permissions        */
S_IRGRP    00040      /* group has read permission         */
S_IWGRP    00020      /* group has write permission        */
S_IXGRP    00010      /* group has execute permission      */
S_IRWXO    00007      /* mask for permissions for others */
S_IROTH    00004      /* others have read permission       */
S_IWOTH    00002      /* others have write permission      */
S_IXOTH    00001      /* others have execute permission    */
```

The following code snippet is an example of they can be used, but without checking whether stat() returned an error:

```
stat("myfile", &statbuffer);
if (statbuffer.st_mode & S_IROTH)
    printf("myfile is readable by others.\n");
```

The masks for retrieving file type are

```
S_IFMT      0170000   /* mask for file type bits        */
S_IFLNK     0120000   /* symbolic link                  */
S_IFREG     0100000   /* regular                        */
S_IFBLK     0060000   /* block device                   */
S_IFDIR     0040000   /* directory                      */
S_IFCHR     0020000   /* character device               */
S_IFIFO     0010000   /* FIFO                           */
```

For example, to retrieve the file type and test whether the file is a directory, we'd write the following code, again without checking for errors from stat():

```
stat("myfile",&statbuffer);
if ( S_IFMT & statbuffer.st_mode == S_IFDIR)
    printf("myfile is a directory.\n);
else
    printf("myfile is not a directory.\n);
```

Because retrieval of the file type is such a frequently performed action, POSIX.1-2017 defines macro functions for testing the file type, which are easier to use than the masks. In the macros below, the m argument is the 16-bit value of the st_mode member:

```
S_ISREG(m)    /* is it a regular file? */
S_ISDIR(m)    /* directory?            */
S_ISCHR(m)    /* character device?     */
S_ISBLK(m)    /* block device?         */
S_ISFIFO(m)   /* FIFO (named pipe)?    */
S_ISLNK(m)    /* symbolic link?        */
S_ISSOCK(m)   /* socket?               */
```

With the macros, the previous example could be written as

```
stat("myfile",&statbuffer);
if ( S_ISDIR(statbuffer.st_mode))
    printf("myfile is a directory.\n);
else
    printf("myfile is not a directory.\n);
```

These macros make it easy to write code to determine whether a given file has specific permissions as well as to determine its type. Before we demonstrate with a few examples, let's explore the special bits mentioned previously.

### The setgid Bit

In Chapter 5, "A Process's User IDs", we introduced the user IDs associated with a process. Processes have an analogous set of group IDs, namely, a *real group ID*, an *effective group ID*, and a *saved setgroup ID*. Their meanings are analogous as well.

The setgid bit is similar to the setuid bit except that, when a file has that bit set and it contains a program, when that program is run, the *effective group ID* of the process becomes the group ID of the group of the file. If the group of a file containing an executable program is *G*, for example, and the file's setgid bit is enabled, then when the program is run, it runs with its effective group ID equal to that of group *G* rather than the group ID of the user running the program, except in a few unusual circumstances.

The setgid bit has a different meaning when the file is a directory. In this case, all files created in that directory inherit their group ID from the directory rather than from the process that creates them. This feature makes sharing files easier, since a directory with an enabled setgid bit will allow users of the same group to add files that all members of the group can use in the same way.

The setgid bit has a few interesting applications, one of which is the `write` command . The `write` command (`/usr/bin/write`), not the `write()` system call, is a command that lets users write to a terminal other than their own. The syntax is

```
$ write username [ ttyname ]
```

If a user has multiple terminals open, the optional second argument lets you specify to which terminal to write. After we enter this command, everything

we type will be displayed on the user's terminal, until we enter an end-of-input signal (CTRL-D). To try it, type `who` to see who's logged on, and which terminals they're using. Suppose I am logged in on terminal `/dev/pts/2`. You could type

```
$ write sweiss /dev/pts/2
Can I bother you?
CTRL-D
```

and wait for my response. Your typing will appear on my terminal window. How is it possible that one person can write on another person's terminal?

The `write` command needs write permission on the terminal on which it wants to write. First take a look at the list of pseudo-terminal devices in `/dev/pts`. The list will look something like

```
$ ls -l /dev/pts
crw-------  1 ariel   tty 136, 1 Mar  5 17:50 1
crw--w----  1 jake    tty 136, 3 Mar  3 16:22 3
crw--w----  1 lindy   tty 136, 5 Mar  3 15:40 5
crw--w----  1 sweiss  tty 136, 7 Mar  5 18:00 7
```

Some of these have the group-write bit set and others do not. All of these belong to the `tty` group, which means that any process that runs with the effective group ID equal to the group ID of `tty` can write to those terminals whose write bit is set. Now take a look at the `write` command's mode bits:

```
$ ls -l /usr/bin/write
-rwxr-sr-x 1 root tty 10124 Jan 27 2024 /usr/bin/write
```

When the setgid bit is enabled, the 'x' representing the execute-bit in the group sector of the mode is replaced by an 's'. You can see that the `write` executable is in the `tty` group and its setgid bit is enabled. When we run `write`, the process that executes it runs with the effective group ID of the `write` program, which is the `tty` group. This implies that the `write` command will be able to write to any terminal whose group-write bit is set. Since it can be annoying to receive messages on your terminal while you're working, Unix provides a simple command to query, enable, or disable this bit:

```
$ mesg [ y/n ]
```

If you enter **`mesg`** alone, it will display `y` or `n`, depending on whether the bit is set. Entering **`mesg y`** enables writing and **`mesg n`** turns it off.

### *The sticky Bit*

The sticky bit, also called the *save-text-image bit*, serves two different purposes when it is applied to files and directories. Originally, Unix was a pure swapping operating system — processes were swapped in and out of memory to maintain the multiprogramming level. The swapping store was a separate disk or a separate partition of a disk that was used exclusively for storing pro-

cess images when they were swapped out. The executable code and other data were kept in contiguous bytes on the swapping store, making reads and writes faster.

A program that was used by many people might go through many memory loads and unloads each day. Putting it in the swapping store made loads and unloads easier, because the file was in one piece. Setting the sticky bit on a program file prevented it from being removed from the swapping store.

If a directory has the sticky bit enabled, then a file that someone creates in the directory will be protected from being deleted or renamed by anyone except that person, the directory's owner, and a process with superuser privileges. Setting the sticky bit on a directory lets all processes put files into it in such a way that only processes with the same effective user ID as the process that created the file can remove those files. For example, Unix systems set the sticky bit on the directory */var/tmp* so that it can be used as a place for processes to write temporary files. If you have that directory on your computer, try this experiment:

```
$ touch /var/tmp/emptyfile    # create or update a file in /var/tmp
$ ls /var/tmp/     # prove that  it's there
emptyfile
--snip--
```

If you have a second user on your host, ask them to delete that file to see if they can. They won't be able to, but you can.

## An Example lstat Program

Let's turn our attention to the `stat()` system call. The `stat (2)` man page has an `EXAMPLES` section containing a complete program that calls `lstat()` to print out a file's metadata. Since the `stat()` and `lstat()` calls differ only in how they treat symbolic links, we can use this program to see how we can print the members of the `stat` structure returned by the `stat()` system call also.

> **SAMPLE CODE IN MAN PAGES**
>
> Some man pages have a `EXAMPLES` section containing one or more example programs to help us understand how to use the function they describe. These example programs are often very helpful and can be used as a good starting point for writing a program, since they will compile and build successfully and are usually documented well enough so that we can understand how to use the functions from that man page.

The program, which I've copied into a file named *lstat_manpage_example.c*, is reproduced in the following listing. I've added a few comments.

*lstat_manpage_example.c*

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

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/sysmacros.h> /* needed for major() and minor() */

int main(int argc, char *argv[])
{
    struct stat sb;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <relative_pathname>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (lstat(argv[1], &sb) == -1) {
        perror("lstat");
        exit(EXIT_FAILURE);
    }
 ❶ printf("ID of containing device:  [%jx,%jx]\n",
            (uintmax_t) major(sb.st_dev),
            (uintmax_t) minor(sb.st_dev));
    printf("File type:                ");
 ❷ switch (sb.st_mode & S_IFMT) {
        case S_IFBLK:  printf("block device\n");            break;
        case S_IFCHR:  printf("character device\n");        break;
        case S_IFDIR:  printf("directory\n");               break;
        case S_IFIFO:  printf("FIFO/pipe\n");               break;
        case S_IFLNK:  printf("symlink\n");                 break;
        case S_IFREG:  printf("regular file\n");            break;
        case S_IFSOCK: printf("socket\n");                  break;
        default:       printf("unknown?\n");                break;
    }

    printf("I-node number:            %ju\n",  sb.st_ino);
    printf("Mode:                     %o (octal)\n", sb.st_mode);
    printf("Link count:               %ju\n", (uintmax_t) sb.st_nlink);
    printf("Ownership:                UID=%ju   GID=%ju\n",
           ❸ (uintmax_t) sb.st_uid, (uintmax_t) sb.st_gid);
    printf("Preferred I/O block size: %jd bytes\n",
           (intmax_t) sb.st_blksize);
    printf("File size:                %jd bytes\n",
           (intmax_t) sb.st_size);
    printf("Blocks allocated:         %jd\n",
           (intmax_t) sb.st_blocks);

    /* These next instructions use the older timestamp names. For example,
    rather then accessing &sb.st_ctim.tv_sec, it accesses &sb.st_ctime */
```

```
    printf("Last status change:       %s", ctime(&sb.st_ctime));
    printf("Last file access:         %s", ctime(&sb.st_atime));
    printf("Last file modification:   %s", ctime(&sb.st_mtime));
    exit(EXIT_SUCCESS);
}
```

We can make several observations about the code.

- The `major()` and `minor()` functions extract the major and minor device ids from the `st_dev` member of the structure, which are both cast to `(uintmax_t)`. The `printf()` function❶ is given the format specification `%jx` to print each value. The man page for `printf(3)` explains that `j` is a *length modifier* that we use when the integer conversion (in this case a hexadecimal conversion `x`) following it corresponds to an `intmax_t` or `uintmax_t` argument. If we look up these two functions, we see that they return an `unsigned int`, which is cast to `uintmax_t`.
- The `switch` statement❷ expression uses the macro bit masks explained in the preceding section. The expression
  `switch (sb.st_mode & S_IFMT)`
  is compared against each of the file type masks described there.
- Most of the other values are cast to either `uintmax_t` or `intmax_t` in this program❸. If they were not cast, we would replace the `j` length modifier by the `l` (for `long`) modifier.
- The program converts timestamp values to strings using `ctime()`. Our programs thus far have used a combination of `localtime()` and `strftime()` so that they are locale-aware.
- This program does not convert the permission bits to their string representation, but leaves them in octal notation. With what we know now, we can make the permission output more human-friendly.

If we build the executable, named `lstat_manpage_example`, we can run it and see what its output is:

```
$ lstat_manpage_example /var/log/lastlog
ID of containing device:  [8,33]
File type:                regular file
I-node number:            917611
Mode:                     100664 (octal)
Link count:               1
Ownership:                UID=0   GID=43
Preferred I/O block size: 4096 bytes
File size:                292292 bytes
Blocks allocated:         576
Last status change:       Fri Aug 25 10:23:49 2023
Last file access:         Sun Sep 10 15:30:51 2023
Last file modification:   Fri Aug 25 10:23:49 2023
```

Although the program doesn't attempt to mimic the output of the `stat` command, it extracts the data from all available members of the `stat` structure and displays it in a human-readable form. It doesn't convert the octal permissions in the mode to string form; soon we'll consider how to do that.

Since this program uses `lstat()` instead of `stat()`, let's see how it treats symbolic links. First, let's create a symbolic link to */var/log/lastlog* in our working directory and run the program with the link as its argument:

```
$ ln -s /var/log/lastlog ./ll  # ll is a soft link to lastlog
$ lstat_manpage_example ll
ID of containing device:  [8,13]
File type:                symlink
I-node number:            6690311
Mode:                     120777 (octal)
Link count:               1
Ownership:                UID=500   GID=500
Preferred I/O block size: 4096 bytes
File size:                16 bytes
Blocks allocated:         0
Last status change:       Sat Sep  2 09:19:55 2023
Last file access:         Sun Sep 11 20:22:36 2023
Last file modification:   Sat Sep  2 09:19:55 2023
```

Comparing this output to the preceding listing, we see that `lstat()` prints information about the link itself, not the link's target. Notice that the link has its own inode number, that it has no data blocks, and that its timestamps are different from its target, `/var/log/lastlog`. Also notice that the permission bits that it displays for a symbolic link are 0777. Recall that POSIX.1-2017 does not require the permission bits in the returned `st_mode` of a symbolic link to have any meaning.

How does the `stat` command treat symbolic links? The man page showed that the `stat` command has a `-L` option. Without this option, when the command is given a symbolic link as its argument, it displays information about the link itself, not the link's target:

```
$ stat ll
  File: ll -> /var/log/lastlog
  Size: 16         Blocks: 0          IO Block: 4096   symbolic link
Device: 813h/2067d Inode: 6690311     Links: 1
Access: (0777/lrwxrwxrwx)  Uid: (  500/ stewart)   Gid: (  500/ stewart)
Access: 2023-09-11 20:22:36.919872028 -0400
Modify: 2023-09-02 09:19:55.041155595 -0400
Change: 2023-09-02 09:19:55.041155595 -0400
 Birth: 2023-09-02 09:19:55.041155595 -0400
```

The filename part of the output shows that *ll* is a symbolic link to */var/log/lastlog*, that it's just 16 bytes in size, and that it has no data blocks. With the `-L` option, `stat` displays information about the target:

```
$ stat -L ll
  File: ll
  Size: 292292     Blocks: 576        IO Block: 4096   regular file
Device: 833h/2099d Inode: 917611      Links: 1
Access: (0664/-rw-rw-r--)  Uid: (    0/    root)   Gid: (   43/    utmp)
Access: 2023-09-11 20:22:36.956868708 -0400
Modify: 2023-08-25 10:23:49.742860606 -0400
Change: 2023-08-25 10:23:49.742860606 -0400
 Birth: 2023-03-10 08:09:20.962419140 -0500
```

The file name listed is *ll*, but the metadata is that of its target, */var/log/lastlog*. For files that are not symbolic links, the output is identical.

We could use the example code from the man page as a starting point to design a program that behaves like the `stat` command. However, looking back at its output, we don't see the file creation time. That's because the `stat` structure returned by the `stat()` family of system calls doesn't contain a timestamp for it. On the other hand, the `stat` command displays the file's creation time, called the *birth time* in its output. The fact that the `stat` command can display file creation time but that it isn't in the `stat` structure returned by the `stat()` or the `lstat()` system calls merits further investigation, since it must be getting this timestamp in another way.

At the bottom of the `stat()` page, the `SEE ALL` section references `statx()`, another system call. The `inode (7)` man page also mentions it. We also know that the inode contains the file creation time in Ext4, but that it wasn't part of the inode in Ext2 and many other filesystems. If we want to write a program that can display file creation time, we can't use `stat()`, but perhaps we can use `statx()`. Let's see what its man page has to say.

## The statx() System Call

The `statx()` man page informs us that it's an extended version of `stat()`, returning information in a `statx` structure rather than a `stat` structure. The `VERSIONS` and `CONFORMING TO` sections of the page note that both the call and the data structure are later additions to Linux, appearing first in kernel version 4.11 (in 2017) and that they are Linux-specific. Therefore, it isn't necessarily available in other Unix systems and hence programs calling `statx()` may not be portable.

Let's examine the `statx()` prototype shown in the `SYNOPSIS` on the man page:

```
SYNOPSIS
    #include <sys/types.h>
    #include <sys/stat.h>
    #include <unistd.h>
    #include <fcntl.h>   /* Definition of AT_* constants */

    int statx(int dirfd, const char *pathname, int flags,
              unsigned int mask, struct statx *statxbuf);
```

The `statx()` function has five parameters, unlike `stat()`, which has two, and calling it is a bit more complex. Also, the synopsis lists four header files that must be included to call this function, but if you're using a Linux system with a version of *glibc* older than 2.28, your program will need to include different header files. In particular, it will require the kernel header files *linux/stat.h* and *linux/fcntl.h*. In Chapter 3, we saw a few different methods for checking the version of glibc.

### The statx Data Structure

We start by examining the `statx` structure returned by the function, which is its last parameter. The definition from the man page follows.

```
struct statx {
    __u32 stx_mask;        /* Mask of bits indicating
                              filled fields */
    __u32 stx_blksize;     /* Block size for filesystem I/O */
    __u64 stx_attributes;  /* Extra file attribute indicators */
    __u32 stx_nlink;       /* Number of hard links */
    __u32 stx_uid;         /* User ID of owner */
    __u32 stx_gid;         /* Group ID of owner */
    __u16 stx_mode;        /* File type and mode */
    __u64 stx_ino;         /* Inode number */
    __u64 stx_size;        /* Total size in bytes */
    __u64 stx_blocks;      /* Number of 512B blocks allocated */
    __u64 stx_attributes_mask;
                           /* Mask to show what's supported
                              in stx_attributes */

    /* The following fields are file timestamps             */
    struct statx_timestamp stx_atime;  /* Last access       */
    struct statx_timestamp stx_btime;  /* Creation          */
    struct statx_timestamp stx_ctime;  /* Last status change */
    struct statx_timestamp stx_mtime;  /* Last modification  */

    /* If this file represents a device, then the next two
       fields contain the ID of the device */
    __u32 stx_rdev_major;  /* Major ID */
    __u32 stx_rdev_minor;  /* Minor ID */

    /* The next two fields contain the ID of the device
       containing the filesystem where the file resides     */
    __u32 stx_dev_major;   /* Major ID */
    __u32 stx_dev_minor;   /* Minor ID */
};

The file timestamps are structures of the following type:
```

```
struct statx_timestamp {
    __s64 tv_sec;    /* Seconds since the Epoch (UNIX time)  */
    __u32 tv_nsec;   /* Nanoseconds since tv_sec             */
};
```

This structure differs from the `stat` structure in that it has extra members, and the member types are different. We'll discuss these types shortly. The additional members in the `statx` structure are:

**`stx_mask`** This has bits to indicate which other fields of the structure have been filled in by the kernel.

**`stx_attributes`** This contains the bitwise-or of various flags that indicate additional attributes of the file, such as whether it's compressed or encrypted.

**`stx_attributes_mask`** This indicates which bits in the mask are actually used.

**`stx_btime`** It is referred to as the *birth time* in the documentation, but is also called the *file creation time*.

Since this structure does contain the birth time of the file, we can use the `statx()` function to implement our version of the `stat` command, but we need to read more of the man page to understand how to call the function and how to use the returned data.

Let's start with the data types of the structure's members. First, the timestamp members such as `stx_mtime` have type `struct statx_timestamp`, unlike the corresponding members of the `stat` structure, whose types are each `struct timespec`. The members of these structures have the same names, but the underlying integer types of the members differ.

The remaining members of the `statx` structure are either `__u16`, `__u32`, or `__u64`. It doesn't declare any members using system data types such as `uid_t`. Although we can take an educated guess that `__u32` is an unsigned 32-bit integer and `__u64` is an unsigned 64-bit integer, we don't really know that for sure. Finding confirmation of this guess is not so easy though. They're not mentioned in the `system_data_types` man page, nor are they native types in the C language. We can attempt various man page searches but none will turn up a page that describes these types. We might be tempted to search on the web for guidance, but before resorting to what might yield an unreliable answer on-line, we can try a more extensive search on our Linux host.

We can confirm that these are not native C types by reading the most recent C standard, *C17*[4], a draft copy of which is free from the International Standards Organization at *https://iso-9899.info/wiki/The_Standard*. Therefore, they must be defined in Linux. Since all type and function definitions exposed to user-space programs are in a header file somewhere, we can do a recursive `grep` search for the pattern `__u64`, starting in */usr/include*, which is the root of all included header files in user-space, piping the output through a pager:

```
$ grep -R '__u64' /usr/include | more  # -R for recursive search
```

```
asm-generic/int-ll64.h:31:__extension__ typedef unsigned long long __u64;
asm-generic/int-ll64.h:34:typedef unsigned long long __u64;
asm-generic/statfs.h:49: __u64 f_blocks;
--snip--
asm-generic/statfs.h:76: __u64 f_ffree;
asm-generic/int-l64.h:30:typedef unsigned long __u64;
--snip--
```

Fortunately, the very first file, */usr/include/asm-generic/int-ll64.h*, has a `typedef` for this type, as does the similarly-named file, */usr/include/asm-generic/int-l64.h*. The initial comments in both files explain. In the first we see:

```
/*
 * asm-generic/int-ll64.h
 *
 * Integer declarations for architectures which use "long long"
 * for 64-bit types.
 */
```

and in the second:

```
/*
 * asm-generic/int-l64.h
 *
 * Integer declarations for architectures which use "long"
 * for 64-bit types.
 */
```

Our compiler will pull in the appropriate header file for our own machine based on definitions it found when it was installed. We now know that this is an unsigned 64-bit integer, regardless of how the C types `long` and `long long` are represented. The definition of `__u32` is also in these header files.

### INTEGER TYPES IN THE KERNEL

Within the Linux kernel, the code has to have a guarantee that the integer type it uses has a specified number of bits, such as 32 or 64, and has the correct signedness. The kernel code cannot rely on C types for this purpose because the standard C integer types are not the same size on all architectures. Therefore, the kernel uses integer types such as `s32`, `u32`, `s64`, and `u64` that are defined in such a way that they're guaranteed to have the correct number of bits.

Because some kernel data structures are exposed to user-space, some user-space header files contain declarations of types that correspond to those kernel types but whose names are preceded by double-underscores, such as `__s32`, `__u32`, `__s64`, and `__u64`.

Although we could use the `j` conversion modifier in the `printf()` conversions, since we know that these types are a fixed number of bits, we can

design the code that prints their values using C types. We just need to cast them to a corresponding C type and call `printf()` with the correct format conversions. Specifically, if `smallnum` and `bignum` are of types `__u32` and `__u64` respectively, then we would print them as follows:

```
printf("%lu" , (unsigned long) smallnum,  )
printf("%llu", (unsigned long long) bignum);
```

assuming we don't need to specify a minimum field width for formatting purposes.

### *Calling statx()*

Let's see how we use the other parameters of the `statx()` system call. We begin with how to specify the file whose metadata we want. The `statx()` function lets us specify that file by one of four different methods:

**An absolute pathname.** If the second argument, `pathname`, starts with a slash, such as `/var/log/lastlog`, it is an absolute pathname that specifies the target file and the first argument is ignored, as in:
`statx(0, "/var/log/lastlog",0,STATX_ALL, &statxbuf);`

**A relative pathname.** If `pathname` does not start with a slash and the first argument, `dirfd` is the macro constant `AT_FDCWD`, then the target file is the one specified by the given pathname relative to the current working directory. If the current working directory is */var/run*, then the file */var/log/lastlog* would be specified using the call:
`statx(AT_FDCWD, "../log/lastlog",0,STATX_ALL, &statxbuf);`

**A directory-relative pathname.** If `pathname` does not start with a slash and `dirfd` is an actual file descriptor that refers to a directory, then the target file is the one specified by the given pathname relative to the directory referred to by `dirfd`. If `varlog_fd` is a valid file descriptor for the directory */var/log*, then
`statx(varlog_fd, "lastlog",0,STATX_ALL, &statxbuf);`
refers to the file */var/log/lastlog* regardless of what the current working directory is at the time of the call.

**A file descriptor.** If `pathname` is an empty string and the macro constant `AT_EMPTY_PATH` is bitwise-or-ed into the third argument, `flags`, then the target file is the one referred to by the file descriptor in its first argument, which, in this case, does not have to refer to a directory. For example, if `fd` is a valid file descriptor for the file */var/loglastlog*, then
`statx(fd, "",AT_EMPTY_PATH,STATX_ALL, &statxbuf);`
refers to the file */var/log/lastlog*.

The second method, using a relative pathname, is easiest, and if we set the first parameter to `AT_FDCWD` but the pathname is absolute, then that parameter is ignored anyway, which means that the pathname may be either relative or absolute. We'll use this method in our program.

Let's turn to the third parameter of the function, which is a integer, a bitwise-or of a set of flags. Their purpose is to modify how the target file is identified. For example, we already saw the use of the AT_EMPTY_PATH flag. Another flag of interest is AT_SYMLINK_NOFOLLOW. When this flag is bitwise-or-ed into the parameter, if the pathname is a symbolic link, the function returns information about the link itself, rather than its target.

The fourth parameter, mask, is an unsigned integer that serves as a bitmask. It is how we can tell the kernel which metadata we want it to return in the structure, such as whether or not we want the timestamps or the file mode, and so on. The man page lists the constants that can be bitwise-or-ed into mask:

```
STATX_TYPE          Want stx_mode & S_IFMT
STATX_MODE          Want stx_mode & ~S_IFMT
STATX_NLINK         Want stx_nlink
STATX_UID           Want stx_uid
STATX_GID           Want stx_gid
STATX_ATIME         Want stx_atime
STATX_MTIME         Want stx_mtime
STATX_CTIME         Want stx_ctime
STATX_INO           Want stx_ino
STATX_SIZE          Want stx_size
STATX_BLOCKS        Want stx_blocks
STATX_BASIC_STATS   [All of the above]
STATX_BTIME         Want stx_btime
STATX_ALL           [All currently available fields]
```

There is no constant to mask the stx_blksize field.

From its man page we know that the stat command lets us specify which data we want to display by using the -c option. These constant bitmasks could be used to implement that option, given that the program has parsed the command line and recorded which fields need to be printed. It would just set the bits of this mask to indicate the fields that it wants to display. However, there's a catch. The man page warns us that,

> It should be noted that the kernel may return fields that weren't requested and may fail to return fields that were requested, depending on what the backing filesystem supports. (Fields that are given values despite being unrequested can just be ignored.) In either case, stx_mask will not be equal to mask.

The kernel may choose to ignore the mask, and when statx() returns, the stx_mask bits indicate which fields have been assigned values by the kernel. Following is a code snippet that determines whether or not the stx_size field has been given a value and, if so, prints it, assuming that all variables in the statx() call have been declared and initialized:

```
if ( statx(AT_FDCWD, pathtofile, flags, mask, &statxbuf) == -1 )
     /* Handle error */
else {
```

```
--snip--
    if ( statxbuf.stx_mask & STATX_SIZE )
       /* Print statxbuf.stx_size */
}
```

A program would need an `if` statement like this one for each different field of the `statx` structure.

If we wanted to design the program with the ability to print only selected fields, we'd need to introduce new Boolean variables with names like `wants_size_field` and `wants_uid_field` and modify this code. After a call to `statx()`, when our program needs to print the data in the structure, it would check the values of these variables and only print them if the corresponding `wants_..._field` variable is set. For example, to print the file size only if the user requested it, our code would be something like this:

```
/* Set wants_size_field_ = TRUE if user requested it, FALSE if not */
--snip--
if ( statx(AT_FDCWD, pathtofile, flags, mask, &statxbuf) == -1 )
      /* Handle error */
else {
--snip--
    if ( wants_size_field && ( statxbuf.stx_mask & STATX_SIZE ) )
       /* Print statxbuf.stx_size */
}
```

The last issue we need to address before designing the main function is how to determine whether the given file argument is a symbolic link, and if it is, how to find the name of its target. The first problem is solved once the program has called `statx()`, because we can use the `stx_mode` member to get the file type and check whether it's a soft link with the macro `S_ISLNK()`:

```
if ( statx(AT_FDCWD, pathname, flags, mask, &statx_buffer) < 0 )
     /* Handle error */
else
    if ( S_ISLNK(statx_buffer.stx_mode))
       /* It's a sym link - process the link */
--snip--
```

A man page search solves how to find the name of the link's target:

```
$ apropos -s2 -a symbolic link
readlink (2)         - read value of a symbolic link
readlinkat (2)       - read value of a symbolic link
```

The `readlink()` system call has the prototype

```
ssize_t readlink(const char *pathname, char *buf, size_t bufsiz);
```

We give it the name of the link in its first argument, and the address of a character string and its length in the second and third arguments. It fills

in the character string with the pathname contained in the link itself, and returns the length of the string, or −1 on an error. The man page tells us that it doesn't add the terminating null byte, so our program has to append it to the returned string. For example,

```
if ( -1 == (nbytes = readlink(pathname, target, sizeof(target))))
    error_mssge(errno, "readlink");
else {
    target[nbytes] = '\0';  /* Add the null byte */
    printf(" File: %s -> %s\n", pathname, target);
    --snip--
```

We're now ready to design and implement a first version of the `stat` command, which we'll call `spl_stat`. This initial version will allow a single option, `-L`. If a file argument is a symbolic link, then if that option is supplied, it will report on the link's target, and if the option is not supplied, it will report on the link itself, like the `stat` command.

## Writing a spl_stat Command

We'll develop this program following a top-down strategy. We'll start with the main program and design the required utility functions afterward, using stubs in their place.

### *Designing the main Function*

The `main()` function logic is relatively simple:

1. Initialize variables such as the mask and flags. For the default behavior, the mask should be `STATX_BASIC_STATS | STATX_BTIME`.
2. Initialize a flag variable named `report_link_data` to contain the flag `AT_SYMLINK_NOFOLLOW`.
3. Localize the program by calling `setlocale()`.
4. Parse the command line for options and arguments. If the `-L` option is found, set `report_link_data` to 0, so that the program reports on link targets instead of links.
5. If the command line is incorrect, exit with a usage message.
6. For each *`pathname`* found on the command line,
    - Call
      `statx(AT_FDCWD, pathname, report_link_data, mask, &statx_buffer)`
    - If the call was not successful, print an error message and skip to the next file.
    - If the call was successful, determine if `pathname` is a symbolic link. If it is, and `report_link_data` is 0, print the name of the target. If it is but `report_link_data` is not 0, print the link and target in the form *`pathname -> link-target`*. If `pathname` is not

a symbolic link, just print its name. In all cases, print the fields of `statx_buffer` afterward.

Let `print_statx()` be the name of the function that prints the fields of the returned `statx` structure. It will have two parameters, the address of a `statx` structure, and an array of integers:

```
void print_statx( struct statx *stx_buf, int what2print[] );
```

The array parameter will have a value for each field of the structure. If that value is 0, the function will not print it. If it's a 1, it will, provided that the field has been given a value in the structure. In this first version of the program, all elements of the array will be set to 1. In the second version, we'll add logic to `main()` to allow the user to select the fields to display.

The main program follows, with most comments omitted to save space. The complete program is in the book's source code distribution.

*spl_stat.c main()*

```
#define _GNU_SOURCE    /* Needed to expose statx() function in glibc */
#include <sys/stat.h>  /* Required for statx()                       */
#include "common_hdrs.h"
#include <locale.h>    /* For localization                           */

#define LOCALE_ERROR        -3
#define NUM_FIELDS          13    /* Number of fields in statx structure */
#include "../include/common_hdrs.h"
#include "../include/common_defs.h"

void print_statx(struct statx *stx, int what2print[]);

int main(int argc, char **argv)
{
    struct statx statx_buffer;    /* statx structure filled by statx()       */
    char         usage_mssge[128];/* string to store usage message           */
    unsigned int mask;            /* mask to pass to statx()                 */
    char         options[] = "L"; /* String for getopt() processing          */
    int          report_link_data;/* Flag indicating whether to report on link*/
    ssize_t      nbytes;          /* Return value of readlink()              */
    char         target[256];
    int          to_print[NUM_FIELDS];
    int          i;
    char         ch;

    mask = STATX_BASIC_STATS | STATX_BTIME;
    for ( i = 0; i < NUM_FIELDS; i++ ) to_print[i] = 1;

    /* Default behavior is to report on symbolic links, not their targets.    */
    report_link_data = AT_SYMLINK_NOFOLLOW; /* See the man page. */

    if ( setlocale(LC_TIME, "") == NULL )
```

```
    fatal_error(LOCALE_ERROR,"setlocale() could not set the given locale");

while  (TRUE) {
    ch = getopt(argc, argv, options);
    if ( -1 == ch ) /* No more options  */
        break;
    switch ( ch ) {
    case 'L':      /* follow symbolic links and report on their targets. */
        report_link_data = 0;
        break;
    case '?' :
        fprintf(stderr,"Found invalid option %c\n", optopt);
        sprintf(usage_mssge, "%s [-L ] files ...\n", basename(argv[0]));
        usage_error(usage_mssge);
        break;
    }
}
/* If no file arguments, print a usage message. */
if (optind >= argc) {
    sprintf(usage_mssge,"usage: %s [-L] files ...\n", basename(argv[0]));
    usage_error(usage_mssge);
}
/* For each file argument, call statx() and print its metadata. */
for ( i = optind; i < argc; i++) {
    if ( statx(AT_FDCWD, argv[i], report_link_data, mask,
        &statx_buffer) < 0 )
         printf("Could not stat file %s\n", argv[i]);
    else {
        if ( S_ISLNK(statx_buffer.stx_mode)) {  /* File's a soft link */
            if ( report_link_data == AT_SYMLINK_NOFOLLOW) {
                /* Report is of the link itself, not its target, so
                   write the file name in the form 'link -> target' */
                errno = 0;
                if ( -1 == (nbytes = readlink(argv[1], target,
                    sizeof(target))))
                    error_mssge(errno, "readlink");
                else {
                    target[nbytes] = '\0';
                    printf("  File: %s -> %s\n", argv[i], target);
                }
            }
            else /* Report is of the target */
                printf("  File: %s\n", argv[i]);
        }
        else
            printf("  File: %s\n", argv[i]);
```

```
                print_statx(&statx_buffer, to_print);
            }
            /* If there's another file, print a dashed separator line */
            if ( i < argc-1 )
                printf("---------------------------------"
                       "----------------------------------------\n");
        }
        return 0;
}
```

*Listing 7-1: The `main()` function of our implementation of the `stat` command*

Basically, the program performs its initializations, sets the locale, gets command-line options and then for each file on the command line, invokes `statx()` and prints out the data in the returned `statx` structure. It prints a dashed line between each file's output, unlike the actual command.

### Designing the print_statx() Function

Writing the function that prints the metadata is mostly an exercise in formatting information properly. Though it might seem tedious, it's a worthwhile endeavor to learn how to use `printf()`. One aspect of this is ensuring we use the correct flags, length modifiers, field widths, and conversion specifiers in the `printf()` format specification strings. The other aspect is trying to make our output conform to the way the actual command's output looks. This is less important, but it's also a good exercise in using `printf()`. The format of the output of `stat` may vary from one file to another and column positions may move slightly, so we use its output just as an approximation for how to format our program's output.

To facilitate planning the output, we redisplay the `stat` command's output here, together with a guide to help identify positions of the printed fields. This time we give it a device file so that we can see how it displays the device type:

```
$ stat /dev/pts/0
  File: /dev/pts/1
  Size: 0               Blocks: 0          IO Block: 1024   character special file
Device: 18h/24d Inode: 4           Links: 1     Device type: 88,1
Access: (0620/crw--w----)  Uid: (  500/ stewart)   Gid: (    5/     tty)
Access: 2023-09-10 10:34:16.146591494 -0400
Modify: 2023-09-10 10:34:16.146591494 -0400
Change: 2023-09-10 09:26:47.146591494 -0400
 Birth: -

123456789 123456789 123456789 123456789 123456789 123456789 123456789
         10        20        30        40        50        60
```

We'll shift the starting position of the text `Inode:...` to align with the `Blocks:...` above it, as in

```
$ stat /dev/pts/0
  File: /dev/pts/1
  Size: 0               Blocks: 0          IO Block: 1024   character special file
Device: 18h/24d         Inode: 4           Links: 1     Device type: 88,1
Access: (0620/crw--w----)  Uid: (  500/ stewart)   Gid: (    5/     tty)
--snip--
123456789 123456789 123456789 123456789 123456789 123456789 123456789
         10        20        30        40        50        60
```

The sequence of fields to be printed, with formatting information is shown in Table 7-1. Some of the information about the data formatting comes from the field type in the `stat` structure described in the man page. For example, `stx_size` is of type `__u64`. All values are supposed to be left-justified except the user ID and the group ID, which are right-justified. Therefore, the table omits information about justification. The timestamp fields all have the same format, indicated by the term *timestamp format*, which is a localized date/time, followed by a nine-digit number of nanoseconds and a timezone shift.

**Table 7-1:** Fields of the *statx* structure to print, with formatting information

| Data Member | Label | Starting Column | Formatting Information |
|---|---|---|---|
| pathname | `File:` | 1 | `char*` printed by `main()` |
| `stx_size:` | `Size:` | 3 | `long unsigned int` |
| `stx_blocks` | `Blocks:` | 25 | `long unsigned int` |
| `stx_blksize` | `IO Block:` | 44 | `unsigned int` |
| file type bits in `stx_mode` | none | 61 | `char*` |
| `stx_dev_major`, `stx_dev_minor` | `Device:` | 1 | single `long unsigned int` in the format hex/dec |
| `stx_ino` | `Inode:` | 25 | `long unsigned int` |
| `stx_nlinks` | `Links:` | 44 | `unsigned int` |
| `stx_rdev_major`, `stx_rdev_minor` | `Device type:` | 57 | two 32-bit unsigned integers in the format *d,d* |
| `stx_mode` | `Access:` | 1 | ( octal numeral of file mode/ permission string ) |
| `stx_uid` | `Uid:` | 28 | ( `unsigned int` / username ) |
| `stx_gid` | `Gid:` | 52 | ( `unsigned int` / group name ) |
| `stx_atime` | `Access:` | 1 | timestamp format |
| `stx_mtime` | `Modify:` | 1 | timestamp format |
| `stx_ctime` | `Change:` | 1 | timestamp format |
| `stx_btime` | `Birth:` | 1 | timestamp format |

Table 7-1 allows us to sketch out the `print_statx()` function, designing the `printf()` format specifications to match the corresponding data types and layout. Some of the members of the `statx` structure need no processing before they're printed, whereas others do. For those not requiring preprocessing, we can use a bit of arithmetic to determine the field widths and format conversion specifiers for the calls to `printf()`.

For example, to determine the field width for the `stx_blksize` value, we reason as follows. Because the label `"IO Block: "` starts in column 44 and is 10 characters long including the space character, and because the next print field is the file type, starting in column 61, the field width for `stx_blksize` should be 61 − 10 − 44 = 7. Since `stx_blksize` is a 32-bit type that we cast upward to `unsigned long` (to be safe), the format specifier should be `"%-7lu"`.

For those members that can be printed without any preprocessing, Table 7-2 shows the `printf()` format specifications based on these calculations. We'll visit how to print those members that do require some preprocessing shortly.

**Table 7-2:** Selected fields of the *statx* structure, with *printf()* format specifications for printing them

| Data | Member Name | printf() Format Specification |
|---|---|---|
| File size | `stx_size` | `" Size: %-16llu"` |
| Number of blocks | `stx_blocks` | `" Blocks: %-10llu"` |
| I/O block size | `stx_blksize` | `" IO Block: %-7lu"` |
| Inode number | `stx_ino` | `"Inode: %-11llu"` |
| Number of links | `stx_nlinks` | `" Links: %-5u"` |
| Device type | `stx_rdev_major`, `stx_rdev_minor` | `" Device type: %u,%u"` |
| File mode | `stx_mode` | `"Access: (%04o/%s)"` |
| user ID | `stx_uid` | `" Uid: (%5d / %s) "` |
| group ID | `stx_gid` | `" Gid: (%5d / %s) "` |

Let's turn our attention to the data that requires some preprocessing before printing. The particular problems that we need to solve are as follows:

- Although we can print the file mode as an octal number without any preprocessing, to print it as a permission string, we need a function that, given a 16-bit file mode, returns a permission string representing that mode. Its prototype will be `char* mode2str( int mode)`.
- Because the command displays both user ID and username, but we only have the user ID, we need a function that, given a user ID, returns the corresponding username as a string. If there is no username, it should return an empty string. This will have prototype `char* uid2name ( uid_t uid )`.
- Similarly, we need a function to return the group name, given a group ID. If there is no group name, it should return an empty string. This will have prototype `char* gid2name ( gid_t gid )`.
- We need to create the numeric representation of the device ids. The device ids are two separate 32-bit integers, but when they're printed by the `stat` command, they are encoded into a single number that is displayed in both hexadecimal and decimal, such as `813h/2067d`. Because the `makedev()` function mentioned in the `system data_types(7)` man page encodes the two values into a single integer, we can employ it here.

- We need to print the file type as a string, such as "regular file", based on the bits in the stx_mode member. We can use the method we described in "The File Mode" on page 306.
- The times displayed by the stat command contain nanosecond accuracy, as in 2010-09-07 10:31:41.823620843, and the timestamp members of the statx structure also store time accurate to the nanosecond, but the strftime() function that prints localized time is given a time argument accurate only to the second (struct tm). In order to format time to the nanosecond, we need to print the nanoseconds as a decimal integer after the string printed by strftime(). We'll create a single function that prints the label followed by the localized time, formatted to include the nanoseconds and time zone after it. Its prototype is
  void print_time(const char* label, struct statx_timestamp *time_field)

Before implementing the functions we just described, let's look at the code for the print_statx() function. The code makes calls to these functions using the specified prototypes for each. The field widths in the printf() specifiers are based on the starting columns we identified in Table 7-1. For fields that are left-justified; their specifiers start with a leading hyphen, as in "%-12lu", which specifies a left-justified field of width 12 for an unsigned long int.

For all fields except stx_blksize, printing their values is preceded by testing that the corresponding bit in the stx_mask has been set. The inline commennts provide further explanation.

*print_statx()*

```
void print_statx(struct statx *stx, int what2print[])
{
    char idstring[64];

    if (stx->stx_mask & STATX_SIZE)
        printf(" Size: %-16llu",   (unsigned long long)stx->stx_size);
    if (stx->stx_mask & STATX_BLOCKS)
        printf("Blocks: %-10llu", (unsigned long long)stx->stx_blocks);

    /* stx_blksize is always returned --- there is no mask for it. */
    printf(" IO Block: %-7lu", (unsigned long )stx->stx_blksize);

    /* Extract the file type from the stx_mode field with the S_IFMT mask.  */
    if (stx->stx_mask & STATX_TYPE)
        switch (stx->stx_mode & S_IFMT) {
            case S_IFIFO:  printf("  FIFO\n"theirtheir);        break;
            case S_IFCHR:  printf("  character special file\n");break;
            case S_IFDIR:  printf("  directory\n");             break;
            case S_IFBLK:  printf("  block special file\n");    break;
            case S_IFREG:  printf("  regular file\n");          break;
            case S_IFLNK:  printf("  symbolic link\n");         break;
            case S_IFSOCK: printf("  socket\n");                break;
            default:
```

```
            printf(" unknown type (%o)\n", stx->stx_mode & S_IFMT);
            break;
    }
else /* This should not happen, but just in case ... */
    printf(" no known type\n");

/* Print out the combined major and minor device ids in both
   hexadecimal and decimal. */
ids2hexdecstr(stx->stx_dev_major, stx->stx_dev_minor, idstring);
printf("Device: %-16s", idstring);

if (stx->stx_mask & STATX_INO)
    printf("Inode: %-11llu", (unsigned long long) stx->stx_ino);
if (stx->stx_mask & STATX_NLINK)
    printf(" Links: %-5lu", (unsigned long) stx->stx_nlink);
/* If the file is a device file, such as a terminal, disk, and so on, the
   statx structure will have the device's major and minor device ids in
   stx_rdev_major and stx_rdev_minor respectively. These are __u32 values.
   We cast upward in case the machine doesn't have 32-bit integers. */

if (stx->stx_mask & STATX_TYPE)
    switch (stx->stx_mode & S_IFMT) {
    case S_IFBLK:
    case S_IFCHR:
        printf(" Device type: %lu,%lu",
              (unsigned long) stx->stx_rdev_major,
              (unsigned long) stx->stx_rdev_minor);
        break;
    }
printf("\n");

/* Print the mode in the form (octal/permissionstr), such as
   (0644/-rw-r--r--).
   To get the first part, bitwise-and with 0777 to zero out the file type
   upper 4 bits and print a 4-char wide field in octal. The second part
   is the call to mode2str().  */
if (stx->stx_mask & STATX_MODE)
    printf("Access: (%04o/%s)", stx_mode & 0777, mode2str( (int) stx_mode);
if (stx->stx_mask & STATX_UID)
    printf("  Uid: (%5ld / %s)  ", (long) stx->stx_uid,
                      uid2name(stx->stx_uid));
if (stx->stx_mask & STATX_GID)
    printf("  Gid: (%5ld / %s)\n", (long) stx->stx_gid,
                      gid2name(stx->stx_gid));
if (stx->stx_mask & STATX_ATIME)
    print_time("Access: ", &stx->stx_atime);
if (stx->stx_mask & STATX_MTIME)
```

```
        print_time("Modify: ", &stx->stx_mtime);
    if (stx->stx_mask & STATX_CTIME)
        print_time("Change: ", &stx->stx_ctime);
    if (stx->stx_mask & STATX_BTIME)
        print_time(" Birth: ", &stx->stx_btime);
}
```

Let's turn our attention to the ancillary functions called by `print_statx()`, namely `char* mode2str()`, `uid2name()`, `gid2name()`, `ids2hexdecstr())`, and `print_time()`.

### *Writing the Auxiliary Functions*

Let's start with the `char* mode2str( int mode)` function. Its single argument is the file's mode, and its return value is a string pointer. Because it returns a pointer to a string, we declare a static string local to the function and return a pointer to it. Because it's static, it isn't on the stack and will stay in memory while the program is running.

*mode2str()*

```
char* mode2str( int mode)
{
    static char str[] = "----------";          /* Initial string   */

    if      ( S_ISDIR(mode) )  str[0] = 'd';    /* Directory         */
    else if ( S_ISCHR(mode) )  str[0] = 'c';    /* Char devices      */
    else if ( S_ISBLK(mode) )  str[0] = 'b';    /* Block device      */
    else if ( S_ISLNK(mode) )  str[0] = 'l';    /* Symbolic link     */
    else if ( S_ISFIFO(mode))  str[0] = 'p';    /* Named pipe (FIFO) */
    else if ( S_ISSOCK(mode))  str[0] = 's';    /* Socket            */

    if ( mode & S_IRUSR ) str[1] = 'r';
    if ( mode & S_IWUSR ) str[2] = 'w';
    if ( mode & S_IXUSR ) str[3] = 'x';

    if ( mode & S_IRGRP ) str[4] = 'r';
    if ( mode & S_IWGRP ) str[5] = 'w';
    if ( mode & S_IXGRP ) str[6] = 'x';

    if ( mode & S_IROTH ) str[7] = 'r';
    if ( mode & S_IWOTH ) str[8] = 'w';
    if ( mode & S_IXOTH ) str[9] = 'x';

    /* Now check the setuid, setgid, and sticky bits */
    if ( mode & S_ISUID ) str[3] = 's';
    if ( mode & S_ISGID ) str[6] = 's';
    if ( mode & S_ISVTX ) str[9] = 't';
    return str;
}
```

The function is relatively simple. It checks each bit of the `mode` argument. If it's set, it replaces the '-' in `str` by the corresponding letter, based on the bitmask constant bitwise-and-ed to it. After it checks the permission bits, it checks the special bits, and for each bit that's enabled, it replaces the corresponding execute bit in the string to either an 's' or a 't'.

The next two functions are `uid2name()` and `gid2name()`. They are nearly the same. In Chapter 6, we learned about the `getpwuid()` function, which returns a pointer to the `passwd` structure, given a user ID. We use that function to get the structure and return the username member of it:

*uid2name()*

```
#include <pwd.h>
char* uid2name ( uid_t uid )
{
    struct  passwd *pw_ptr;
    if ( ( pw_ptr = getpwuid( uid ) ) == NULL )
          return "";
    else
      return pw_ptr->pw_name ;
}
```

The required header file is shown in the listing as a reminder that we need to include it. If for some reason, there is no entry in the password database, it returns an empty string.

We haven't yet needed to work with group information, but a good guess would be that there's a corresponding set of functions for groups. The corresponding function for getting the group name, given the group ID, might be `getgrgid()`. A man page search will confirm this — entering **`apropos -a 'gid' 'get'`** lists several functions, but the ones that mention a file's group are the ones we need:

```
getgrgid (3)         - get group file entry
```

This function returns a pointer to a `group` structure, which has a member named `gr_name`. Our function can use this in the same way that `uid2name()` used `getpwuid()`:

*gid2name()*

```
#include <grp.h>
char* gid2name ( gid_t gid  )
{
    struct group *grp_ptr;
    if ( ( grp_ptr = getgrgid(gid) ) == NULL )
          return "";
    else
          return grp_ptr->gr_name;
}
```

The next task is to print the device ids in the required format. We encapsulate that logic in a function named `ids2hexdecstr()`). That function needs to construct a string in which the first part is the hexadecimal code for

the stx_dev_major and stx_dev_minor fields. The format specifier, "%02x%02xh" converts the next two integer values to hexadecimal with leading zeros without intervening space, each with a minimum field width of two characters, followed by the letter 'h'. For example,

```
printf("%02x%02xh", 255, 32);
```

prints ff20h.

The second part of the string is the decimal equivalent of the combined major and minor device ids. According to its man page, the makedev() function, given two 32-bit device ids, constructs and returns a single \dev_t (which is long unsigned int) device id:

```
#include <sys/sysmacros.h>
    dev_t makedev(unsigned int maj, unsigned int min);
```

This function is not required by POSIX.1-2017, implying that this code may not be portable. We could avoid using it if we want by converting the hexadecimal string we just constructed to a decimal using strtol() with a base of 16. We'll opt to use the less portable approach by calling makedev(), casting the dev_t return value to an unsigned long for printing:

```
#include <sys/sysmacros.h>
void ids2hexdecstr(unsigned int major, unsigned int minor, char* buffer)
{
    sprintf(buffer, "%02x%02xh/%lud", major, minor,  makedev(major, minor));
}
```

The last part of this program is the print_time() function. In Chapter 4 we learned how to format date/time strings. Examining the output of the stat command, we see that the dates and times are in the format

```
yyyy-mm-dd hh:mm:ss.ddddddddd -hh:mm
```

where the ddddddddd is a number of nanoseconds in the range [0,999999999]. Table C-1 in Appendix C lists the date and time format specifiers that we can give to stftime(). They're also in its man page. If we look at that list, we can see that "%F %T" is the format that would give us the date and time as yyyy-mm-dd hh:mm:ss. To print the nanoseconds in a left-justified field of nine characters with leading zeros we can use "%09u". The resulting function follows.

*print_time()*

```
void print_time(const char* label, struct statx_timestamp *time_field)
{
    struct tm *bdtime;                /* Broken-down time                */
    char       time_string[100];      /* String storing formatted time   */
    char       timezone[32];          /* To store time offset            */
    time_t     time_val;              /* For converted tv_sec field      */

    time_val = time_field->tv_sec;          /* Convert to time_t          */
```

```
    bdtime = localtime(&time_val);          /* Convert to broken-down time */
    if (bdtime == NULL)                     /* Check for error             */
        fatal_error(EOVERFLOW, "localtime");

    if ( strftime(time_string, sizeof(time_string), "%F %T", bdtime) == 0 )
        fatal_error(BAD_FORMAT_ERROR,"strftime failed\n");

    printf("%s%s.%09u", label, time_string, time_field->tv_nsec );
    if ( 0 == strftime(timezone, 32, " %z", bdtime) )
        fatal_error(BAD_FORMAT_ERROR, "Error printing time zone\n");
    printf("%s\n", timezone);
}
```

We've completed the program. To save space here, the complete listing is omitted, but is in the source code distribution for the book.

We compile and build the program with the command

```
$ gcc -Wall -g -I ../include spl_stat.c -L../lib -lspl -o spl_stat
```

and run it on a few different files, comparing the output to that of the actual `stat` command, as shown here:

```
$ spl_stat  spl_stat.c
 File  spl_stat.c
 Size: 10777           Blocks: 24         IO Block: 4096    regular file
Device: 0813h/2067d    Inode: 6690318     Links: 1
Access: (0664/-rw-rw-r--)  Uid: (  500 / stewart)    Gid: (  500 / stewart)
Access: 2023-09-10 15:31:57.754777183 -0400
Modify: 2023-09-10 15:31:52.954929789 -0400
Change: 2023-09-10 15:31:52.994928519 -0400
 Birth: 2023-09-10 15:31:52.950929916 -0400
$ stat spl_stat.c
  File: spl_stat.c
  Size: 10777         Blocks: 24         IO Block: 4096   regular file
Device: 813h/2067d    Inode: 6690318     Links: 1
Access: (0664/-rw-rw-r--)  Uid: (  500/ stewart)   Gid: (  500/ stewart)
Access: 2023-09-10 15:31:57.754777183 -0400
Modify: 2023-09-10 15:31:52.954929789 -0400
Change: 2023-09-10 15:31:52.994928519 -0400
 Birth: 2023-09-10 15:31:52.950929916 -0400
```

This first run shows that our program's output is almost identical to that of the command. Let's see how it compares when given a device file argument, which will require the device type to be printed.

```
$ spl_stat /dev/tty
  File: /dev/tty
  Size: 0               Blocks: 0          IO Block: 4096     character special file
Device: 0005h/5d        Inode: 12          Links: 1     Device type: 5,0
```

```
Access: (0666/crw-rw-rw-)  Uid: (    0/ root)    Gid: (    5/ tty)
Access: 2023-09-21 16:07:46.650574320 -0400
Modify: 2023-09-21 16:07:46.650574320 -0400
Change: 2023-09-21 16:07:46.650574320 -0400
 Birth: 2023-09-21 16:07:42.320000000 -0400
$ stat /dev/tty
  File: /dev/tty
  Size: 0               Blocks: 0          IO Block: 4096   character special file
Device: 5h/5d Inode: 12          Links: 1     Device type: 5,0
Access: (0666/crw-rw-rw-)  Uid: (    0/    root)   Gid: (    5/     tty)
Access: 2023-09-21 16:07:46.650574320 -0400
Modify: 2023-09-21 16:07:46.650574320 -0400
Change: 2023-09-21 16:07:46.650574320 -0400
 Birth: 2023-09-21 16:07:42.320000000 -0400
```

Our program prints leading zeros in the device id, whereas the `stat` command does not, otherwise the output is the same.

Finally, let's see how it behaves when the file is a symbolic link. We'll use the same *ll* link as before:

```
$ $ spl_stat -L ll
 File: ll
 Size: 292292          Blocks: 576        IO Block: 4096    regular file
Device: 0833h/2099d    Inode: 917611      Links: 1
Access: (0664/-rw-rw-r--)  Uid: (    0 / root)    Gid: (   43 / utmp)
Access: 2023-09-11 21:21:13.592747728 -0400
Modify: 2023-08-25 10:23:49.742860606 -0400
Change: 2023-08-25 10:23:49.742860606 -0400
 Birth: 2023-03-10 08:09:20.962419140 -0500

$ spl_stat ll
 File: ll -> /var/log/lastlog
 Size: 16              Blocks: 0          IO Block: 4096    symbolic link
Device: 0813h/2067d    Inode: 6690311     Links: 1
Access: (0777/lrwxrwxrwx)  Uid: (  500 / stewart)    Gid: (  500 / stewart)
Access: 2023-09-11 20:22:36.919872028 -0400
Modify: 2023-09-02 09:19:55.041155595 -0400
Change: 2023-09-02 09:19:55.041155595 -0400
 Birth: 2023-09-02 09:19:55.041155595 -0400
```

You can see that the program implements the `-L` option, because with it, it reports on the target, and without it, it reports on the link itself.

### Designing an Enhanced spl_stat Command

If we want to create a second version of the program that allows the user to suppress printing of selected fields, the only changes are in the option-handling code in the main program and in the `print_statx()` function. We

outline the changes here and leave development of the actual program as an exercise.

First, we declare an enumerated type:

```
enum field2print {
    typef, modef, nlinkf, uidf, gidf, atimef, mtimef, ctimef,
    inof, sizef, blocksf, blksizef, btimef, NUM_FIELDS };
```

We'll use this type as a set of index values into an array of flags named `to_print`, declared in the `main()` function. If `to_print[nlinkf]` is `FALSE` for example, then the program should not print the `stx_nlinks` data, and if `to_print[nlinkf]` is `TRUE` then it should. Initially all fields are suppressed:

```
BOOL to_print[NUM_FIELDS];
for ( i = typef; i < NUM_FIELDS; i++ )
    to_print[i] = FALSE;
```

Next, in `main()`, we need to parse the option string. This is the most work. We don't have to use the same syntax as the `stat` command. We could simplify it and use options of the form

```
-a  # show atime
-b  # show btime
-c  # show ctime
-t  # show type
-p  # show mode
--snip--
```

choosing unique, mnemonic letters for each field when possible. If we want to be faithful to the syntax of the actual `stat` command, then we'd need to write a separate function that parsed a string of the form `-c" %a  %w ... "` matching the options of that command. We'll choose the simpler method here. In the option-handling loop, when an option is found, the corresponding flag in the array would be set:

```
int main(int argc, char **argv)
    char         options[] = "abctpL...";
    BOOL         to_print[NUM_FIELDS];
    char         ch;
--snip--

    /* Parse the command line for options. */
    while  (TRUE) {
        /* Call getopt, passing argc and argv and the options string */
        ch = getopt(argc, argv, options);
        if ( -1 == ch ) /* No more options  */
            break;
        switch ( ch ) {
        case 'a': to_print[atimef] = TRUE; break;
        case 'b': to_print[btimef] = TRUE; break;
```

```
        case 'c': to_print[ctimef] = TRUE; break;
        case 'n': to_print[nlinkf] = TRUE; break;
__--snip--
  _      }
    }
```

The last changes would be in the `print_statx()` function. We would need to replace every `if` statement such as

```
if (stx->stx_mask & STATX_ATIME)
```

by

```
if (to_print[atimef] && stx->stx_mask & STATX_ATIME)
```

If we make these changes throughout the function then only those fields requested by the user will be printed. However, the formatting will not be the same as it is in the original program. The actual `stat` command does not attempt to preserve that formatting. We could, if we wanted, replace the missing fields by blank strings of the same length.

## Writing a spl_statfs Command

Let's return to the original problem we wanted to solve, namely developing a program that can display the metadata of a filesystem. What we learned while developing a `stat` command is good preparation for this task.

### *The statfs() System Call*

Earlier we discovered a system call named `statfs()` that prints a filesystem's metadata. We begin by looking at its man page:

```
$ man -s2 statfs
STATFS(2)                 Linux Programmer's Manual                STATFS(2)

NAME
       statfs, fstatfs - get filesystem statistics

SYNOPSIS
       #include <sys/vfs.h>    /* or <sys/statfs.h> */

       int statfs(const char *path, struct statfs *buf);
       int fstatfs(int fd, struct statfs *buf);

DESCRIPTION
       The  statfs()  system call returns information about a mounted filesys
       tem.  path is the pathname of any file within the  mounted  filesystem.
       buf  is  a  pointer to a statfs structure defined approximately as fol
       lows:
```

```
struct statfs {
    __fsword_t f_type;    /* Type of filesystem (see below) */
    __fsword_t f_bsize;   /* Optimal transfer block size */
    fsblkcnt_t f_blocks;  /* Total data blocks in filesystem */
    fsblkcnt_t f_bfree;   /* Free blocks in filesystem */
    fsblkcnt_t f_bavail;  /* Free blocks available to
                             unprivileged user */
    fsfilcnt_t f_files;   /* Total inodes in filesystem */
    fsfilcnt_t f_ffree;   /* Free inodes in filesystem */
    fsid_t     f_fsid;    /* Filesystem ID */
    __fsword_t f_namelen; /* Maximum length of filenames */
    __fsword_t f_frsize;  /* Fragment size (since Linux 2.6) */
    __fsword_t f_flags;   /* Mount flags of filesystem
                             (since Linux 2.6.36) */
    __fsword_t f_spare[xxx];
                   /* Padding bytes reserved for future use */
};
```

This is probably the function we need. This system call returns various pieces of information about the filesystem containing the file specified in its first parameter. That information is returned in the `statfs` structure, whose address is the second parameter.

Before we study the `statfs` data structure in detail, let's read more of the man page, particularly the relevant remarks and warnings.

- In the `CONFORMING TO` section, it states that this call is only available in Linux, implying that our code won't be portable unless we find alternative methods that are more portable and include macros to check on which system the program is compiled.
- In the `NOTES` section, it mentions that the `__fsword_t` type is an internal type in glibc that may not be recognized by our compiler. The recommended solution is to cast variables of this type to `unsigned int`, with the warning that it may not work.
- The page also warns us that, "Nobody knows what `f_fsid` is supposed to contain (but see below)." The following explanation elaborates — the `f_fsid` field is supposed to contain the filesystem's unique ID. Different Unix distributions return that ID in different ways. in Linux, the `f_fsid` field is of type `fsid_t` which is defined in *sys/vfs.h* as

```
struct { int val[2];}
```

- The *Linux Standard Base* (*LSB*), another standard developed under the auspices of the Linux Foundation, (see *https://wiki.linuxfoundation.org/lsb/lsb-introduction*) has deprecated the library call `statfs()` and advises us to use `statvfs()` instead. In fact, if we look through the POSIX list of system calls and library functions, we don't find `statfs(2)` but we do find `statvfs()`.

In short, the man page has enough discouraging warnings that we should consider the alternatives before making any decisions about how to design our program.

### The statvfs() Library Function

We begin by reading the `statvfs()` man page, in the hope that we'll be able to use this function instead.

```
STATVFS(3)             Linux Programmer's Manual             STATVFS(3)

NAME
       statvfs, fstatvfs - get filesystem statistics

SYNOPSIS
       #include <sys/statvfs.h>

       int statvfs(const char *path, struct statvfs *buf);
       int fstatvfs(int fd, struct statvfs *buf);

DESCRIPTION
       The  function statvfs() returns information about a mounted filesystem.
       path is the pathname of any file within the mounted filesystem. buf is a
       pointer to a statvfs structure defined approximately as follows:

           struct statvfs {
               unsigned long  f_bsize;    /* Filesystem block size */
               unsigned long  f_frsize;   /* Fragment size */
               fsblkcnt_t     f_blocks;   /* Size of fs in f_frsize units */
               fsblkcnt_t     f_bfree;    /* Number of free blocks */
               fsblkcnt_t     f_bavail;   /* Number of free blocks for
                                             unprivileged users */
               fsfilcnt_t     f_files;    /* Number of inodes */
               fsfilcnt_t     f_ffree;    /* Number of free inodes */
               fsfilcnt_t     f_favail;   /* Number of free inodes for
                                             unprivileged users */
               unsigned long  f_fsid;     /* Filesystem ID */
               unsigned long  f_flag;     /* Mount flags */
               unsigned long  f_namemax;  /* Maximum filename length */
           };

       Here  the  types  fsblkcnt_t  and fsfilcnt_t are defined in <sys/types.h>.
       Both used to be unsigned long.
--snip--
```

This is a library function, not a system call. The rest of the man page explains that it is supported by calls to `statfs()` in Linux. The statement that

the structure is "defined approximately" is not explained further, other than that some members will not have meaningful values on some filesystems.

On the positive side, the filesystem ID in this structure is a simple integer and there are no members whose types are internal *glibc* types. On the negative side, it doesn't have the `f_type` field, so we can't use this function to get the filesystem type. Lastly, the man page explains that, to get all mounting flags bitwise-or-ed into `f_flags`, we need to define `_GNU_SOURCE`.

If we use this function, we need a different method of getting the filesystem type. If we use `statfs()`, our code will only work on Linux systems. This problem is unlike the problems we tackled in the previous chapters of the book, which all fell under the parts of the API standardized by POSIX.1-2017, because the lack of a single standard means that we either have to write a very complex program to make it portable, or write a simpler one that we know will only work on Linux.

It would be useful to see the filesystem type in our output. Getting the filesystem type without calling `statfs()` is much harder because we'd need to learn about how filesystems are mounted and where their types are stored in the kernel's data structures for maintaining information about mounts.

### A Hybrid Solution

Given the preceding arguments, we'll follow a hybrid approach; we'll use `statvfs()` to get most of the metadata that we want to display, and call `statfs()` to get the filesystem type and nothing more.

As we did when we designed the `spl_stat` program, we'll use the output of `stat -f` as a guide to format the output of our program. Let's display that output again to see its format, putting column numbers underneath to help plan our output:

```
$ stat -f /var/log
  File: "/var/log"
    ID: 7e58ac747798a2a5 Namelen: 255     Type: ext2/ext3
Block size: 4096       Fundamental block size: 4096
Blocks: Total: 11936443   Free: 10290961   Available: 9676510
Inodes: Total: 3055616    Free: 3020523

123456789 123456789 123456789 123456789 123456789 123456789
0         10        20        30        40        50
```

We need to map the displayed data items to the members of the `statvfs` structure. For most of the labeled data shown, reading the `statvsf(3)` and `stat(1)` man pages is enough determine to which members of the structure they correspond. The one member that does not seem obvious is `f_frsize`, but by process of elimination we can conclude it must be the data appearing after the label `Fundamental block size`. The correspondence is therefore as follows:

**ID** : `f_fsid`

**Namelen** : `f_namemax`

**Type** : `f_type`, not part of `struct statvfs`

**Block size** : `f_bsize`

**Fundamental block size** : `f_frsize`

**Total blocks** : `f_blocks`

**Free blocks** : `f_bfree`

**Available** : `f_bavail`

**Total inodes** : `f_files`

**Free inodes** : `f_ffree`

The filesystem type, is not part of that structure; we'll get it by calling `statfs()`. Table 7-3 shows the formatting information that the preceding output implies.

**Table 7-3:** Fields output by *stat -f* with formatting information and either *struct statvfs* member names or other source of data

| Data Member | Label | Starting Column | Type |
|---|---|---|---|
| command arg | `File:` | 3 | char*, printed by main function |
| `f_fsid` | `ID:` | 5 | Hexadecimal unsigned long int |
| `f_namemax` | `Namelen:` | 25 | Unsigned long integer |
| external | `Type:` | 43 | char* |
| `f_bsize` | `Block size:` | 0 | Unsigned long integer |
| `f_frsize` | `Fundamental block size:` | 24 | Unsigned long integer |
| `f_blocks` | `Blocks: Total:` | 0 | Unsigned long integer |
| `f_bfree` | `Free:` | 27 | Unsigned long integer |
| `f_bavail` | `Available:` | 44 | Unsigned long integer |
| `f_files` | `Inodes: Total:` | 0 | Unsigned long integer |
| `f_ffree` | `Free:` | 27 | Unsigned long integer |

We can use the same method of calculating field widths as we used for the `spl_stat` program. For example, the printing field for `f_bfree` member starts in column 27. including its label, `"Free: "`, which is 6 characters long. The next printing field starts in column 44. Therefore, the width of the format specifier for `f_bfree` should be 44 − 6 − 27 or 11. The format specifier, since its type is `unsigned long`, should be `"%-11lu"`.

The last non-trivial part of the program is printing a string representation of the filesystem type, since the `statfs` structure's filesystem type field is an integer encoding that has to be decoded. Here's where Unix filtering tools and the `vi` editor come in handy. The `statfs()` man page has a list of filesystem types that can be values of the `f_ftype` member:

```
ADFS_SUPER_MAGIC      0xadf5
AFFS_SUPER_MAGIC      0xadff
AFS_SUPER_MAGIC       0x5346414f
```

```
    ANON_INODE_FS_MAGIC   0x09041934 /* Anonymous inode FS (for
                                        pseudofiles that have no name;
                                        e.g., epoll, signalfd, bpf) */
--snip--
```

We can copy that list to a file and open it in the `vi` editor. In the editor, with just a few global substitutions, including deleting all comments, deleting all occurrences of `_SUPER_MAGIC` deleting all occurrences of `_MAGIC`, and replacing the remaining underscores by hyphens, the file will look like:

```
ADFS      0xadf5
AFFS      0xadff
AFS       0x5346414f
ANON-INODE-FS   0x09041934
--snip--
```

We can then convert all uppercase to lowercase using the `vi` command `1,$s/./\L&/g`:

```
adfs      0xadf5
affs      0xadff
afs       0x5346414f
anon-inode-fs   0x09041934
--snip--
```

and rearrange each line using the command

```
:1,$s/^\([^ ]*\)  *\([0-9a-z]*\)[ ]*$/case \2: return "\1";/
```

so that they look like:

```
case 0xadf5: return "adfs";
case 0xadff: return "affs";
case 0x5346414f: return "afs";
case 0x09041934 : return "anon-inode-fs";
--snip--
```

These lines become the cases of a `switch` statement inside a function with prototype `char* fstype2name( struct statfs statfs_buf )` that returns the string associated with the number from the `f_ftype` member of the structure argument:

```
char* fstype2name( struct statfs statfs_buf )
{
    switch (statfs_buf.f_type) {
        case 0xadf5: return "adfs";
        case 0xadff: return "affs";
        case 0x5346414f: return "afs";
        case 0x09041934 : return "anon-inode-fs";
--snip--
        default: return "unknown type";
```

```
    return
}
```

To save space here, the complete function is only displayed in the in the book's source code distribution.

The final step before compiling and building the program is to make sure that our program will print the filesystem ID correctly. The man page for `statfs` raised a red flag about this when, in the `NOTES` section, it mentioned that in Linux and some other Unix systems, the type `fsid_t` of the `f_fsid` member of the `statfs` structure is defined as `struct { int val[2]; }`, in other words, as a pair of integers, rather than a long integer. To check this, we write a small program that does nothing except printing the filesystem ID contained in the `f_fsid` member of the `statvfs` structure, and compare its output to that of the `stat -f` command. The program, named *fsidtest.c*, follows.

*fsidtest.c*

```
#define _GNU_SOURCE
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/statvfs.h>

int main(int argc, char **argv)
{
    struct statvfs statvfs_buffer;
    if ( statvfs(argv[1],  &statvfs_buffer) == 0 )
        printf("%lx\n", statvfs_buffer.f_fsid);
    return 0;
}
```

Sure enough, when we build and run the executable, the filesystem ID that it outputs is different from the one output by `stat -f.` command:

```
$ fsidtest /var/log
7798a2a57e58ac74
$ stat -c"%i" -f /var/log
7e58ac747798a2a5
```

The order of the upper four bytes and lower four bytes is the reverse of what `stat -f` displays . The `f_fsid` member is in fact a sequence of two 32-bit numbers, but the high-order four bytes should be printed after the low-order four bytes. This is easily handled in the `print_statvfs()` function by masking out the upper bytes first to get the low-order four bytes, storing them into a variable, and then right-shifting the upper four bytes into a second variable, and printing them in reverse order, as follows:

```
unsigned int low = statvfs_buf.f_fsid & 0xFFFFFFFF;
unsigned int high = ( statvfs_buf.f_fsid  >> 32)& 0xFFFFFFFF;
```

```
printf("    ID: %08x%08x", low, high);
```

The complete program, with the preceding changes made in `print_statvfs()`, and the body of the `fstype2name()` function omitted, appears in Listing 7-2.

*spl_statfs.c*

```
#define _GNU_SOURCE
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <sys/vfs.h>
#include <sys/statvfs.h>
#include "../include/common_hdrs.h"
#include "../include/common_defs.h"

/* Given a statfs structure, this returns the human-readable filename stored
   in its f_ftype member.  */
char* fstype2name( struct statfs statfs_buf )
{
    switch (statfs_buf.f_ftype ) {
--snip--
    }
}

void print_stat(struct statvfs statvfs_buf, char* fstype)
{
    unsigned int low = statvfs_buf.f_fsid & 0xFFFFFFFF;
    unsigned int high = ( statvfs_buf.f_fsid  >> 32)& 0xFFFFFFFF;
    printf("    ID: %08x%08x", low, high);
    printf(" Namelen: %-8lu", statvfs_buf.f_namemax);
    printf("Type: %s\n", fstype);
    printf("Block size: %-11lu", (unsigned long) statvfs_buf.f_bsize);
    printf("Fundamental block size: %lu\n", statvfs_buf.f_frsize);
    printf("Blocks: Total: %-10lu", statvfs_buf.f_blocks);
    printf("Free: %-11lu", statvfs_buf.f_bfree);
    printf(" Available: %lu\n",statvfs_buf.f_bavail);
    printf("Inodes: Total: %-10lu", statvfs_buf.f_files);
    printf("Free: %lu\n", statvfs_buf.f_ffree);
}

int main(int argc, char **argv)
{
    struct statvfs statvfs_buffer; /* statvfs structure filled by statvfs()*/
    struct statfs  statfs_buffer;  /* statfs structure filled by statfs()  */
    char           mssge[128];     /* string to store error messages       */
```

```
    /* If no file arguments, print a usage message. */
    if (argc < 2) {
        sprintf(mssge,"usage: %s file \n", basename(argv[0]));
        usage_error(mssge);
    }
    printf("  File: \"%s\"\n", argv[1]);
    errno = 0;
    if ( statvfs(argv[1],  &statvfs_buffer) < 0 ) {
         sprintf(mssge, "Could not statvfs file %s\n", argv[1]);
         fatal_error(errno, mssge);
    }
    errno = 0;
    if ( statfs(argv[1],  &statfs_buffer) < 0 ) {
         sprintf(mssge,"Could not statfs file %s\n", argv[1]);
         fatal_error(errno, mssge);
    }
    print_stat(statvfs_buffer, fstype2name(statfs_buffer));
    return 0;
}
```

*Listing 7-2: A program to output a filesystem's metadata, given any file contained in it*

The `main()` function checks that there is a file argument and prints its name if it finds one. It then calls `statvfs()` to get the `statvfs` structure, and calls `statfs()` to get the `statfs` structure. Its last step is to call `print_stat()`, passing the `statvfs` structure and the filesystem type string returned by the call to `fstype2name(statfs_buffer)`.

### Testing spl_statfs

We're ready to test the program and compare its output to that of the `stat -f` command. First we'll run them both on the */dev* directory:

```
$ stat -f  /dev
  File: "/dev"
    ID: 986fcbcacd116e7e Namelen: 255     Type: tmpfs
Block size: 4096       Fundamental block size: 4096
Blocks: Total: 2016429    Free: 2016429    Available: 2016429
Inodes: Total: 2016429    Free: 2015803
$ spl_statfs /dev
  File: "/dev"
    ID: 986fcbcacd116e7e Namelen: 255     Type: tmpfs
Block size: 4096       Fundamental block size: 4096
Blocks: Total: 2016429   Free: 2016429     Available: 2016429
Inodes: Total: 2016429   Free: 2015803
```

The outputs are identical. They both identify the filesystem as a `tempfs` type, and all numeric values match. Next, we'll try them on the filesystem of a USB drive inserted into a USB port:

```
$ stat -f /media/guest/USB_stick/
  File: "/media/guest/USB_stick/"
    ID: 84000000000 Namelen: 1530    Type: msdos
Block size: 4096       Fundamental block size: 4096
Blocks: Total: 471778     Free: 347737     Available: 347737
Inodes: Total: 0          Free: 0
$ spl_statfs /media/guest/USB_stick/
  File: "/media/guest/USB_stick/"
    ID: 840     0        Namelen: 1530    Type: msdos
Block size: 4096       Fundamental block size: 4096
Blocks: Total: 471778    Free: 347737      Available: 347737
Inodes: Total: 0         Free: 0
```

Again the output it identical and our program correctly identified the filesystem type and the filesystem ID. Last, we show the output that both programs produce when the filesystem is on a CDROM disk:

```
$ stat -f  /media/guest/music-files/
  File: "/media/stewart/music-files/"
    ID: b0000000000 Namelen: 255     Type: isofs
Block size: 2048       Fundamental block size: 2048
Blocks: Total: 2048681    Free: 0          Available: 0
Inodes: Total: 0          Free: 0
$ spl_statfs  /media/stewart/music-files/
  File: "/media/guest/music-files/"
    ID: b0000000000 Namelen: 255     Type: isofs
Block size: 2048       Fundamental block size: 2048
Blocks: Total: 2048681    Free: 0          Available: 0
Inodes: Total: 0          Free: 0
```

These few sample runs do not constitute thorough testing of the program. Before we can make this code available for use, we should test it on a larger set of filesystems.

Completing this exercise has enabled us to solve a few other relatively easy problems. For example, we can design functions to display how many free blocks are available to users, or approximately what fraction of the blocks are in use. The `df` command does this, so we're now able to write out own version of `df` as well.

## Summary

In a very real sense, files are the core of a Unix operating system. Ordinary files can contain data and programs, directory files organize sets of files, and device special files enable programs to interact with devices in the same way as they do with regular files. Files have metadata as well, which are the statistics and properties associated with them, such as their size, permissions, type, and so on. Filesystems are an integrated collection of data structures written onto a storage device, together with the software that maintains

those data structures, designed to organize files and provide a programming interface for accessing them. A typical disk-based filesystem is decomposed into block groups, each of which contains a superblock, an inode table, and bitmaps for locating the inodes and data blocks.

Because Unix systems enable multiple, distinct filesystems to be integrating into a single, tree-like directory hierarchy through a procedure called mounting, at any given time, more than one type of filesystem can be a part of this hierarchy. These different filesystems are implemented in different ways and have different interfaces for their services. Many Unix kernels, and in particular, Linux kernels, contain an abstraction layer named the virtual file system (VFS). The VFS creates a common interface for system calls needing access to files and the filesystem, regardless of their underlying type.

The Linux kernel provides a few system calls for obtaining the metadata associated to files, including `stat()`, `lstat()`, and `statx()`. It also provides a separate set of calls for accessing filesystem metadata, including `statfs()`, and the C library provides the POSIX-conforming `lstatvfs()`. This chapter shows how these calls can be employed to implement a few commands for users to enter to obtain information about files and mounted filesystems on their host machine.

## Exercises

1. If the blocks in the filesystem are each 4096 bytes, and disk addresses are four bytes each, how many indirect blocks are needed to access the blocks of:
   (a) a file of size 5120KB?
   (b) a file of size 2048GB?
2. If a file is of size 1TB, how many disk addresses would be accessed to get the starting address of the very last block of the file, once the inode has been read into memory, assuming block size is 4KB?
3. If the blocks in a block group are each $2^B$ bytes, and the data block bitmap must fit into a single block, what is the maximum number of bytes of storage possible in this block group?
4. Write a function `newer()` with prototype

```
$ int newer(const char* pathname1, const char* pathname2);
```

that returns 1 if the file `pathname1` was last modified no earlier than the file `pathname2`, otherwise it returns 0. If their last modifications happened at the exact same nanosecond of time, it returns 1. If it encounters errors that prevent it from returning, such as pathnames that do not exist, it returns -1.

   (a) Write a main program with two expected command-line arguments that calls this function and outputs the name of the newer file, or exits with an error message.

(b) Modify this program to accept command-line options `-a`, `-b`, `-c`, and `-m` that constrain the program to use access time, birth time, status change time, and modification time respectively instead of the default, modification time.

5. Write a function named `samefile()` with prototype

```
$ int samefile(const char* pathname1, const char* pathname2);
```

that returns 1 if `pathname1` and `pathname2` are links to the same file, and 0 if they are not. If it encounters errors that prevent it from returning, such as pathnames that do not exist, it returns -1. Write a main program with two expected command-line arguments that calls this function and outputs a 1 or a 0 or exits with an error message.

6. Write a function named `sameowner()` with prototype

```
$ int sameowner(const char* pathname1, const char* pathname2);
```

that returns 1 if `pathname1` and `pathname2` are owned by the same user, and 0 if they are not. If it encounters errors that prevent it from returning, such as pathnames that do not exist, it returns -1. Write a main program with two expected command-line arguments that calls this function and outputs a 1 or a 0 or exits with an error message.

7. Write the complete program for version 2 of the `spl_stat` program, outlined in "Designing an Enhanced spl_stat Command."

8. Read the man page for `df` and run it to see what it displays without any options. Design a version of this command that displays the same data for a filesystem when given its device file pathname as an argument, for example, `df /dev/sda1`.

# 8

# THE DIRECTORY HIERARCHY

So far we've concentrated on the programming interface related to regular files. We learned how to perform basic and more advanced I/O in Chapters 5 and 6, and how to retrieve file and filesystem attributes in Chapter 7. Although we've explored the command-level interface for working with directories, such as commands for navigation, creation, deletion, and so on, we've yet to explore the programming interface related to directories and the directory hierarchy. For example, how can a program list all of the entries in a given directory, retrieve a single directory entry, change its current working directory, or get the absolute pathname of the current working directory? Still further, how can a program traverse selected parts of the directory hierarchy, such as by a depth first or breadth first search? In this chapter we explore the API related to directories and the directory hierarchy with the goal

of being able to write programs to perform these types of tasks. We'll also provide an overview of filesystem mounting, because as you'll see, how filesystems are mounted plays a part in how programs can perform various tasks related to directories.

## Directory Structure

Before we start to explore the API for interacting with directories, let's review what we know about them so far. First, as far as the kernel is concerned, directories are like regular files except that they have more restrictive form:

- They have a precisely-defined structure. A directory consists of a set of (*inode number, filename*) pairs called links. The inode number is an index into the inode table in the filesystem in which the directory resides, and the filename is the name in that directory for the referenced file. We often use the term *link* interchangeably with *filename*.
- They are never empty, because every directory has two unique entries, "`.`", called *dot*, and "`..`", called *dot-dot*, which refer respectively to the directory itself and to the parent directory. The exception to this rule is that in the root directory, whose name is *"/"*, "`.`" and "`..`" refer to the same inode, or put another way, *"/"* and *"/.."* are the same directory.
- Directories can only be created and modified by very specific system calls, unlike regular files, which can be created by `open()` and `creat()`.

Some commands that read from files can also read from directories, but the results are usually unpredictable. Some commands that do this on one system may not do this on another. For example, on some systems, the `cat` and `od` commands may display the contents of a directory as a stream of bytes, but because a directory is not a text file, the output of `cat` will look garbled. The output of `od` may look normal. However, other implementations of these commands will output an error message if their argument is a directory, such as

```
$ cat mydir
cat: mydir: Is a directory
```

The `open()` system call will open a directory, provided that it is opened in read-only mode, and the `close()` call will close it. Although we can open a directory with `open()`, on most systems the `read()` system call will fail to read it. The reason that `open()` is allowed to open a directory is to access the metadata in its inode, not to read its contents. We'll see later than being able to open a directory and create an open file description for it is useful in a few situations, such as when we want to save our current working directory and return to it later.

On some Unix implementations, the `read()` system call may succeed when given a directory argument. The following program can be run to check whether it is successful when given a directory.

*testdircalls.c*

```
#include "common_hdrs.h"

int main(int argc, char* argv[])
{
    int fd;
    char buf[2];

    if ( argc < 2 )
        printf("usage: %s <directory-path>\n", argv[0]);
    else {
        errno = 0;
        fd = open(argv[1], O_RDONLY);
        if ( -1 == fd )
            fatal_error(errno, "open");
        errno = 0;
        if ( -1 == read(fd, buf, 1) )
            perror("read() was not successful");
        else
            printf("read() was successful.\n");
        errno = 0;
        if ( -1 == close(fd) )
            fatal_error(errno, "close");
    }
    return 0;
}
```

If we compile and build it with the command

```
$ gcc -L../lib  -o testdircalls testdircalls.c -lspl
```

and run it with a directory argument we'll see whether reading failed:

```
$ testdircalls .
read() was not successful: Is a directory
```

The calls to `open()` and `close()` succeeded, but not the call to `read()`. Our next goal is to find those system calls and library functions that are intended to work with directories.

## Processing Directories

Some of the simplest commands that we routinely use involve directories. The most obvious example is the `ls` command, which lists the files in each directory entered on its command line. In fact, the arguments to `ls` can also

be non-directory files, such as regular files and special files, but without any options all it does is print out their names:

```
$ ls myfile yourfile
myfile  yourfile
```

Its more interesting usage is when the argument is a directory. In that case it displays the links contained in that directory, and with various command options, it will display selected metadata as well.

Because `ls` outputs different information depending on whether it's given a directory or a non-directory file, its main program must check the file type before it does much work. We already know how to do this, namely by calling `stat()` or `statx()` and checking the file type in the file type bits of the `st_mode` member of the returned structure, as in this code snippet:

```
if ( -1 != stat(argv[1],&statbuffer)) {
    if ( S_ISDIR(statbuffer.st_mode)) /* Display links in argv[1]. */
        list_dir_contents(argv[1]);
    else
        printf("%s", argv[1]);
}
```

in which `list_dir_contents()` is some function that would list the directory's contents. We'd like to discover how, at the programming level, we can retrieve the contents of a directory. Following our usual procedure, we'll start with a man page search to find system calls or library functions that might help us. It's reasonable to search for the two terms *read* and *directory* exactly and occurring simultaneously:

```
$ apropos -s2,3 -e -a read directory
readdir (2)          - read directory entry
readdir (3)          - read a directory
readdir (3posix)     - read a directory
readdir_r (3)        - read a directory
```

The first match, `readdir()` in Section 2, must be a system call. We'll take a look at that man page first, after which we'll read the man page for what is likely to be a library function, `readdir()` in Section 3.

```
$ man 2 readdir
READDIR(2)                Linux Programmer's Manual               READDIR(2)

NAME
       readdir - read directory entry

SYNOPSIS
       int readdir(unsigned int fd, struct old_linux_dirent *dirp,
                   unsigned int count);

       Note: There is no glibc wrapper for this system call; see NOTES.
```

```
DESCRIPTION
       This  is  not the function you are interested in.  Look at readdir(3)
       for the POSIX conforming C library interface.  This  page  documents
       the  bare  kernel  system call interface, which is superseded by get
       dents(2).
--snip--
```

The DESCRIPTION tells us that we shouldn't be using this function but should instead look at the library function in Section 3. It does however mention a system call that has replaced this one, the getdents() call. If we read its man page:

```
$ man getdents
GETDENTS(2)              Linux Programmer's Manual             GETDENTS(2)

NAME
       getdents, getdents64 - get directory entries

--snip--

       Note: There is no glibc wrapper for getdents(); see NOTES.

DESCRIPTION
       These  are  not  the interfaces you are interested in.  Look at read
       dir(3) for the POSIX-conforming C library interface.  This page docu
       ments the bare kernel system call interfaces.
--snip--
```

we see the same warning. At this point, it's pretty clear that this is like posted property and we should stay off of it. For this project, we're going to work with the POSIX-conforming library functions instead of system calls. One clear advantage will be that our code will be portable.

### *The readdir() Library Function*

We'll begin by reading the Section 3 man page for readdir(), which is the Linux page for it, after which we'll take a look at the POSIX man page for it in Section 3posix, since it might contain information not present in the Linux page.

```
READDIR(3)               Linux Programmer's Manual              READDIR(3)

NAME
       readdir - read a directory

SYNOPSIS
       #include <dirent.h>
```

```
       struct dirent *readdir(DIR *dirp);

DESCRIPTION
       The readdir() function returns a pointer to a dirent structure repre
       senting the next directory entry in the directory stream  pointed  to
       by dirp.  It returns NULL on reaching the end of the directory stream
       or if an error occurred.

       In the glibc implementation, the dirent structure is defined as  fol
       lows:

           struct dirent {
               ino_t          d_ino;       /* Inode number */
               off_t          d_off;       /* Not an offset; see below */
               unsigned short d_reclen;    /* Length of this record */
               unsigned char  d_type;      /* Type of file; not supported
                                              by all filesystem types */
               char           d_name[256]; /* Null-terminated filename */
           };

       The  only fields in the dirent structure that are mandated by POSIX.1
       are d_name and d_ino.  The other fields are unstandardized,  and  not
       present on all systems; see NOTES below for some further details.
--snip--
```

Given a pointer, `dirp`, to a *directory stream* this function returns a structure representing a directory entry. We don't know exactly what a directory stream is yet, nor how we get one, but based on the man page, we know that the `DIR*` type is a pointer to one. We'll address these issues later. The man page states a few important facts about calling the function:

- Successive calls to `readdir()` with the same directory stream pointer return successive entries in that directory stream, and when all entries have been accessed, it returns a `NULL` pointer.
- If an error occurs in a call, it returns a `NULL` pointer. To distinguish between an error and the end of the directory, we need to set `errno` to zero before the call and check it after.
- The entry returned by `readdir()` may be overwritten by subsequent calls to `readdir()` for the same directory stream, because it might be a statically-allocated variable in the library. Therefore, before calling it again, if we want to save the returned data, we have to copy it into a local variable.

Now let's read about the `dirent` structure returned by the call, to make sure we understand how to use it.

### The dirent Structure

Despite this data structure's having several fields, only the `d_ino` and `d_name` fields are guaranteed by POSIX to be part of it. The first field contains the inode number and the second stores the `NULL`-terminated filename.

#### Filenames and the dirent Structure

The `NOTES` mention that POSIX.1-2017 doesn't specify a length for this filename even though the Linux man page declares it as having 256 characters including the `NULL` byte. Programs should not depend on its having any particular length. POSIX only ensures that its length is at most `NAME_MAX` characters. Therefore, for the program to be portable, any variables that store the filename should be declared to be at least `NAME_MAX+1` bytes. For example, if a local variable named `filename` has to store a name from the `d_name` member, it should be declared as follows:

```
char filename[NAME_MAX+1];
```

to allow for the terminating `NULL` byte.

**ABOUT `NAME_MAX` AND SYSTEM LIMITS**

The `NAME_MAX` symbol is an example of *system limit* in Unix. It specifies the maximum number of bytes in a filename. It's a particular kind of limit called a *pathname variable value* because it defines a limit related to pathnames. Its actual value may vary at run-time. On some Unix implementations, its value might be fixed, but on others it might vary from one pathname to another because the underlying filesystem supports different filename lengths. A program can get its actual value at runtime by calling `pathconf(`*`dirpath`*`, _PC_NAME_MAX)` where *`dirpath`* is the pathname of any directory on the particular filesystem. For example,

```
int maxname = pathconf("/home/snw", _PC_NAME_MAX);
```

returns the maximum filename length allowed on the filesystem containing my home directory. System limits are covered in more detail in Appendix B.

It's natural to wonder whether a program calling `readdir()` will need to include some other header file to use this `NAME_MAX` value, or whether including *dirent.h* is sufficient. In general, man pages usually list every header file needed to use the functions they describe. In this case, the `dirent.h` header file does expose the definition of `NAME_MAX`, albeit through several levels of nested `#include` directives. We can verify this by compiling the following one-line program:

```
#include <dirent.h>
void main()
{  int n = NAME_MAX; }
```

If the symbol weren't made available by including *dirent.h*, we'd get an error message from the compiler, but it compiles without errors. If we put a `printf()` instruction in this program, we'd most likely see that its value is 255, but this is implementation-dependent. POSIX.1-2017 only requires it to be at least 14 bytes.

The `d_name` member of the structure is the one member we're most interested in, but we'll also examine the `d_type` member.

#### File Types and the dirent Structure

The `dirent` structure also has a `d_type` field, which contains the type of the entry, such as whether it's a regular file, a directory, a symbolic link, and so on. Because this field isn't required by POSIX.1-2017, it may not be a member of the structure on some systems. This implies that a program that uses this field will not be completely portable. On systems with *glibc*, such as Linux, a program can determine whether or not the field is actually in the structure with the macro `_DIRENT_HAVE_D_TYPE`; it's only defined if the structure has the `d_type` member. The code referring to `d_type` should then be protected by a conditional macro test:

```
#ifdef _DIRENT_HAVE_D_TYPE
    we have d_type member so we can get type information with it
#else
    we don't have d_type; we need to call stat() to get type
#endif
```

Even if `d_type` is a member of the structure, it may not have type information because not all filesystems provide it; the Ext2/3/4 and BSD filesystems do provide it. The reason it's present on some systems is that, if a program needs to know the type of the directory entry, it's faster to get it by using this field than by making a call to `stat()` or `lstat()`, which are more time-consuming.

The *glibc* library exposes a set of macro constants with names such as `DT_LNK`, `DT_DIR`, `DT_REG`, and so on, for the value in `d_type`. To make them available on *glibc* 2.20 or later, the feature test macro `_DEFAULT_SOURCE` should be defined before any include directives; on earlier versions of *glibc*, the macro `_BSD_SOURCE` must be defined. A program can define both macros safely. The `readdir()` man page has the complete list of these macro constants. If our programs don't need the type of the directory entry, we don't need to bother with the added complexity of using the `d_type` member in a portable way.

### Directory Streams

Nowhere in the man page for `readdir()` is there any information about what a `DIR` is, but the `NOTES` section tells us how to get one and provides a clue: "A directory stream is opened using `opendir(3)`." From this, we understand that a `DIR` object is a directory stream, and we learned how to obtain one. This `opendir()` function is also listed in the `SEE ALSO` section, along with several other functions that are likely to be part of the directory API.

Based on our experience in Chapter 6 with the `passwd` database API, it's likely that `opendir()` may serve a purpose similar to that of `setpwent()`, but with respect to directories, by initializing an iterator that retrieves successive directory entries until all have been accessed. This is a good time to look at the POSIX page for `readdir()`, because it might explain more about directory streams. In fact, it has the following description:

> The type DIR, which is defined in the <dirent.h> header, represents a directory stream, which is an ordered sequence of all the directory entries in a particular directory. Directory entries represent files; files may be removed from a directory or added to a directory asynchronously to the operation of readdir().
> The readdir() function shall return a pointer to a structure representing the directory entry at the current position in the directory stream specified by the argument dirp, and position the directory stream at the next entry. It shall return a null pointer upon reaching the end of the directory stream.

Although we don't need to know the structure of a `DIR` to use `readdir()`, out of curiosity, we might like to see it. In fact, `DIR` is defined by a `typedef` in *dirent.h*:

```
typedef struct __dirstream DIR;
```

but there is no definition of `struct __dirstream` in any userspace header files in the system. This is because POSIX allows `__dirstream` to be an incomplete type. An *incomplete type* is a type that describes an object but lacks the information needed to determine its size. Each implementation of Unix must define it, and is free to define it as it chooses, but it need not expose that implementation in any userspace header files. The *dirent.h* header file declares `struct __dirstream` and makes `DIR` equivalent to it, but does not define its members. This gives programmers the ability to declare objects of type `DIR*`, but not the ability to access the members of a `DIR` object.

**NOTE** *If you write a program that references a `__dirstream` object, the compiler will report an error that its size is unknown. If you instead declare a `__dirstream*` variable, the program will compile, because the pointer's size is known. You will not be able to dereference this pointer and use what it points to because your program does not have access to its implementation, but it's implemented in the libraries that use it. This is a form of information hiding in C.*

If you download the source code for a recent version of *glibc*, such as 2.37 or later, you'll find the definition of this structure in the file *sysdeps/unix/sysv/linux/dirstream.h*.

It's time to look at the `opendir()` function, whose purpose is to return a directory stream.

### The opendir() Library Function

The man page for `opendir()` begins as follows:

```
OPENDIR(3)                Linux Programmer's Manual                OPENDIR(3)

NAME
       opendir, fdopendir - open a directory

SYNOPSIS
       #include <sys/types.h>
       #include <dirent.h>

       DIR *opendir(const char *name);
       DIR *fdopendir(int fd);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       fdopendir():
           Since glibc 2.10:
               _POSIX_C_SOURCE >= 200809L
           Before glibc 2.10:
               _GNU_SOURCE

DESCRIPTION
       The  opendir() function opens a directory stream corresponding to the
       directory name, and returns a pointer to the directory  stream.   The
       stream is positioned at the first entry in the directory.

--snip--
```

This function is much simpler than `readdir()`. We give it the pathname of a directory and it returns a pointer to the beginning of a directory stream, so that the first call to `readdir()` on that stream returns the first entry in the directory. Our program will need to declare a variable of type `DIR*` to receive the returned address. If the function fails, it returns a `NULL` pointer. Note that this function requires the *sys/types.h* header file. There's nothing significant in the `NOTES` section for us to be concerned about at this point. A code snippet illustrating how to call the function follows:

```
DIR           *dirp;
struct dirent *dir_entry;

dirp = opendir("/home/stewart");
if ( NULL != dirp )
    /* call readdir() */
    dir_entry = readdir(dirp);
--snip--
```

Whenever we open something, we ought to close it. That's the general rule in programming. If there's an `opendir()` function, there's likely to be a

closedir() function that our programs should call to close a directory stream and free up its resources.

### The closedir() Library Function

A man page search shows that there is a closedir() function. The closedir() function closes the given open directory stream. The relevant part of its man page is

```
SYNOPSIS
       #include <sys/types.h>
       #include <dirent.h>

       int closedir(DIR *dirp);

DESCRIPTION
       The  closedir()  function closes the directory stream associated with
       dirp.  A successful call to closedir()  also  closes  the  underlying
       file  descriptor associated with dirp.  The directory stream descrip
       tor dirp is not available after this call.

RETURN VALUE
       The closedir() function returns 0 on success.  On error,  -1  is  re
       turned, and errno is set appropriately.
```

Notice that this function also needs the *sys/types.h* header file, and that it returns -1 and sets errno on failure, like a system call. The only possible error is passing it an invalid directory stream pointer, such as one that's already been closed.

## A Simple ls Program

To demonstrate the use of the few functions we've just discovered, we'll implement a simplified version of the ls command, which, for arguments that are directories, lists the filenames in them, and for arguments that are non-directory files, just lists their names. We'll make it act like the real ls in that, when it isn't given any arguments, it lists the files in the working directory. This program won't accept any command-line options.

We'll put the logic of listing all files in a single directory into a function named listdir(), which is given a directory stream pointer, dirp, and an integer, flags, that encodes a set of flags. The listdir() function repeatedly calls readdir(dirp) to get the next entry from the dirp stream and print its filename member, until readdir(dirp) returns NULL. If readdir() reports an error when trying to read an entry, listdir() prints a message and skips the file. The flags parameter isn't used in this version of listdir(); it's there to make the function extensible.

**EXTENSIBLE DESIGN**

*Extensibility* is a measure of the ease with which the functionality of a software design or artifact can be extended. Designing with extensibility in mind allows for unanticipated future improvements and enhancements.

The `main()` function's job is to process the command line. For every pathname argument on the command line, `main()` will attempt to get a directory stream for it by calling `opendir()`. If the call fails and sets `errno` to `ENOTDIR` it means that the argument is not a directory and therefore, instead of calling `listdir()`, it will print the argument. If `opendir()` fails for any other reason, it skips the argument without exiting. The complete program is shown in Listing 8-1. I've named the source file *spl_ls1.c* anticipated enhanced versions to follow.

*spl_ls1.c*

```
#include "common_hdrs.h"
#include <dirent.h>

/* listdir(dirp,flag) prints the filenames contained in the directory stream
   dirp, one per line, including . and .., in  the order the stream delivers
   them. */
void listdir( DIR *dirp, int flags )
{
    struct dirent   *direntp;      /* Pointer to directory entry structure  */
    BOOL            done = FALSE;  /* Flag to control loop execution        */

    while ( !done ) {
        errno = 0;
        direntp = readdir( dirp );                        /* Get next entry. */
        if ( direntp == NULL && errno != 0 )
            /* Not the end of the stream, but an error from readdir()       */
            perror("readdir");
        else if ( direntp == NULL ) /* errno == 0, nothing left in stream   */
           done = TRUE;
        else
           printf("    %s\n", direntp->d_name );          /* Print it.      */
    }
    printf("\n");
}

int main(int argc, char *argv[])
{
    DIR   *dirp;
    int    i;
    int    ls_flags = 0;
```

```
    if ( 1 == argc ) {      /* No arguments; use current working directory. */
        errno = 0;
        if ( ( dirp = opendir(".") ) == NULL )
            fatal_error(errno, "opendir");           /* Could not open cwd. */
        else
           listdir( dirp, ls_flags );
    }
    else {         /* For each command-line argument, call opendir() on it. */
        for ( i = 1; i < argc; i++) {
            errno = 0;
            if ( ( dirp = opendir(argv[i]) ) == NULL ) {
                if ( errno == ENOTDIR )  /* It's not a directory. */
                    printf("%s\n", argv[i] );
                else  /* It's an error. */
                    error_mssge(errno, argv[i]);
            }
            else {  /* A successful open of a directory */
                printf("%s:\n", argv[i] );
                listdir( dirp, ls_flags);
                closedir(dirp);
            }
        }
    }
    return 0;
}
```

*Listing 8-1: A program that prints the contents of all directories in its argument list*

To demonstrate the program's behavior, I created a directory named *testing* containing a few files and directories, and removed read permission from one of the directories to see whether the program would handle this error well. The directory is depicted in Figure 8-1.

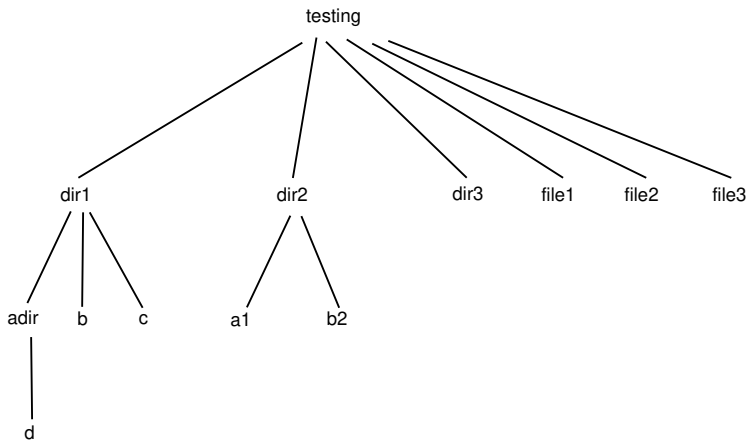



*Figure 8-1: The* testing *directory*

The following listing shows that *dir3* has no read permission:

```
$ ls -gG  testing/dir3    # The -gG supresses owner and group information.
total 12
drwxr-xr-x 3 4096 Sep 27 10:26 dir1/
drwxr-xr-x 2 4096 Sep 27 10:27 dir2/
d-wx--x--x 2 4096 Sep 27 10:36 dir3/
--snip--
```

I compile and build this program using

```
$ gcc -Wall -g spl_ls1.c -L../lib -lspl -o spl_ls1
```

Following are two runs of it, one on *testing*, and the other on some directories and files within it.

```
$ spl_ls1 testing
testing:
    dir1
    file3
    file1
    dir3
    file2
    dir2
    .
    ..
$ cd testing; spl_ls1 dir1 file1 dir3 file2
dir1:
    b
    c
    .
    adir
    ..

file1
dir3: Permission denied
file2
```

Notice that it correctly listed the contents for the directories for which it had permission, but not for *dir3*, for which it correctly reported the permission error. The order in which it prints a directory's contents appears to be random. It isn't sorted in any obvious way. Also, unlike the real `ls` command, our program lists the dot ("`.`") and dot-dot ("`..`") entries.

This program is a good start. It was relatively easy to design and write, owing to the fact that the directory API provides functions that allow us to iterate over all directory entries. In addition, we didn't need to use any of the non-POSIX members of the `dirent` structure for this version of `ls`, which made the program simpler.

We could improve the program in a few different ways. First, we can easily suppress printing of the dot and dot-dot entries, as well as all entries that are supposed to be hidden because their names start with dot. Another relatively easy improvement would be to list more than one filename per line, which would require mostly just a bit of arithmetic and some output format planning. Other enhancements are more challenging. One would be to print the filenames according to some specified ordering, such as by their names or times of last modification, and so on. Another would be to filter the output so that we list only files that meet a supplied condition, such as those that are directories, or those whose names match a pattern. These last two enhancements can't be implemented easily with just the set of functions we know about so far. We need to do a bit of research to see what other tools are available in the directory API. Trying to implement some of these enhancements is a good exercise for learning more about this API.

## Other Functions in the Directory API

The `SEE ALSO` section of the `readdir()` man page lists several library functions that work with directories in one way or another. The ones whose names end in `dir` include `rewinddir()`, `seekdir()`, `telldir()`, and `scandir()`. The others whose names don't match that pattern are `ftw()`, `dirfd()`, and `offsetof()`.

The last two functions, `dirfd()` and `offsetof()`, are not as relevant to our immediate objectives as are the others. The `dirfd()` function returns a file descriptor for the directory opened by a call to `opendir()`, and the `offsetof()` function is not specific to directories; it allows a program to obtain the offset within a C `struct` of one of its members, measured in bytes. It is needed occasionally because the sizes of some members of a structure can vary at runtime, such as the `d_name` member of the `dirent` structure. We'll examine the other functions listed there, starting with the ones whose names end in `dir`.

The first, `rewinddir()`, simply resets the directory stream so that reading begins at its first entry again. Its name is suggestive of this. Its prototype is

```
#include <sys/types.h>
#include <dirent.h>
void rewinddir(DIR *dirp);
```

We need the `rewinddir()` function for those occasions when a program has to make another pass across all of the entries in a directory stream. One reason for a second pass is that, in the first pass, it didn't find the information it needed without calling `stat()` on the entries. In a second pass, it could call `stat()` on all of the files. Another use case is when a first pass is needed to count the number of files satisfying some condition, and if the count is above a threshold value, a second pass is made to process those files satisfying the condition.

### The telldir and seekdir Library Functions

The two functions `seekdir()`, and `telldir()` are inter-related, in that neither is useful in a program without the other. Their prototypes are

```
#include <dirent.h>
long telldir(DIR *dirp);
void seekdir(DIR *dirp, long loc);
```

The first, `telldir()`, returns a `long` integer that can be used to return to the entry that would be read by the next call to `readdir()` in its directory stream argument, `dirp`. In essence, it's saving the current position of the stream iterator. The second, `seekdir()`, given the directory stream `dirp` and a `long` integer returned by `telldir()`, positions the stream's internal iterator so that the next call to `readdir()` reads the entry at that position in the stream. Combined, these two functions provide a way to save a position in the stream and return to it later.

**NOTE** *The `telldir()` man page warns us not to assume that the long integer returned by `telldir()` is simply an offset relative to the start of the directory. Modern filesystems can represent directories using hash tables or search trees to improve performance. For these filesystems, the value returned by `telldir()` and used by `readdir()` is what the page calls a* cookie*, meaning an integer value from which the actual address of the entry can be derived.*

One application of these functions is to process the entries in a given directory that satisfy a given condition before those that don't. For example, we could use them to print all entries that are directories preceding all non-directory entries. Because this is a good exercise in using the directory API, we'll write a second version of an `ls` command, named `spl_ls2`, that does exactly this.

Let's sketch out its algorithm. Each time the program reads an entry in the given directory stream, it checks whether or not it's a directory. If it's not a directory, it saves it in a list to print later, and if it is a directory, it prints it immediately. Since this requires knowing an entry's type, this exercise will also demonstrate how we can use the `_DIRENT_HAVE_D_TYPE` feature test macro to conditionally compile a program on systems that may or may not have the `d_type` member in the `dirent` structure. Therefore, our first task is to write a Boolean-valued function that determines whether or not an entry is a directory, which we'll name `isdir()`.

The following listing contains its implementation. The function checks whether or not its `struct dirent*` argument is a directory entry. If our host system's `dirent` structure contains a `d_type` member, it uses it, otherwise it calls `stat()` to get the type. Since this is a compile-time decision, the function uses the feature test macro to choose which code to include. Because we'll probably use this function in several other programs, we'll add it to our `libspl` library and create a header file named *dir_utils.h* that includes its prototype, so that we don't need to include its code in every program that uses it.

*isdir()*

```
/* Returns TRUE if *direntp represents a directory, and FALSE otherwise. */
BOOL isdir( const struct dirent *direntp)
{
#ifdef _DIRENT_HAVE_D_TYPE         /* We have the d_type member.     */
    if( direntp->d_type  == DT_DIR)
#else                              /* We don't have it - call stat() */
    struct stat    statbuf;
    stat(direntp->d_name, &statbuf);
    if ( statbuf.st_mode & S_IFDIR )
#endif
        return TRUE;
    else
        return FALSE;
}
```

*Listing 8-2: A function that checks whether a `dirent` structure is a directory entry*

This `isdir()` function makes the rest of a revised `listdir()` function simpler. Following is a rough outline of how the revised `listdir()` will process a single directory.

1. Given a directory stream argument `dirp`, which was opened successfully by the main program, it repeatedly performs the following steps.
   (a) It calls `telldir()` to save the current position, say in a variable named `pos`.
   (b) It reads the next entry: `direntp = readdir(dirp)`.
   (c) It calls `isdir(direntp)` to check whether this entry is a directory.
   (d) If it's a directory, it processes it, meaning it prints its name. If it isn't, it saves the position `pos` in a list of locations to process later.
2. When all entries have been read, it exits the loop, and, for each saved position `pos` in the list, seeks to that position using `seekdir()` and processes it, meaning in this case, prints it.

Notice that the program must call `telldir()` before it calls `reaaddir()`, because `telldir()` returns the position *it is about to read*, not the one just read.

We need to decide whether to use a linked list or an array to store the saved positions. A linked list has more overhead because of the possibly frequent calls to `malloc()` for each non-directory entry, and the extra code for linked list management. An array is faster but will need to be resized if it reaches capacity. Despite the linked-list's greater overhead, it leads to a simpler design.

We'll modify the previously written `listdir()` function so that, if its `flags` argument contains a flag to turn on *directories-first* processing, it will print directories before non-directories, and if not, it will print the entries in the order presented to it by the calls to `readdir()`.

The revised function will need the support of a few linked list functions, namely, one to append a node to the end, one to print the list, and one to erase it. To save space, only their prototypes are included here; their implementations are included in the book's source code distribution. The list definition and the function prototypes are shown in Listing 8-3.

```
/* Linked list node that stores an offset returned by telldir(). */
typedef struct listnode{
    long pos;                    /* The offset                    */
    struct listnode *next;       /* Pointer to nextnode in list   */
} poslist;                       /* Pointer to a list             */

/** save(p, &pos_listptr) saves  position p onto the end of the list
    pointed to by pos_listptr. Because the list head might be
    changed, its address is passed, not its value.                */
void save(long pos,  poslist **list);

/* printlist(dirp, pos_list) prints the filenames whose offsets were
   saved into pos_list.  */
void printlist(DIR* dirp, poslist *list)

/* eraselist(&list) erases the list pointed to by list.          */
void eraselist(poslist **list)
```

*Listing 8-3: Linked list utility functions for the revised `listdir()` function*

The preceding functions are called by the revised `listdir()`, which is displayed in Listing 8-4. In the listing, the parts of the function that have been modified are in bold.

*listdir()*

```
void listdir(DIR *dirp, int flags)
{
    struct dirent  *entry;
    long int       pos;
    poslist        *saved_positions = NULL;

    while ( 1 ) {
        pos = telldir(dirp);         /* Save current position.    */
        errno = 0;                   /* Try to read entry.        */
        if ( NULL == (entry = readdir(dirp) ) && errno != 0 )
            perror("readdir");     /* Error reading entry.        */
        else if ( entry == NULL )
            break;
        else {
            if ( (flags & LIST_DIRS_FIRST) && !isdir(entry) )  {
                save(pos, &saved_positions);
                continue;
            }
```

```
            printf("%s/\n", entry->d_name);
        }
    }
    if ( flags & LIST_DIRS_FIRST )
        printlist(dirp, saved_positions);
    eraselist(&saved_positions);
}
```

*Listing 8-4: A revised `listdir()` function that can list a directory's entries with all directory names preceding all non-directory names, with new code in bold*

The major changes to the revised function's code include the call to `telldir()` at the top of the loop, the inserted directory test, and the post-processing at the end of the loop to print the saved entries. I also modified the output by appending a `/` character to the end of directory names so that they can be identified.

The only changes to the main program are the declaration of the macro constant, `LIST_DIRS_FIRST = 1` and initialization of `ls_flags` to `LIST_DIRS_FIRST` instead of 0, so that the call `listdir(dirp, ls_flags)` turns on the new feature. I'll name this program *spl_ls2.c*. For brevity, I don't list the complete program here, but it's provided in the book's source code distribution. We can build the executable, naming it `spl_ls2` and run it on the same directory that we ran `lsdir`:

```
$ spl_ls2 testing
testing:
dir1/
dir3/
dir2/
./
../
file3
file1
file2
```

All directory names precede all non-directory names. This output includes the dot (`.`) and dot-dot (`..`) entries, which are directories. You should run this program on other directories to convince yourself that it works like this.

### The scandir() Library Function

The name `scandir()` suggests that this function can scan a directory, possibly to look for something. We'll begin by looking at the synopsis on its man page:

```
#include <dirent.h>
int scandir(const char *dirp, struct dirent* **namelist,
        int (*filter)(const struct dirent *),
        int (*compar)(const struct dirent **, const struct dirent **));
```

```
int alphasort(const struct dirent **a, const struct dirent **b);
int versionsort(const struct dirent **a, const struct dirent **b);
```

Because this function's prototype is more complex than any we've seen so far, we'll begin by going through the mechanics of calling it. After that we'll consider what it does and how we can use it.

### How We Call scandir()

Unlike other functions we've seen so far, `scandir()` has parameters that are functions. More accurately, `(*filter)` and `(*compar)` are both *pointers* to functions. A parameter that is a pointer to a function is called a *function pointer parameter*. The third parameter of `scandir()` is declared as

```
int (*filter)(const struct dirent *)
```

This declares `filter` to be a pointer to a function whose single argument is a pointer to a constant `dirent` structure and whose return type is `int`. A function such as the following matches it:

```
int skipdot(  const struct dirent *direntp)
{
    if (strcmp(direntp->d_name, ".") == 0
        || strcmp(direntp->d_name, "..") == 0)
        return 0;
    else
        return 1;
}
```

Similarly, the fourth parameter is declared as

```
int (*compar)(const struct dirent **, const struct dirent **)
```

which states that `compar` is a pointer to a function expecting two arguments, each of type `const struct dirent**`, that returns an `int`. The `alphasort()` function shown in the man page has the exact same prototype as the fourth parameter (`compar`) and could be passed as the fourth argument to `scandir()`.

The `scandir()` function also has a triply-indirect parameter, `namelist`. This parameter is the address of a dynamic array of pointers to `dirent` structures. It is referred to by three levels of indirection — the address is the first, the array name is the second, and the array entries themselves are pointers, hence the third. If a program declares `dp_array` to store the address of a pointer to a `dirent` structure as follows:

```
struct dirent*  *dp_array; /* Not struct dirent* dp_array[] */
```

then it would pass `dp_array`'s address, `&dp_array` as the second argument to `scandir()`. Putting this all together, we could call `scandir()` as follows:

```
int returnval = scandir("/home/snw/", &dp_array, skipdot, alphasort);
```

Although we now know how to call `scandir()`, we still don't know what it does. That comes next.

**FUNCTION POINTER PARAMETERS**

In C, when a function has a function pointer parameter, a calling program can pass to it a pointer to a function whose prototype matches that of the parameter. Let's consider an example. Suppose that we declare the function `f()` as follows:

```
double f( int (*funcp)( int, int ), int*  );
```

Then the first parameter of `f()` is `funcp`, a pointer to a function whose prototype is

```
 int function_name ( int, int );
```

where *`function_name`* is any valid function name. A calling program can pass the address of any function whose prototype is of this form as the first parameter of `f()`. Suppose that `g()` is defined to be the following function:

```
int g( int x, int y ) { return x*y; }
```

The prototype of `g()` matches that of `*funcp`. Therefore we can pass `g()` in a call to `f()` as follows:

```
double res = f(g, 12);}
```

Since the compiler replaces the name of a function by its address, it isn't necessary to call it like this, although it is also correct:

```
double res = f(&g, 12);}
```

In contrast, if the function `h()` has the following prototype:

```
int h( int*, int* );
```

then it is an error to pass it to `f()`, because its prototype doesn't match the parameter's exactly. See the function *functionptr_demo.c* in the book's source code distribution for a more thorough example that also uses a `typedef` to declare a function pointer type.

### What scandir Does

Let's assume that the `scandir()` function's first argument is the pathname of a directory, `dirp`. It opens that directory and makes a pass across every entry in the directory's stream. For each entry, it calls the `(*filter)` function on that entry. The `(*filter)` function returns an integer. For each entry for which that return value is nonzero, `scandir()` stores a pointer to its `dirent` structure in `namelist`, sorted using the comparison function `(*compar)`

on the pairs of entries. If the call passes `NULL` to the filter function parameter (`*filter`), all entries are stored in the array, and if it passes `NULL` to the comparison function parameter, no sorting takes place.

The comparison function is not limited to comparing the entries by their filenames — it can compare the two entries by examining any data in the entry, even if that involves calling `lstat()` on them. A comparison function must have two parameters that are constant pointers to `dirent` structures, and it must return an integer. The traditional return values of comparison functions are −1, 0, and 1, but it only needs to return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. For example, the following function could be used to compare two entries by the sizes of the files:

```
int cmpbysize(const struct dirent **a, const struct dirent **b)
{
    struct stat a_sb, b_sb;
    stat((*a)->d_name, &a_sb);
    stat((*b)->d_name, &b_sb);
    return ( a_sb.st_size - b_sb.st_size );
}
```

This example returns numbers that are not necessarily -1, 0, or 1. Also, it has no error-handling, which is omitted to save space. If we just want to sort the entries alphabetically in accordance with the locale's `LC_COLLATE` value, we can pass the `alphasort()` function to `scandir()`, because `alphasort()` calls `strcoll()` internally, which is locale-aware.

When `scandir()` has returned, the `namelist` array contains all entries that met the filter's conditions, sorted by the sorting criteria embodied in the comparison function, and the return value of the function is the number of entries in that array. The `scandir()` function allocates its own memory for the `namelist` array. A program calling `scandir()` must not declare its own storage for it, but it must free the memory allocated to the array when it no longer needs it.

The man page has a simple example that illustrates how to call the function and free the memory. We include it here, modified slightly to reduce space:

*scandir_manpage_example.c*

```
#define _DEFAULT_SOURCE
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   struct dirent **namelist;
   int n;
```

```
    if ( (n = scandir(".", &namelist, NULL, alphasort)) == -1)
        exit(EXIT_FAILURE);
    while (n--) {
        printf("%s\n", namelist[n]->d_name);
        free(namelist[n]);
    }
    free(namelist);
    exit(EXIT_SUCCESS);
}
```

This program prints the entries in the current working directory. The first name printed is the last one in the array, because it prints the directory entries in the reverse of the collating order.

We can use the `scandir()` function to write an improved version of *spl_ls2.c*, our program that lists directory contents with directories first. Not only will it print the directories first, but it will sort the entries alphabetically in the ordering of the current locale. It won't need to open the directory and read it using the `opendir()` and `readdir()` functions because `scandir()` bypasses that work. The first step is to write the comparison function, which is shown in Listing 8-5.

*dirsfirstsort()*

```
int dirsfirstsort(const struct dirent **a, const struct dirent **b)
{
    if ( isdir(*a) )
        if (! isdir(*b) ) /* a is a directory but b is not.         */
            return -1;
        else /* Both a and b are directories; sort alpabetically.   */
            return (alphasort(a,b));
    else
        if ( isdir(*b) )  /* b is a directory but a is not.         */
            return 1;
        else /* Neither a nor b is a directory; sort alpabetically. */
            return (alphasort(a,b));
}
```

*Listing 8-5: A comparison function that sorts directories before non-directories, and alphabetically if the two entries are the same*

This function orders the entries by sorting directories ahead of non-directories and breaks ties alphabetically. It calls the `isdir()` function presented in Listing 8-2 to determine whether an entry is a directory or not.

The next step is to write the function that prints a single directory's entries using `scandir()`. This function, which I've named `scan_one_dir()`, is displayed in the following listing:

*scan_one_dir()*

```
int scan_one_dir(const char* dirname, ❶ void (*process)(const struct dirent*))
{
    struct dirent **namelist;  /* An array of pointers to dirent structs   */
    int i,n;
```

```
    errno = 0;
    if ( (n = scandir(dirname, &namelist, NULL, dirsfirstsort) ) < 0)
        fatal_error(errno, "scandir");

    for (i = 0; i < n; i++){   /* Process every entry saved into namelist. */
        process(namelist[i]);  /* Process this dirent structure.           */
        free(namelist[i]);     /* Free the dirent structure.               */
    }
    free(namelist);            /* Free the namelist array that was
                                  allocated by scandir().                  */
    return(EXIT_SUCCESS);
}
```

This function passes `dirsfirstsort()` to `scandir()` as its comparison function and uses no filter, so that no entries are excluded from the `namelist` array.

Now that we've seen how to use function pointer parameters, I'm taking advantage of them here. Rather than designing this function narrowly so that it can only print the filenames in the entries, I make it more general. Specifically, its second parameter ❶ is a pointer to a function that will process the `dirent` structures saved in `namelist`. The function pointer parameter is named `process()` in the listing and has a `dirent` structure argument and a `void` return value. We can pass any `void` function to it that has a single `dirent` structure argument. Since our program just prints filenames, the function passed to this parameter is one that just prints the entry's filename. Another program could pass a different function to it, for example, one that calls `stat()` on the filename to retrieve its metadata and process that metadata in some particular way.

We're ready to assemble the program. To save space, the listing does not show those functions already displayed in previous listings; the complete program is in the book's source code distribution.

*spl_ls3.c*

```
#define _DEFAULT_SOURCE      /* For glibc > 2.10               */
#define _BSD_SOURCE          /* For versions of glibc <  2.19  */
#include "common_hdrs.h"
#include <dirent.h>
#include "dir_utils.h"       /* for isdir() and dirsfirstsort()  */

/* print(dp) prints the filename of entry *dp. If it's a directory, it
    appends a trailing '/' to its name. */
void print( const struct dirent* direntp )
{
    printf("%s", direntp->d_name);
    if ( isdir(direntp) )
        printf("/");
    printf("\n");
}

int main(int argc, char* argv[])
```

```
{
    if ( setlocale(LC_TIME, "")  == NULL )
        fatal_error( LOCALE_ERROR,
                 "setlocale() could not set the given locale");
    if ( 1 == argc ) /* If no arguments, list the CWD.                    */
        scan_one_dir(".", print);
    else {           /* Otherwise, for each argument, scan it.           */
        int i;
        for (i = 1; i < argc; i++ ) {
            printf("\n%s:\n", argv[i]);     /* Print the argument.        */
            scan_one_dir(argv[i], print);   /* pass the print() function. */
            if ( i < argc-1 ) printf("\n"); /* Put a newline before next. */
        }
    }
    exit(EXIT_SUCCESS);
}
```

*Listing 8-6: A program using `scandir()` to list all contents of all arguments, sorted in collating order, with directories preceding non-directories*

After setting up the current locale, the main program calls `scan_one_dir()` passing it the `print()` function for each command-line argument. We build it and run it on our test directory:

```
$ spl_ls3  testing
testing:
./
../
dir1/
dir2/
dir3/
file1
file2
file3
```

Because of space limitations, this output is intentionally small. You can run this program on much larger directories to see that it sorts the entries in the correct order.

The various library functions that we've examined in the directory API are all designed to work within a single directory, as a flat structure. They don't descend into subdirectories. We need to know how to design programs that can descend into subdirectories and process entire directory hierarchies, because many problems require this, as we'll see in the next section. We're about to explore various ways to accomplish this, including a few different APIs designed to recursively descend directories.

## Processing the Directory Hierarchy

In Chapter 1, I pointed out that the directory hierarchy is tree-like, but is not a tree. Let's review why this is true. The first reason is that symbolic links can create cycles, because a symbolic link can point back to an ancestor in the tree, as depicted in Figure 8-2. In the figure, directories are in bold and the symbolic link is a dashed line.

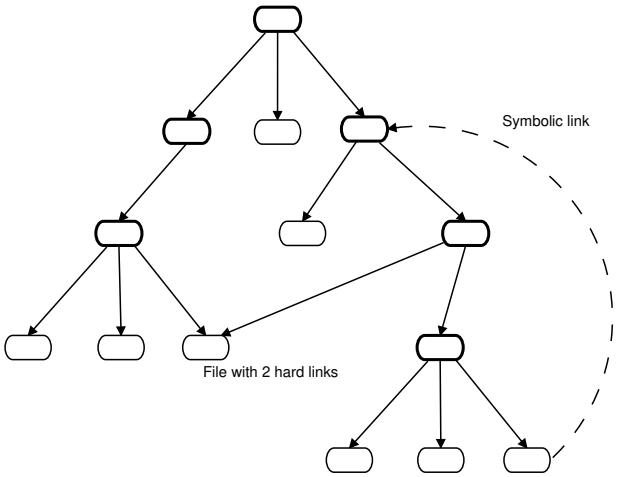



*Figure 8-2: A portion of the directory hierarchy with a symbolic link creating a cycle*

If none of the symbolic links created cycles, the hierarchy still wouldn't be a pure tree because hard links allow files to have multiple names in different directories, as that same figure illustrates. Files that are contained in more than one directory are nodes with more than one parent, but in a tree, each node other than the root has a single parent.

In Linux and most Unix systems, the hierarchy has no cycles if it has no symbolic links. This is because, in these systems, a directory node can never be the target of a hard link, which implies that no node can have an edge leading back to an ancestor. For brevity, even though it isn't technically a true tree, we'll refer to a directory hierarchy as a *directory tree*, or just a *tree*, when the meaning is clear.

The fact that the hierarchy is tree-like suggests that many of the algorithms that we use to process trees can be applied to process the hierarchy. Some of these algorithms are important enough for us to explore now, because the paradigms that they embody are the basis for many practical and useful Unix tools.

Some of the most useful tools in Unix allow us to traverse the entire directory tree rooted at a given node, performing some type of processing in each node. This is what the `find` command does, for example. It lets us search the tree rooted at a given directory, searching for files that satisfy specified conditions and performing actions on the files that satisfy those conditions. In the simplest case, we can use `find` to search for files whose

names match a pattern and print out the relative pathnames of those that do.

Other commands such as `ls`, `rm`, `cp`, and `grep` have a recursive option, usually either `-R` or `-r`, that make those commands act on every file in the directory trees rooted in their directory arguments. All of the preceding commands work top-down, usually in a depth-first manner, processing an entire subtree before processing any sibling subtrees.

Some tools, such as `tree` and `du`, short for *disk usage*, also work on entire directory trees. The `tree` command displays the entire directory tree rooted at a given directory, visually indenting files at deeper levels of the tree. The `du` command summarizes the amount of disk space used by files in a given directory hierarchy. Its man page states that it acts recursively on directories. In other words, when we enter **`du` *`dirname`***, `du` recursively descends *`dirname`*, collecting and reporting the total disk usage of every directory in its tree, after which it prints the grand total of all directories. For example, this is a run of it on the *testing* directory:

```
$ du testing
4 testing/dir3
4 testing/dir1/adir
8 testing/dir1
4 testing/dir2
32 testing
```

This output suggests that `du` processes all child nodes before it processes their parent, which is an example of a *post-order traversal*. The numbers in the output are counts of the number of 1024-byte blocks used by all files in each directory; we'll explore the command in more detail later in this chapter.

All of the preceding examples traversed the tree by *descending* it, visiting all nodes or selected nodes in the subtree rooted at a given node. Algorithms that traverse the tree in this manner are called *tree walks*. In contrast to tree walks, some commands *ascend* the tree. The `pwd` command is a good example of this; it travels up the tree starting in the current working directory until it reaches the root directory, in order to construct the absolute pathname of the current working directory. Traveling up the tree presents interesting challenges that are very different from those that we'll encounter in trying to traverse the tree downward and recursively.

Among the various commands that walk the tree, the `du` command is a good one to implement; we'll learn a lot in the process. Since we already know how to use the `stat()` family of system calls to obtain disk usage metadata for a file, collecting disk usage metadata won't be difficult. The challenge is how to walk the tree recursively, and in particular, to process its nodes in a post-order manner. Therefore, our approach to solving this problem is first to develop a couple of programs that just walk the tree, not trying to collect disk usage information. When we've worked out the algorithm for performing the tree walk, we'll augment it with the ability to report disk usage.

We'll also learn a lot by tackling a problem that has to ascend the tree. Implementing a version of the `pwd` command is a good exercise in ascending the tree. We'll discover that it isn't as simple as it might seem.

Before we start researching solutions to these problems we need to learn a bit more about the concept of filesystem mounting, which is what we cover next. After that, we'll explore potential ways to perform tree walks, as preparation for implementing the `du` command. Once we decide on a good method, we'll implement a simplified version of that command.

## Mounting File Systems

In Chapter 1 I introduced the concept of filesystem mounting, which I'll explain in more detail now because it's relevant to the problems we're about to solve. It's easiest to understand with a concrete example. The superuser can issue the `mount` command to mount a filesystem onto the directory hierarchy at a specific point; the command

```
$ mount device  directory
```

attaches the filesystem on the given device into the directory hierarchy at the specified directory. That directory is then called the filesystem's *mount point*. In general, you need superuser privilege to mount a filesystem, and you typically need to provide the command more information than in this example, such as the filesystem type and its unique identifier.

When a filesystem is mounted onto the directory hierarchy, the root directory of that filesystem replaces the directory on which it's mounted, and that directory's previous contents are hidden by the mount. When the filesystem is *unmounted*, using the `umount` command (not `unmount`)), the directory's contents are restored.

### An Example of Mounting

To illustrate, Figure 8-3 depicts a portion of the top of the directory hierarchy of a hypothetical Unix system.

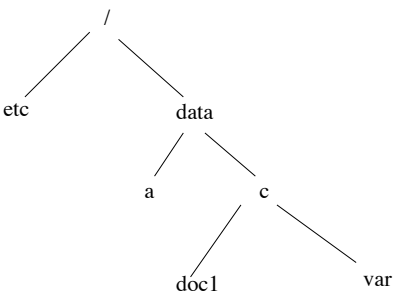



*Figure 8-3: Initial file hierarchy without any mounts*

The root of this hierarchy has a subdirectory named *data* with two subdirectories named *a* and *c*. The *c* directory is not empty. Suppose that there's another device, */dev/hdb*, that has a filesystem on it, depicted in Figure 8-4.

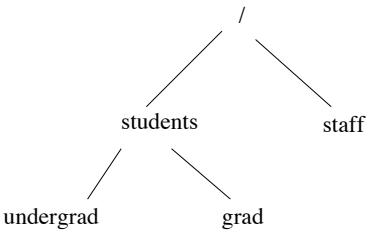



*Figure 8-4: File system* /dev/hdb

The root of this filesystem has two subdirectories named *staff* and *students*, and *students* has subdirectories *grad* and *undergrad*. Now suppose we mount this second filesystem on the directory */data/c* by entering (as superuser):

```
$ mount /dev/hdb /data/c
```

If the mount is successful, then */data/c* becomes the mount point for the filesystem */dev/hdb*, and we say that */dev/hdb* is mounted on the directory *c*. The files *doc1* and *var* are hidden until the file system is unmounted, at which point they'll reappear. The directory hierarchy after the mount is depicted in Figure 8-5.

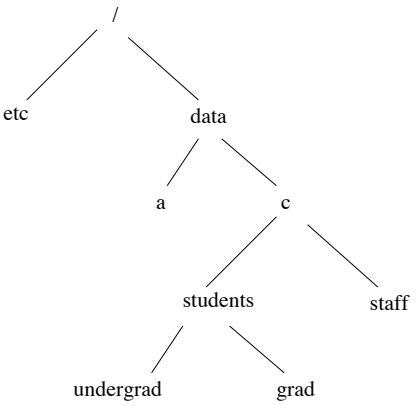



*Figure 8-5: Directory hierarchy after the mount of* /dev/hdb *onto* /data/c

The absolute pathnames of all files in the mounted filesystem start with */data/c* now, such as */data/c/students/grad*.

When a directory becomes a mount point, the kernel restructures the directory hierarchy. Although the directory contents are masked by the root directory of the mounted file system, the kernel stores the hidden contents and a record of the mount. Different versions of Unix implement mounting in different ways; implementation is not part of any standard. However, it's pretty much universally true that a process can recognize when a directory *dir* is a mount point because the device ID of the directory's parent, say *parent*, is different from that of *dir*. This is because *dir* is the root of the mounted filesystem and *parent* is a node on the filesystem to which it's attached.

### Finding Mount Points

There are a few ways to tell where the mount points are in the directory hierarchy. One is the `df` command, which is intended to show the amount of disk space available on the mounted filesystems that contain the filenames it's given on the command line. By default, the POSIX-conforming version of it outputs several fields on each line, but the GNU version, found on Linux, allows us to limit its output to just the name of the device and its mount point by giving it the `--output=source,target` option. Without any filenames it shows all mounted filesystems:

```
$ df --output=source,target
--snip--
/dev/sdc2      /
/dev/sdd3      /var
/dev/sdb3      /home
/dev/sdc1      /boot
/dev/sdd4      /data/research_resources/physics/articles/more_articles
```

If we want to know the filesystem and mount point for our current working directory, then entering **`df --output=source,target .`** shows us:

```
$ pwd   # To see what our working directory is
/home/stewart/unixbook/demos
$ df --output=source,target  .
Filesystem     Mounted on
/dev/sdb3      /home
```

We can also use the `mount` command. Without any options, it displays all mounted filesystems with information about each. By giving it the `-t` *type* option, it limits the output to mounts of the requested type. For example, to see the Ext4 filesystems, I can enter

```
$ mount -t ext4
--snip--
/dev/sdc2 on / type ext4 (rw,relatime,errors=remount-ro)
/dev/sdd3 on /var type ext4 (rw,relatime)
/dev/sdb3 on /home type ext4 (rw,relatime)
/dev/sdc1 on /boot type ext4 (rw,relatime,stripe=4)
/dev/sdd4 on /data/research_resources/physics/articles/more_articles  ...
```

This command also displays the mount options, which I haven't discussed here. You can read about mount options on the `mount` man page.

A third method is the `findmnt` command, which also displays mounted filesystems. By default, its output is presented in a tree-like format using HTML, and the fields are similar to `df`'s output. We can limit the output to just the filesystem and mount point with the `-o SOURCE,TARGET` option, and limit the types with `-t` *type*.

```
$ findmnt -t ext4  -o SOURCE,TARGET
```

```
SOURCE    TARGET
--snip--
/dev/sdc2  /
/dev/sdb4  /data
/dev/sdd4  /data/research_resources/physics/articles/more_articles
/dev/sdd3  /var
/dev/sdb3  /home
/dev/sdc1  /boot
```

This is just a brief summary of these commands. The `mount` command is also used for mounting filesystems, but you need superuser privileges on the computer for that purpose.

### Duplicate Inode Numbers

The advantage of mounting is that it simplifies the user's conceptualization and navigation of the file hierarchy. One problem that it introduces is that there may be files with the same inode number in the directory hierarchy, since inode numbers are unique only within a single filesystem.

In fact, in most Unix systems, and in Linux in particular, the root directory of every filesystem is inode number 2. Inode number 1 is used to record bad blocks in the file system, and index 0 is unused in the inode table. A typical Unix system may have several filesystems all mounted directly under `/`. Viewing the inode numbers in the top-level directory, you're likely to see several subdirectories all of which have inode number 2, because they're all mount points for attached filesystems.

Given that multiple files can have the same inode numbers when filesystems are mounted on the directory hierarchy, the only way to uniquely identify a file is with a pair consisting of the inode number and the device ID of the filesystem on which it resides. That number is always stored in the inode. Without the device ID, the inode number is ambiguous. This is also why the kernel doesn't allow us to create a hard link for a file on a different filesystem. To see this, suppose in the preceding example that the file */data/a/doc1* has inode number 52. Suppose that the file */students/undergrad/hwk1* on */dev/hdb* also has inode number 52. If we could create a hard link across filesystems, then the command

```
$ ln  /data/a/doc1   /data/c/students/grad/doc1
```

would result in two links in the `/dev/hdb` filesystem, each having the same inode number, but these two inode numbers would refer to two different inodes. This would break the filesystem, unless directories were able to store device numbers as well as inode numbers with filenames, which would require rewriting a lot of the kernel. All hard links to a file must be in a single file system.

The subject of mounting and mount points will play a role in our solutions to the remaining problems of this chapter.

## Tree Walks

Let's turn to the problem of walking through a directory tree. One way to walk a directory's tree is to implement a recursive function using only the first set of library functions from the preceding section, namely `opendir()`, `readdir()`, and `closedir()`. With these, we only need to modify our *spl_ls1.c* program slightly.

### A Recursive Tree Walk Using readdir

To modify the *spl_ls1.c* program so that it can visit the entire tree rooted at a given directory, we need to change the main program slightly and revise `listdir()`. The main program will still open the root directory of the tree walk, as it did before, by calling `opendir()`. However, since `listdir()` will also need to open any directory it finds as it's reading the directory stream, it will need the name of the directory it's processing. Let's redisplay the main loop of `listdir()` to make this clear.

```
while ( !done ) {
    errno = 0;
    direntp = readdir( dirp );
    if ( direntp == NULL && errno != 0 )
        perror("readdir");
    else if ( direntp == NULL )
        done = TRUE;
    else
        printf("    %s\n", direntp->d_name );❶
}
```

In this code, we need to replace the call to `printf()` ❶, by programming logic such as the following, which excludes the requisite error-handling:

```
printf("%s\n",direntp->d_name);
if ( isdir(direntp) ) {
    subdirp = opendir(direntp->d_name);❷
    listdir(subdirp, flags);
    closedir(subdirp);
}
```

In other words, `listdir()` prints the entry's filename and then checks if it's a directory. If it is, it recursively calls `listdir()`. Unfortunately this doesn't work, for two reasons.

The first is best explained with an example. Consider the fragment of a directory tree shown in Figure 8-6.

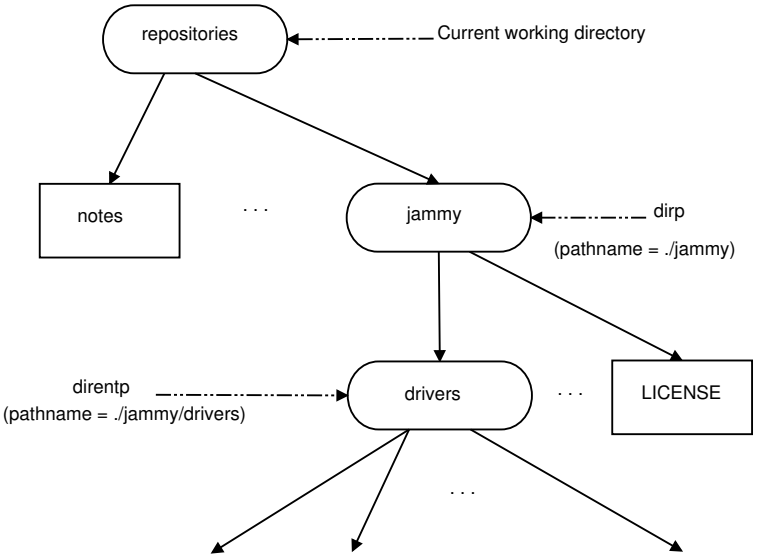



*Figure 8-6: A directory being processed by `opendir()` and `readdir()`*

The figure shows that the main program's working directory at the time it's called is *repositories*. Suppose that it was invoked with the command

```
$ spl_ls1 jammy
```

The `main()` function in turn calls `opendir("jammy")`. Since *jammy* is a pathname relative to the current working directory *"repositories"*, the call is successful; the *jammy* directory is opened and the `dirp` directory stream pointer is returned, which `main()` passes in its call `listdir(dirp)`. Inside `listdir()`, while processing the *jammy* directory, `readdir()` is called and returns a `direntp` pointer to the subdirectory named *drivers*. The `isdir(direntp)` function detects that it is a directory and calls `opendir(direntp->d_name)`. This is the problem. At that point `direntp->d_name` is the name `"drivers"`, but the pathname *drivers* is only relative to *jammy*, not to the process's current working directory, *"repositories"*. Therefore `opendir()` will fail with the `ENOENT` error, ("No such file or directory"). The pathname that should be passed to `opendir()` is *jammy/drivers*, not *drivers*.

More generally, the pathname passed to `opendir()` must be relative to the current working directory. In the current implementation of `listdir()`, the relative pathname of the directory being processed isn't available in the `listdir()` function's parameter list. Therefore, we need to modify its prototype to include it and also modify the calls to it in `main()`. The `listdir()` function needs to declare a variable to store the longest possible pathnames that might be needed for this purpose. We'll therefore declare a variable, `pathname` in it, with size `PATH_MAX`, defined in *limits.h*.

The second problem is related to the dot and dot-dot entries. They're both directories. When `readdir()` reads the entry for dot, `isdir()` returns true. The recursive call to `listdir()` will start right back in the beginning,

and an infinite loop will ensue because the program will keep returning to process the same directory over and over again!

The solution to this problem is to check whether the current entry is `.` or `..` and skip it if it is. We can now assemble a correct version of the recursive solution. The revised code for `listdir()` and the main program are shown in the following listing. We'll name this recursive version of our program, *spl_ls_rec.c*. For brevity, the include directives and some comments are omitted.

*spl_ls_rec.c*

```
void listdir( DIR *dirp, char *dirname,  int flags )
{
    struct dirent *direntp;       /* Pointer to directory entry structure  */
    BOOL          done = FALSE;   /* Flag to control loop execution        */
    char          pathname[PATH_MAX];  /* Pathname of file to open         */
    DIR           *subdirp;       /* Dir stream for subdirectory           */
    char          *name;

    while ( !done ) {
        errno = 0;
        direntp = readdir( dirp );
        if ( direntp == NULL && errno != 0 )
            perror("readdir");
        else if ( direntp == NULL ) /* Implies end of stream. */
           done = TRUE;
        else {
            name = direntp->d_name;
            if ((strcmp(name, ".") != 0) && (strcmp(name, "..") != 0)) {
                sprintf(pathname, "%s/%s", dirname, name);
                printf("%s\n",pathname);
                if ( isdir(direntp) ) {
                    errno = 0;
                    if ( ( subdirp = opendir(pathname) ) == NULL )
                        error_mssge(errno, name);
                    else {
                        listdir(subdirp, pathname, flags);
                        closedir(subdirp);
                    }
                }
            }
        }
    }
    printf("\n");
}

int main(int argc, char *argv[])
{
    DIR   *dirp;
    int    i;
```

```
    int    ls_flags = 0;

    if ( 1 == argc ) {
        errno = 0;
        if ( ( dirp = opendir(".") ) == NULL )
            fatal_error(errno, "opendir");   /* Could not open cwd. */
        else
           listdir( dirp, ".",  ls_flags );
    }
    else {  /* For each command-line argument, call opendir() on it. */
        for ( i = 1; i < argc; i++) {
            errno = 0;
            if ( ( dirp = opendir(argv[i]) ) == NULL ) {
                if ( errno == ENOTDIR )  /* It's not a directory. */
                    printf("%s\n", argv[i] );
                else  /* It's an error. */
                    error_mssge(errno, argv[i]);
            }
            else {  /* Directory was opened successfully. */
                printf("\n%s:\n", argv[i] );
                listdir( dirp, argv[i], ls_flags);
                closedir(dirp);
            }
        }
    }
    return 0;
}
```

*Listing 8-7: A program to demonstrate recursive listing of a directory hierarchy*

We'll build and run this program on the *testing* directory we used earlier in the chapter (see Figure 8-1):

```
$ spl_ls_rec testing
testing:
testing/dir1
testing/dir1/b
testing/dir1/adir
testing/dir1/adir/d
testing/dir1/c
testing/file3
testing/file1
testing/dir3
testing/file2
testing/dir2
testing/dir2/b2
testing/dir2/a1
```

You can see that it recursively displays the files and directories, but there's no apparent ordering of the files. This is a consequence of using `readdir()` to read the entries, since it doesn't return them in sorted order.

### A Recursive Tree Walk Using scandir()

One way to overcome the lack of sorting is to base a recursive tree walk on *spl_ls3.c* instead. That program used the `scandir()` function, which sorted the filenames using `alphasort()`. To reduce the code size, we'll remove the directory-first processing that we coded into *spl_ls3.c* and concentrate on adding recursion to the program. We'll also remove the function pointer parameter to `scan_one_dir()` and the `print()` function. The `scan_one_dir()` function from that program, modified to include the recursive call, is shown in the following listing, with the changes highlighted in bold.

*scan_one_dir() (revised)*

```
int scan_one_dir(const char* dirname )
{
    struct dirent **namelist;
    int i, n;
    char   pathname[PATH_MAX];

    errno = 0;
    if ( (n = scandir(dirname, &namelist, NULL, alphasort) ) < 0)
        fatal_error(errno, "scandir");

    for (i = 0; i < n; i++) {
        if (strcmp(namelist[i]->d_name, ".") != 0
            && strcmp(namelist[i]->d_name, "..") != 0) {
            printf("%s/%s\n",dirname,namelist[i]->d_name);
            if ( isdir(namelist[i]) ) {
                sprintf(pathname, "%s/%s", dirname, namelist[i]->d_name);
                scan_one_dir(pathname);
            }
        }
        free(namelist[i]);
    }
    free(namelist);
    return(EXIT_SUCCESS);
}
```

Because the only change to the main program is the removal of the function argument to the two calls to `scan_one_dir()`, to save space, we won't re-display it here. The revised program is named *spl_ls_rec2.c* in the book's source code distribution. We build and run this new version on the same *testing* directory:

```
$ spl_ls_rec2 testing
testing:
```

```
testing/dir1
testing/dir1/adir
testing/dir1/adir/d
testing/dir1/b
testing/dir1/c
testing/dir2
testing/dir2/a1
testing/dir2/b2
testing/dir3
testing/file1
testing/file2
testing/file3
```

You can see that the pathnames are all correct and that they're sorted by filename. It might be possible to base our implementation of `du` on this program, but before we make that decision, let's review what `du` does.

The `du` command traverses each directory tree it's given. It accumulates the disk usage of every file in a directory and then prints the disk usage of that directory. It does this recursively, so that if a file is a directory, it first descends into that directory to accumulate the usage, and recursively descends into its subdirectories. This implies that it does, in fact, perform a post-order traversal of each directory tree.

The default block size that it uses for reporting is 1024-bytes, but it depends on the environment variables of the user running it, as well as some system settings. Also by default, `du` doesn't print the disk usage of ordinary files, even though they're added into the directory totals. With the `-a` option it prints the block counts for all files, not just directories. Consider this run of it:

```
$ du -a testing
0 testing/dir1/b
0 testing/dir1/adir/d
4 testing/dir1/adir
0 testing/dir1/c
8 testing/dir1
4 testing/file3
4 testing/file1
4 testing/dir3
4 testing/file2
0 testing/dir2/b2
0 testing/dir2/a1
4 testing/dir2
32 testing
```

Studying its output, we can see that it descends into directories in a depth-first manner, reaches their leaf nodes, returns and prints the total counts for the parent directories. It prints the counts for the directories onlly *after* it

visits all of their children. Figure 8-7 illustrates the portion of the path taken by `du` on the *dir1* subdirectory.

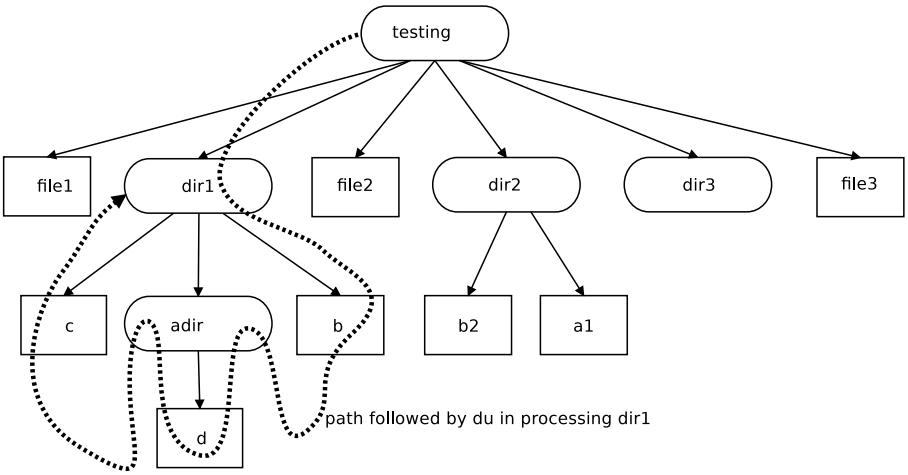



*Figure 8-7: A portion of the path taken by du -a on the* testing *directory.*

It descended into *testing/dir1*, then *testing/dir1/b*, and since that was a leaf node, it then descended into *testing/dir1/adir* and then into *testing/dir1/adir/d*, backed out and visited *testing/dir1/c*. Our goal is to write a program that can can process the tree in the same way.

### The nftw Tree Walk Function

Before deciding how to write our program, let's consider the other potentially useful function mentioned in the `readdir()` man page, namely `nftw()`. Its man page is

```
NAME
       ftw, nftw - file tree walk

SYNOPSIS
       #include <ftw.h>
       int nftw(const char *dirpath,
               int (*fn) (const char *fpath, const struct stat *sb,
                         int typeflag, struct FTW *ftwbuf),
               int nopenfd, int flags);

       #include <ftw.h>
       int ftw(const char *dirpath,
               int (*fn) (const char *fpath, const struct stat *sb,
                         int typeflag),
               int nopenfd);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):
```

```
       nftw(): _XOPEN_SOURCE >= 500

DESCRIPTION
       nftw() walks through the directory tree that is located under the di
       rectory dirpath, and calls fn() once for each entry in the tree.  By
       default,  directories are handled before the files and subdirectories
       they contain (preorder traversal).
__--snip--
```

The `nftw()` function is designed to walk a directory tree. It has a function pointer parameter that can be applied at each node of the tree. Its description states that directories are processed before their files and subdirectories, which implies that it's a pre-order traversal, but further down in the man page, it indicates that by supplying an appropriate flag, it can be made to perform post-order processing.

The man page also has information about a second, related function, `ftw()`, but it notes that this is an older function and that `nftw()` was designed to replace it. The older `ftw()` function is now deprecated.

The `nftw()` function is given four arguments. The first is the pathname of a directory, the second is a function that will be called on each entry that it visits, the third is an integer that specifies the maximum number of file descriptors it's allowed to use, and the last is a set of flags that influence its behavior. Let's go through each of these arguments and how they're used.

The first argument is the root of the tree that it will process. Given the pathname to a directory, `dirpath`, the `nftw()` function recursively descends the directory hierarchy rooted in `dirpath`. For each entry that it finds, it calls the function pointed to by its second argument, `fn()`, passing it the following arguments:

**`fpath`** This is the pathname of the entry. If `dirpath` is a relative pathname, then `fpath` is a pathname relative to the process's current working directory at the time `nftw()` was called. If `dirpath` is an absolute pathname, then `fpath` is also an absolute pathname.

**`sb`** This is a pointer to a `stat` structure containing information about the object, filled in as if `stat(fpath, sb)` or `lstat(fpath, sb)` was called to retrieve the metadata.

**`typeflag`** This is an integer flag that encodes more information about the entry. Its value is exactly one of the following predefined constants:

- **`FTW_F`** The entry `fpath` is a regular file.
- **`FTW_D`** The entry `fpath` is a directory.
- **`FTW_DNR`** The entry `fpath` is a directory that cannot be read by the process. In this case, the `fn()` function won't be called for any of its descendants.
- **`FTW_DP`** The entry `fpath` is a directory and all of its files and subdirectories have been visited already, because the `FTW_DEPTH` flag was set in the `flags` argument of `nftw()`.

**FTW_NS** The stat() function failed on the entry, most likely because the process did not have execute permission on the parent directory. In this case, the stat buffer passed to (*fn) is undefined.

**FTW_SL** The entry fpath is a symbolic link, and the FTW_PHYS flag was set in the flags passwd to nftw().

**FTW_SLN** The entry fpath is a broken, or *dangling* symbolic link, one that doesn't point to an existing file, and the FTW_PHYS flag was not set in flags. In this case the stat buffer was filled with information about the link itself instead of its target.

**ftwbuf** This is a pointer to an FTW structure, which is defined as follows:

```
struct FTW {
     int base;
     int level;
};
```

The FTW structure provides information about the filename in the fpath pathname passed to (*fn). Specifically, base is the character offset of the entry's filename in the fpath pathname. For example, if fpath is *testing/dir1/adir* and the entry being processed is the file *adir*, then base contains the length of the string *testing/dir1/*.

The level member of the structure indicates the depth of the entry relative to the root of the walk, which is the directory passed to the call to nftw(). The root directory is level 0. If a parent node has level $n$, then all of its children are at level $n + 1$. In this example, level would contain 2, since *adir* is two levels below *testing*.

Any programmer-defined function can be passed to the fn function pointer parameter, provided that its prototype matches that of the parameter. If so, nftw() calls this function for every entry that it visits. The function will have access to the stat structure returned by a call to stat() on that entry as well as the information encoded in its typeflag argument. A significant drawback of the (*fn) parameter's declaration is that it has no *hooks* that we can use to pass other data to it. In other words, there are no parameters in the function prototype that a program can use to pass other data items to the function. One consequence of this design is that, in order for this function to access any program variables that can retain data across calls, those variables must either be declared with static linkage or have file scope.

For example, to compute the total number of blocks used by all entries in the subtree rooted at a given directory, we would either have to make the total block count a static variable within the function that we define, or declare the variable with global scope. This will be clear when we look at an example, which we'll do shortly.

The third parameter to the nftw() function, nopenfd, is the maximum number of file descriptors that nftw() should use while traversing the file tree. Each time that nftw() visits a directory, it opens it and obtains its file descriptor. After it descends that directory's subtree and returns to its parent, it closes that descriptor. Therefore, one open file descriptor is needed

for each level of the tree from the root of the search to the current level. If `nopenfd` is smaller than the depth of the tree, then to reach the deeper entries, `nftw()` will be forced to close descriptors of ancestors in the tree in order to continue descending the tree. This degrades its performance. If a process makes `nopenfd` large enough, this problem is avoided, but the number of open file descriptors a process can have is limited by the kernel.

### PROCESS RESOURCE LIMITS

The maximum number of open files that a process is allowed is an example of a *process resource limit*. A process requires many different types of resources, such as memory for its stack, time on the CPU, and storage for open file descriptions. The kernel sets limits on the amount of resources of each type that a process can use. With the appropriate system calls, a process can get the values of these limits and modify them.

At the command level, `prlimit` can be used to query and modify these resource limits. For example `prlimit -n` lists the resource limit on open files:

```
$ prlimit -n
RESOURCE DESCRIPTION              SOFT    HARD UNITS
NOFILE   max number of open files 1024 1048576 files
```

The difference between soft and hard limits, and a discussion of how a process can access and modify system and resource limits, including system calls related to them, is covered in Appendix B.

The fourth parameter of `nftw()` is an integer that can be used to pass in a bitwise-or of zero or more of the following constants, which control aspects of its behavior, such as how `ntfw()` handles mount points and soft links, what it uses as its current working directory, and whether it follows a pre-order or post-order traversal of the tree.

**`FTW_CHDIR`** If set, `nftw()` changes its current working directory to each directory as it processes the files in that directory. If it isn't set, `nftw()` doesn't change the current working directory.

**`FTW_DEPTH`** If set, `nftw()` processes all files in a directory before processing the directory itself; in other words, it performs a post-order traversal. If it isn't set, `nftw()` processes directories before any of their files, which we call *pre-order traversal*.

**`FTW_MOUNT`** If set, `nftw()` does not cross mount points, meaning that it only processes files in the same file system as `fpath`.

**`FTW_PHYS`** If set, `nftw()` performs a physical walk and does not follow symbolic links. If it visits a file that is a symbolic link, it processes the link itself, not its target. If it isn't set, it follows symbolic links but does not visit any file twice. If `FTW_PHYS` is not set and `FTW_DEPTH` is set, `nftw()` follows soft links but does not process any directory that would be a descendant of itself.

**FTW_ACTIONRETVAL** This flag is only available under *glibc* 2.3.3 or later, with _GNU__SOURCE defined to expose it. If it's set, the next node that nftw() visits is determined by the return value of (*fn). The return values that it responds to are FTW_CONTINUE, FTW_SKIP_SIBLINGS, FTW_SKIP_SUBTREE, and FTW_STOP. For example, if (*fn) returns FTW_CONTINUE, then nftw() continues normal processing, whereas if fn() returns FTW_SKIP_SIBLINGS, then nftw() will skip visiting any remaining siblings of the current entry and instead return to the parent. The other two flags will cause it to skip processing subtrees or to stop respectively.

The nftw() function visits the entries in the tree rooted at dirpath until one of the following conditions occurs:

- An invocation of (*fn) returns a non-zero value and FTW_ACTIONRETVAL is not set, in which case nftw() stops and returns that value.
- The FTW_ACTIONRETVAL flag is set and (*fn) returns FTW_STOP, in which case it stops and returns that value.
- It detects an error, in which case it returns -1 and sets errno to indicate the error; or
- It has visited all nodes of the tree, in which case it returns 0.

Let's look at an example. Although the man page for nftw() has an example program, I created a slightly different one whose behavior is similar to that of the tree command. This program displays the name of every file in the tree rooted at its argument directory, indented on the line by an amount of space proportional to its depth in the tree, as a way to visualize the directory hierarchy. The program accepts three user-supplied options with the following meanings:

**-m** : The program does not cross mount points.

**-d** : The program does a post-order traversal instead of a pre-order traversal.

**-p** : The program does not follow symbolic links. Without it, it does.

When we write a program that uses nftw(), all of the logic is essentially in the function that it calls at every node. In this first example, that function is named display_info(). Let's take look at its code, which follows.

*display_info()*

```
#include "common_hdrs.h"
#include <ftw.h>

#define  MAXOPENFD   20       /* Maximum number of file descriptors to open */

int display_info(const char *fpath, const struct stat *sb,
                 int tflag, struct FTW *ftwbuf)
{
    char  indent[PATH_MAX];                      /* A blank string */
    const char *basename = fpath + ftwbuf->base; ❶ /* Filename of entry */
    int   width          = 4*ftwbuf->level; ❷    /* Length of leading path */
```

```
    /* Fill indent[] with a string of 4*level spaces and null-terminate it.*/
    memset(indent, ' ', width);
    indent[width] = '\0';

    /* Print out indent followed by filename (not full path).*/
    printf("%s%-30s", indent, basename );

    /* Check flags and print a message if need be.    */
    if ( tflag == FTW_DNR )       printf(" (unreadable directory)");
    else if ( tflag == FTW_SL )   printf(" (symbolic link)" );
    else if ( tflag == FTW_SLN )  printf(" (broken symbolic link)" );
    else if ( tflag == FTW_NS )   printf(" (stat failed) " );

    printf("\n");
    return 0;               /* Tell nftw() to continue. */
}
```

This function is relatively simple; it doesn't use the `stat` structure argument and it doesn't alter its behavior in response to its `tflag` argument, other than by printing a message based on its value. It uses the `level` and `base` members of the `FTW` structure to indent and format the output. It prints the last component of the pathname by setting `basename` to point to the first character in the pathname after first `ftwbuf->base` ❶ many characters. The indentation is `4*ftwbuf->level` spaces ❷. The `memset()` function fills the memory pointed to by its first argument with a fixed number of identical bytes. It's a convenient and efficient way to create a string with a fixed number of spaces. The main program is also fairly simple:

```
int main(int argc, char *argv[])
{
    int flags = 0;
    int ch;
    char options[] = ":dpm"; /* Three possible options */
    opterr = 0;

    while  (TRUE) {
        if ( -1 == (ch = getopt(argc, argv, options)) )
            break;
        switch ( ch ) {
        case 'd':  flags |= FTW_DEPTH;    break;
        case 'p':  flags |= FTW_PHYS;     break;
        case 'm':  flags |= FTW_MOUNT;    break;
        default:   fprintf (stderr, "Bad option found.\n");
            return 1;
        }
    }
    errno = 0;
```

```
    if (optind < argc)
        while (optind < argc) {
            if ( -1 == nftw(argv[optind], display_info, MAXOPENFD, flags))
                fatal_error(errno, "nftw");
            optind++;
        }
    else if ( -1 == nftw(".", display_info, MAXOPENFD, flags) )
            fatal_error(errno, "nftw");
    else
        exit(EXIT_SUCCESS);
}
```

The main program checks the command line for options and arguments and then calls `nftw()` for each command-line argument, passing it the argument, the function to call, the maximum number of open file descriptors it should use, and optional flags. Without any arguments, it processes the current working directory.

The complete program, named *nftw_demo.c* is in the source code distribution for the book. To demonstrate its behavior, I created a test directory named *testdir*, whose contents are displayed in a tree-like format here:

```
testdir
    linktosubdir1 -> subdir1
    subdir1
        subsubdir1
            link2tosubdir1 -> ../../subdir1
            subsubsubdir1
                testfile1
                testfile2
        subsubdir2
    subdir2
```

This directory has several levels of nested subdirectories one of which has a symbolic link, *link2tosubdir1@*, that creates a cycle. Following is a run of this program on this directory with the `-p` passed to it so that it does not follow symbolic links but displays them instead:

```
$ nftw_demo -p testdir
testdir
    linktosubdir1                 (symbolic link)
    subdir2
    subdir1
        subsubdir2
        subsubdir1
            link2tosubdir1                (symbolic link)
            subsubsubdir1
                testfile1
                testfile2
```

Here's a run without the `-p` option:

```
$ nftw_demo testdir
testdir
    linktosubdir1
        subsubdir2
        subsubdir1
            subsubsubdir1
                testfile1
                testfile2
    subdir2
```

In the first run, it indicates which files are soft links and doesn't follow them. In the second run, it follows the links. It begins by following the *linktosubdir1* link and displays the tree rooted at *subdir1*. When it returns to *testdir*, it doesn't re-enter *subdir1* because, as the man page tells us, `nftw()` does not report on any file twice by following symbolic links.

Now that we see what is possible with the `nftw()` function as well as the `scandir()` function, we need to decide which we should use to implement our initial version of the `du` command. If we use `nftw()`, the `(*fn)` function will need block-scoped (local) variables that have `static` duration or access to file-scoped variables. The alternative is a recursive solution based on `scandir()`. To avoid the recursion, which might be slow, we'll opt to write it based on `nftw()` and see how well we do.

### Writing a du Command

The `du` command has several options, but we'll write a simple version of it that accepts no options. Because we'd like to see the disk usage of all files, we'll implement the equivalent of `du -a` and name it `spl_du`.

The `spl_du` command won't follow symbolic links, otherwise it may overcount file usage or count files that are not within the directory argument. For the first version, it won't cross mount points, so that it reports disk usage only within a single file system. It's easy enough to add an option later to let it cross mount points.

Because the program has to do a post-order traversal of the tree, we'll pass the the `FTW_DEPTH` flag to `nftw()` function. We'll use Figure 8-8 to demonstrate its behavior. Since the way we draw the tree has nothing to do with the order of the files in the directories, for convenience, we can assume that `nftw()` visits the children of a single node, from left to right. Therefore, in the tree in the Figure 8-8 the files are visited in the order *srcs*, *cpy*, *bin*, *pics*, *stuff*, *data*, *work*, *ideas, projects*, *file2*, *garbage*, *misc*, and finally *snw*.

The key problem is how the program can recursively accumulate the sizes of the files that it visits. It has to be able to print out the size of each file that it visits, and when it reaches a directory, to print out that directory's total usage. For example, in Figure 8-8, when it returns to the *data* entry after visiting its children, it has to print the total usage of *pics*, *stuff*, and the size of the *data* directory itself. In addition, it has to add to this sum the sizes of

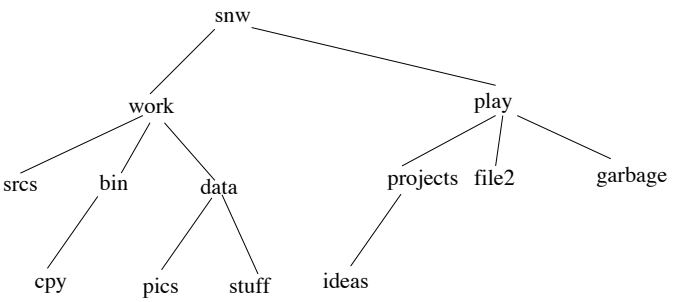



*Figure 8-8: Sample tree hierarchy.*

*srcs*, and the current accumulation in *bin* and add this amount to a running total to pass up to the *work* entry when it returns to it. This suggests that if the program keeps a set of running disk usage totals indexed by the level in the tree, it should be able to record the total disk usage at every directory in the tree, including the directory at the tree's root.

Let's call the function that we pass to `nftw()`, `file_usage()`. Since `file_usage()` has no parameter that can be used to pass any program state information to it, the only way for separate invocations of it and the main program to share data is by putting state information in file scope. Therefore, to record the number of disk blocks used in each level of the tree, the program will need to declare an array

```
#define MAXDEPTH  50  /* some large number to be determined */
static  uintmax_t total_usage[MAXDEPTH];
```

in file scope. It has to be in file scope because the main program needs to initialize each element of the array to zero each time it begins a new tree walk for a directory passed to it, and the `file_usage()` function needs to update it.

The reason to declare the element type `uintmax_t` is that it's the largest unsigned integer type available. This type is declared in *stdint.h*, so we'll need to include that header in the main program. The `system_data_types` man page describes the type and notes that to print values of that type, the program needs to use the `%ju` format specifier in the `printf()` format specification list.

The choice of constant for `MAXDEPTH` is easily changed. If we want, the program could make a system call to obtain the maximum number of open files allowed for the process and dynamically allocate an array of that size, but for now, we'll assume that the depth of the tree is never greater than 50, so we'll define `MAXDEPTH` to be 50.

The prototype for `file_usage()` is

```
int file_usage(const char *fpath, const struct stat *sb,
               int tflag, struct FTW *ftwbuf)
```

At any instant of time, `file_usage()` is visiting a specific file in the tree. Let's call this file the *current file*, and call its level, the *current level*, and let's use

the variable cur_level to represent that level. We'll call the level of the file processed immediately before the current file the *previous level* and we'll use the variable prev_level to store that level.

Both of these variables take on values up to MAXDEPTH and no larger. The current file has a total usage that we can store in the variable named cur_usage. This is the usage of the actual file, not the sum of the disk usage of any children it may have. Directories are usually allocated a single block, by default of size 4096 bytes on most systems. The file_usage() function can get the disk usage of the current file from the stat structure passed into the function; it's in the st_blocks member of the stat structure. This value is the number of 512-byte blocks; in order to print the number of 1024-byte blocks, we'll divide it by 2.

Let's think about what file_usage() has to do for each visited entry. Its actions depend entirely on the values of both cur_level and prev_level. To maake the discussion precise, we define a *left sibling* of a tree node as a sibling that is to the left of that node in the tree's depiction, and a *right sibling* analogously. The file_usage() function must ensure that the following *invariant* assertion is true immediately after it has finished processing a file:

> total_usage[cur_level] is the sum of the sizes of all trees whose roots are at level cur_level and are left siblings of the current file, plus the size of the subtree rooted at the current node.

Suppose first that prev_level < cur_level. This implies that we just descended from a node closer to the root of the tree. There is only one way in which this can happen during a post-order traversal — when we reach a leaf node that is leftmost in its tree. For any other node, either the previous node will be at the same level or will be below it. Therefore, in this case, we've just reached a bottom level of the tree and we need to set total_usage[cur_level] to the current file's usage and copy this into total_usage[cur_level]:

```
cur_usage = sb->st_blocks/2;
total_usage[cur_level] = cur_usage;
```

Observe that total_usage[cur_level] satisfies the invariant assertion in this case.

Let's consider the next case, in which prev_level == cur_level. In this case we're visiting a file that is a right sibling of the one previously visited and it cannot be a directory, because if it were, we'd be returning to it from a node at a greater level. This implies that we have visited all left siblings of the current file and that the current file has no children. Therefore, we need to update total_usage[cur_level] by adding the current file's disk usage to it:

```
cur_usage = sb->st_blocks/2;
total_usage[cur_level] += cur_usage;
```

Assuming that the invariant was true prior to this call to file_usage(), it remains true as a result of adding cur_usage to it, since cur_usage is the disk usage of the subtree rooted at this file and total_usage[cur_level] is now the to-

tal usage of the trees rooted at the left siblings of this node plus this node's total usage.

The last case to consider is when `prev_level > cur_level`. In a post-order traversal, this can only occur when the previous node is a child of the current node and the program has just returned to a directory all of whose children have been visited. Foe example, Figure 8-8, if the current node is *play*, the previous node must be *garbage*, since we visit them i a left-to-right order. Therefore, the total disk usage accumulated in the previous node's level must be added to the disk usage of this parent directory, and this sum must be printed as the total usage of this directory.

We now take advantage of the invariant assertion with respect to `total_usage[prev_level]`. The algorithm only returns to a parent directory immediately after visiting its rightmost child. Since the last node processed was the rightmost child, and its level is `prev_level`, `total_usage[prev_level]` must be the sum of the usages of all subtrees of this directory. Therefore the disk usage to display for this directory is the number of blocks used by the directory plus `total_usage[prev_level]`:

```
cur_usage    = total_usage[prev_level] + sb->st_blocks/2;
```

To preserve the invariant assertion, we also need to add the new value of `cur_usage` to `total_usage[cur_level]`. You should convince yourself that, by doing so, the invariant is true for `total_usage[cur_level]`. The next step is the less obvious one — we must reset `total_usage[prev_level]` to 0 — so that the combined actions are:

```
cur_usage    = total_usage[prev_level] + sb->st_blocks;
total_usage[cur_level] += cur_usage;
total_usage[prev_level] = 0;
```

To see why we have to zero `total_usage[prev_level]`, consider what would happen when `file_usage()` returns and is then called for the next file. Using the file tree in Figure 8-8, suppose the current file is the directory `work`, and `file_usage()` just processed the directory named *data*. Then `cur_level` = 1 and `prev_level` = 2. The next file that `file_usage()` will process is *ideas*, and then *projects*. After it visits *ideas* and returns to *projects*, `total_usage[1]` must have the value 0, otherwise the size of `projects` will include the sizes of *srcs*, *bin*, and *data*. In other words, every time that we finish a level of siblings in a subtree, having reached the rightmost sibling, and return to their parent, we must zero out the entry in the `total_usage[]` array for the children's level. The only chance to do this is when we've added its value into the total usage of the parent and are finished with that node. Doing this preserves the invariant, since no nodes are currently being visited in that level anymore.

We're ready to assemble the initial version of the `spl_du` command. Our command will print a message next to a file name if that file had a problem such as being a broken link or an unreadable directory. The real `du` command prints a message to the standard error stream instead. To reduce the size of the listing, I omitted comments. The complete listing is in the source code distribution.

*spl_du.c*

```
#include "common_hdrs.h"
#include <ftw.h>
#include <stdint.h>
#include <limits.h>

#define  MAXDEPTH  100
static  uintmax_t total_usage[MAXDEPTH];

int file_usage(const char *fpath, const struct stat *sb,
                 int tflag, struct FTW *ftwbuf)
{
    static int prev_level = -1;
    int        cur_level;
    uintmax_t  cur_usage;

    cur_level = ftwbuf->level;
    if ( cur_level >= MAXDEPTH ) {
        fprintf(stderr, "Exceeded maximum depth.\n");
        return -1;
    }
    if ( prev_level == cur_level ) {
        cur_usage = sb->st_blocks/2;
        total_usage[cur_level] += cur_usage;
    }
    else if ( prev_level > cur_level ) {
        cur_usage    = total_usage[prev_level] + sb->st_blocks/2;
        total_usage[cur_level] += cur_usage;
        total_usage[prev_level] = 0;
    }
    else {
        cur_usage = sb->st_blocks/2;
        total_usage[cur_level] = cur_usage;
    }
    printf("%ju\t%s", cur_usage, fpath);
    prev_level = cur_level;

    if ( tflag == FTW_DNR )      printf(" (unreadable directory)\n");
    else if ( tflag == FTW_SL )  printf(" (symbolic link)\n" );
    else if ( tflag == FTW_SLN ) printf(" (broken symbolic link)\n" );
    else if ( tflag == FTW_NS )  printf("stat() failed\n" );

    return 0;   /* To tell nftw() to continue */
}

int main(int argc, char *argv[])
{
    int flags = FTW_DEPTH | FTW_PHYS | FTW_MOUNT;
```

```
    int status;
    int i = 1;

    if ( argc < 2 )  {
        memset( total_usage, 0, MAXDEPTH*sizeof(uintmax_t));
        if ( 0 != (status = nftw(".", file_usage, 20, flags)) )
           fatal_error(status, "nftw");
    }
    else
        while (i < argc) {
            memset( total_usage, 0, MAXDEPTH*sizeof(uintmax_t));
            if ( 0 != ( status = nftw(argv[i], file_usage, MAXDEPTH, flags)) )
                fatal_error(status, "nftw");
            i++;
        }
    exit(EXIT_SUCCESS);
}
```

*Listing 8-8: An implementation of a simplified version of the du command*

Running `spl_du` and `du -a` on a test directory produces the same block counts and list of files:

```
$ du -a testdir
4 testdir/subdir2
0 testdir/subdir1/subsubdir1/subsubsubdir1/testfile2
0 testdir/subdir1/subsubdir1/subsubsubdir1/testfile1
4 testdir/subdir1/subsubdir1/subsubsubdir1
0 testdir/subdir1/subsubdir1/link2tosubdir1
8 testdir/subdir1/subsubdir1
4 testdir/subdir1/subsubdir2
16 testdir/subdir1
0 testdir/linktosubdir1
24 testdir
$ spl_du testdir
4 testdir/subdir2
0 testdir/subdir1/subsubdir1/subsubsubdir1/testfile2
0 testdir/subdir1/subsubdir1/subsubsubdir1/testfile1
4 testdir/subdir1/subsubdir1/subsubsubdir1
0 testdir/subdir1/subsubdir1/link2tosubdir1 (symbolic link)
0 testdir/subdir1/subsubdir1/testfile1.lnk
8 testdir/subdir1/subsubdir1
4 testdir/subdir1/subsubdir2
16 testdir/subdir1
0 testdir/linktosubdir1 (symbolic link)
24 testdir
```

However, this doesn't mean it's correct. If we test it on a few more directories, we'll discover a problem. I constructed another test directory, named *testdir2*, containing a file named d1 with 560 1K blocks:

```
$ ls -s testdir2/d1
560
```

I then created a few hard links to *d1*:

```
$ cd testdir2; ln d1 d2 ; ln d1 d3
$ cd ..
```

and ran `du -a` on *testdir*:

```
$ du -a testdir2
560 testdir2/d1
564 testdir2
```

Not only don't we see *d2* and *d3* in the output, but the total usage doesn't include them, as it shouldn't, because they're just different names for the single file *d1*. Running our version of the command produces different output:

```
$ spl_du testdir2
560 testdir2/d1
560 testdir2/d3
560 testdir2/d2
1684 testdir2
```

This program, as it stands, counts files with multiple links as many times as they have links. If a file has two names in two different subdirectories of our root directory, each will be counted. Since the purpose of this command is to report the amount of disk space a directory tree uses, it isn't useful in its current state; we need to modify it.

How can we fix it? Since every directory entry has the number of the inode for the file, we could just check each time whether we've already added the disk usage for the actual file by saving the inode number each time we process a file. The `stat` structure contains the inode number, `st_ino`, of the file and the link count, `st_nlink`. If the link count is only one, we know there are no other links, but if it's greater than one, there might be. Unfortunately, this idea won't work, which I'll explain.

On any single filesystem, every file is uniquely identified by its `st_ino` value. If we allow the program to cross mount points, then the inode number does not uniquely identify files, because there can be files with the same inode number in two different filesystems. The program might think it's already added disk usage for the current file when it hasn't, because the inode number refers to a file on a different device. We need to know the device as well. The `stat` structure contains the device id, `st_dev`, of the filesystem in which the file is located, and we need to make it part of a file's unique identification.

Suppose that we create a set named `visited` that stores the `(st_ino, st_dev)` pairs of all non-directory files that the tree walk has visited that have two or more links. Initially `visited` would be empty. Each time that the tree walk visits a new entry, it would check whether it's a non-directory file with a link count greater than one. If so, it would check whether the entry is already in `visited`. If it is, it would skip the entry, and if not, it would process it and add the entry to the set. This modification would prevent double-counting of files with more than one name. The only reason for checking whether the link count is greater than one is to save space in the set and save time, because there's no benefit to storing entries in the set if they only have one name.

There are several ways to implement the `visited` set, but the two most efficient would be either a hash table or a search tree. The hash table should have close to $\Omega(1)$ performance for each search and/or insertion if it has a good hash function and enough storage capacity. A search tree would require $\Omega(n \log n)$ steps to search a set of size $n$. I decided to use a hash table. Let's name a second version of the program, with these corrections, *spl_du2.c*.

I don't include the implementation of the hash table here; the file *hash.c* is available in the source code distribution for the book. The header file *hash.h* exposes the following functions from *hash.c* that the program will call:

```
typedef unsigned long long  hash_val;
BOOL insert_hash ( hash_table* h, hash_val val );
BOOL is_in_hash  ( hash_table  h, hash_val val);
void init_hash   ( hash_table* h, int initial_size  );
void free_hash   ( hash_table* h);
```

Although the definition of the `hash_table` is also in *hash.h*, we don't need to see it to use these functions. However, for the same reason that `total_usage[]` needs to be in file scope, the hash table representing the `visited` set must also be in file scope in our program:

```
static  uintmax_t total_usage[MAXDEPTH];  /* Total disk usage for level n   */
static  hash_table  visited;              /* Set of inodes already visited  */
```

The program needs to hash `(st_ino, st_dev)` pairs, but the functions `is_in_hash()` and `insert_hash()` expect a single number. Therefore, if the `file_usage()` function were to call `is_in_hash()` and `insert_hash()` directly, those functions would have to be modified so that they accepted both an inode number and a device id as arguments.

Rather then designing them to accept two numbers, I took the approach of designing a more general hash table that could be used by other programs. Our program can encode the inode number and the device id into a single `unsigned long long int` before calling these functions. We don't need a sophisticated encoding algorithm to encode the inode number and the device id. There are typically only a very small number of separate filesystems on a single computer, so multiplying the inode number by the device id should be sufficient. We can always fine-tune the encoding at a later time.

The following two functions are essentially wrappers for the calls to the hash table functions:

```
/* was_visited(i,d) returns TRUE if the pair was already visited. */
BOOL  was_visited(ino_t inode, dev_t dev)
{
    hash_val  val = inode * dev ;
    return is_in_hash(visited, val);
}

/* mark_visited(i,d) inserts (i,d) into the visited set and returns
   TRUE if successful and FALSE on an error. */
BOOL  mark_visited(ino_t inode, dev_t dev)
{
    hash_val  val = inode * dev ;
    return insert_hash(&visited, val);
}
```

The revised `file_usage()` function follows, with the changed portions highlighted in bold.

*file_usage() (revised)*

```
int file_usage(const char *fpath, const struct stat *sb,
                int tflag, struct FTW *ftwbuf)
{
    static int prev_level = -1;
    int        cur_level;
    BOOL       already_visited = FALSE;
    uintmax_t  cur_usage;

    cur_level = ftwbuf->level;
    if ( cur_level >= MAXDEPTH ) {
        fprintf(stderr, "Exceeded maximum depth.\n");
        return -1;
    }
    if ( prev_level == cur_level ) ❶ {
        if ( sb->st_nlink == 1 ) {
            cur_usage = sb->st_blocks/2;
            total_usage[cur_level] += cur_usage;
        }
        else {
            already_visited = was_visited(sb->st_ino, sb->st_dev);
            if ( !already_visited) {
                cur_usage = sb->st_blocks/2;
                total_usage[cur_level] += cur_usage;
                if ( ! mark_visited(sb->st_ino, sb->st_dev) )
                    fatal_error(-1, "Could not insert inode into hash table");
            }
        }
```

```
    }
    else if ( prev_level > cur_level ) ❷ {
        cur_usage    = total_usage[prev_level] + sb->st_blocks/2;
        total_usage[cur_level] += cur_usage;
        total_usage[prev_level] = 0;
    }
    else ❸{
        if ( sb->st_nlink == 1 || S_ISDIR(sb->st_mode) ) {
            cur_usage = sb->st_blocks/2;
            total_usage[cur_level] = cur_usage;
        }
        else {
            already_visited = was_visited(sb->st_ino, sb->st_dev);
            if (!already_visited ) {
                cur_usage = sb->st_blocks/2;
                total_usage[cur_level] = cur_usage;
                if ( ! mark_visited(sb->st_ino, sb->st_dev) )
                    fatal_error(-1, "Could not insert inode into hash table");
            }
        }
    }
    if ( ! already_visited ) {
        printf("%ju\t%s", cur_usage, fpath);
        if ( tflag == FTW_DNR )      printf(" (unreadable directory)\n");
        else if ( tflag == FTW_SL )  printf(" (symbolic link)\n" );
        else if ( tflag == FTW_SLN ) printf(" (broken symbolic link)\n" );
        else if ( tflag == FTW_NS )  printf("stat() failed\n" );
    }
    prev_level = cur_level;
    return 0;
}
```

Notice that we don't check for whether the entry is a directory when `prev_level == current_level` ❶. Because of the nature of the post-order traversal by `nftw()`, it cannot be a directory in this case and therefore, we just check how many links it has. When `prev_level > current_level` ❷, it must be a directory and we don't need to check the link count. When `prev_level < current_level` ❸, it might be a directory with no children, or an ordinary file. In this case we check the link count only if it isn't a directory.

The only changes to the main program are the initialization of the hash table and the freeing of the memory it uses. The entire `main()` function is displayed in the following listing, with the changes highlighted in bold.

*main() (for spl_du2.c)*

```
int main(int argc, char *argv[])
{
    int flags = FTW_DEPTH | FTW_PHYS /*| FTW_MOUNT*/;
    int status;
    int i = 1;
```

```
    if ( argc < 2 )  {
        init_hash(&visited, INITIAL_HASH_SIZE);
        memset( total_usage, 0, MAXDEPTH*sizeof(uintmax_t));
        if ( 0 != (status = nftw(".", file_usage, 20, flags) ) )
           fatal_error(status, "nftw");
        free_hash(&visited);
    }
    else
        while (i < argc) {
            init_hash(&visited, INITIAL_HASH_SIZE);
            memset( total_usage, 0, MAXDEPTH*sizeof(uintmax_t));
            if ( 0 != ( status = nftw(argv[i], file_usage, MAXDEPTH, flags)))
                fatal_error(status, "nftw");
            else {
                i++;
                free_hash(&visited);
            }
        }
    exit(EXIT_SUCCESS);
}
```

The `INITIAL_HASH_SIZE` is a macro value. The hash table insertion function is designed to resize the hash table if it gets more than half full, so the choice of initial size is not that important, because it will grow as needed. I made it 1024 for this version of the program.

Also, in this version, the `FTW_MOUNT` flag is commented out. If we want to see whether the program correctly displays disk usage of files with the same inode number but on different filesystems, we have to allow the program to cross mount points. The complete program is named *showdu2.c* and is included in the source code distribution for the book.

We can build the executable and run it on a few directories that contain files with multiple links. For example, we can run it on the *testdir2* directory that started this discussion:

```
$ spl_du2 testdir2
560 testdir2/d2
564 testdir2
```

Only one of the files is counted. If you experiment with this revised version of the program, you'll see that it does not over-count any files with multiple names.

We could add enhancements to this program, such as options to limit the depth of the traversal, or to turn on and off crossing mount points, or to change the display units, but these are not critical. We've achieved the goal of this exercise, to explore how to use the `nftw()` function and to implement a useful command. These enhancements are left as exercises.

### The fts Tree Traversal Functions

The SEE ALSO section of nftw()'s man page mentioned the fts functions. Before we leave the topic of tree traversals, we take a brief look at them.

Unlike nftw(), fts is not a single function, but an integrated set of related functions, in much the same way that opendir(), readdir(), rewinddir(), and closedir() are inter-related functions. The fts functions have their origin in the BSD distributions, starting with 4.4BSD, and they aren't POSIX functions. They are available in most Linux distributions though. The fts set of functions are

```
FTS     *fts_open(char * const *path_argv, int options,
                int (*compar)(const FTSENT **, const FTSENT **));
FTSENT *fts_read(FTS *ftsp);
FTSENT *fts_children(FTS *ftsp, int instr);
int     fts_set(FTS *ftsp, FTSENT *f, int instr);
int     fts_close(FTS *ftsp);
```

Just as opendir() creates a directory stream object and returns a pointer to it, fts_open() creates a *handle* that is used by the other functions. A *handle* is a pointer to an FTS structure. Unlike nftw(), which does not allow the application to control the order in which files are searched other than whether it is pre-order or post-order, fts allows the calling program to specify this order.

The fact that these functions are not part of the POSIX standard implies that any application that uses them may not be portable. On the other hand, we can do much more with fts functions than we could with nftw(), because these functions are much more flexible and have hooks for program data, so that we don't need file-scoped or static variables to store the program's larger state information.

I'll explain the basics of these functions and give a small example to illustrate how they're used. This can be a powerful tool for use when the program you're writing does not need to be portable. We'll start with a summary of the fts functions:

- We call fts_open() first. We pass it an array of strings representing the roots of trees that we want to traverse, an integer that encodes various options, and a comparison function to be used for determining the order in which files are visited. It returns a handle for an FTS structure.
- The fts_read() function visits a file each time it's called. The handle returned by fts_open() is passed to fts_read(). Each file in the tree is visited just once, except for directories, which are visited before and after their children. fts_read() returns a pointer to an FTSENT structure for each file that it visits. The FTSENT structures have a member that allows them to be linked together.
- If the currently visited file is a directory, then then a call to the fts_children() function returns a pointer to a linked list of FTSENT structures representing all of the children in this directory.

- The `fts_set()` function allows a file to be reprocessed after it has been returned by a call to `fts_read()`.
- The `fts_close()` function is the clean-up function. After processing the entire directory tree passed to `fts_open()`, we call `fts_close()` to close the stream and clean up resources.

To use these functions, we need to know the contents of the `FTSENT` structure, because that's what characterizes each visited file. That structure is defined in the header file */usr/include/fts.h*.

```
typedef struct _ftsent {
    unsigned short fts_info;     /* flags for FTSENT structure */
    char          *fts_accpath;  /* access path */
    char          *fts_path;     /* root path */
    short          fts_pathlen;  /* strlen(fts_path) */
    char          *fts_name;     /* filename */
    short          fts_namelen;  /* strlen(fts_name) */
    short          fts_level;    /* depth (-1 to N) */
    int            fts_errno;    /* file errno */
    long           fts_number;   /* local numeric value */
    void          *fts_pointer;  /* local address value */
    struct ftsent *fts_parent;   /* parent directory */
    struct ftsent *fts_link;     /* next file structure */
    struct ftsent *fts_cycle;    /* cycle structure */
    struct stat   *fts_statp;    /* stat(2) information */
} FTSENT;
```

Although this structure has many members, for relatively simple applications, we won't use many of them. A list of the most important members, with brief descriptions, follows.

**`fts_info`** An integer that encodes information about the type of object represented by this structure.

**`fts_accpath`** A path for accessing the file from the current directory.

**`fts_path`** The path for the file relative to the root of the traversal. This path contains the path specified to `fts_open()` as a prefix.

**`fts_name`** The file name.

**`fts_errno`** Upon return of a `FTSENT` structure from the `fts_children()` or `fts_read()` functions, with its `fts_info` field set to `FTS_DNR`, `FTS_ERR` or `FTS_NS`, the `fts_errno` field contains the value of the external variable errno specifying the cause of the error. Otherwise, the contents of the `fts_errno` field are undefined.

**`fts_number`** This field is provided for the use of the application program and is not modified by the `fts` functions. It is initialized to 0.

**`fts_pointer`** This field is provided for the use of the application program and is not modified by the `fts` functions. It is initialized to `NULL`.

**fts_parent** A pointer to the FTSENT structure referencing the file in the hierarchy immediately above the current file, i.e. the directory of which this file is a member. A parent structure for the initial entry point is provided as well, however, only the fts_level, fts_number and fts_pointer fields are guaranteed to be initialized.

**fts_link** Upon return from the fts_children() function, the fts_link field points to the next structure in the NULL-terminated linked list of directory members. Otherwise, the contents of the fts_link field are undefined.

**fts_statp** A pointer to a stat structure for the file.

The fts_info field provides information about the visited file, encoded into an integer value. It contains exactly one of the following values:

**FTS_D** A directory being visited in pre-order.

**FTS_DC** A directory that causes a cycle in the tree. The fts_cycle field of the FTSENT structure will be filled in as well.

**FTS_DEFAULT** Any FTSENT structure that represents a file type not explicitly described by one of the other fts_info values.

**FTS_DNR** A directory that cannot be read. This is an error return, and the fts_errno field will be set to indicate what caused the error.

**FTS_DOT** A dot file that wasn't specified as a file name to fts_open().

**FTS_DP** A directory being visited in post-order.

**FTS_ERR** This is an error return, and the fts_errno field is set to indicate what caused the error.

**FTS_F** A regular file.

**FTS_NS** A file for which no stat information was available. The contents of the fts_statp field are undefined.The fts_errno field will be set to indicate what caused this error.

**FTS_NSOK** A file for which no stat information was requested. The contents of the fts_statp field are undefined.

**FTS_SL** A symbolic link.

**FTS_SLNONE** A symbolic link with a non-existent target. The contents of the fts_statp field reference the file characteristic information for the symbolic link itself.

Let's compare this family of functions to the nftw() function:

- The fts_info member has information similar to that found in the tflag argument to the (*fn) function by nftw(). It characterizes the visited file.
- Unlike the nftw() function, the structure representing the current file, FTSENT, has hooks for the application to use. Specifically, fts_number and fts_pointer can be used for application-specific data, making it

possible to change state and data across different invocations of the `fts_read()` function.

- The `fts_parent` field provides a means to access the parent node, which `ntfw()` does not do.
- The `FTSENT` structure has `stat` information for the returned file through its `fts_statp` member, provided no error occurred.
- The name of the file is in the `fts_name` member. The `fts_path` member has the pathname of the file relative to the root of the tree walk.

Let's look at a small example that illustrates how we can use the `fts_read()` function in a tree walk. The program, which we'll name *fts_demo.c*, displays the sizes, in bytes, or all files in the directory's tree, and after processing all files, it prints out the name and size of the largest file that it found. The comparison function that it passes to the `fts_open()` function compares two entries by name, using the current locale's collating sequence:

```
#include "common_hdrs.h"
#include <fts.h>

int namecmp(const FTSENT **s1, const FTSENT **s2)
{
    return (strcoll((*s1)->fts_name, (*s2)->fts_name));
}
```

The `strcoll()` function was introduced in Chapter 4. It compares two strings based on the current locale.

The main program processes the directory whose pathname is given as the program's first argument. In addition to printing a file's size, it indents each visited file's pathname by a number of spaces proportional to its level in the tree, to make it easy to see the nested directory structure.

*main() for fts_demo.c*

```
int main(int argc, char* argv[])
{
    FTS *tree;
    FTSENT *file;
    char errmssge[128];
    char largest_file[PATH_MAX];
    size_t  max = 0, size;

    if ( argc < 2 ) {
        sprintf(errmssge,"%s directory\n", argv[0]);
        usage_error(errmssge);
    }

    char *dir[] = { argv[1],  NULL };❶
    if ( NULL == (tree = fts_open(dir, FTS_PHYSICAL , namecmp)))
        fatal_error(errno, "fts_open");
```

```
    errno = 0;
    while ((file = fts_read(tree))) {
        switch (file->fts_info)❷ {
        case FTS_DNR:  /* Cannot read directory */
            fprintf(stderr, "Could not read %s\n", file->fts_path);
            continue;
        case FTS_ERR:  /* Miscellaneous error */
            fprintf(stderr, "Error on %s\n", file->fts_path);
            continue;
        case FTS_NS:  /* stat() error: continue to next files. */
            fprintf(stderr, "Could not stat %s\n", file->fts_path);
            continue;
        case FTS_DP: /* Ignore post-order visit to directory. */
            continue;
        }
        /* Check if this is largest file so far. */
        size = file->fts_statp->st_size;❸
        if ( max < size ) {
            max = size;
            strncpy(largest_file, file->fts_path, 1+file->fts_pathlen);
        }
        printf("%12ld\t%*s%s\n"❹, size,
                4*(file->fts_level), " ", file->fts_path);
        errno = 0; /* Set errno to 0 again before next fts_read(). */
    }
    if (errno != 0)
        fatal_error(errno, "fts_read");

    printf("Largest file is %s with size %lu\n", largest_file, max);
    if (fts_close(tree) < 0)
      fatal_error(errno, "fts_close");
    return(EXIT_SUCCESS);
}
```

*Listing 8-9: A program that shows how to use the `fts` functions*

The program starts by passing a `NULL`-terminated array of directory pathnames ❶ and the comparison function to the `fts_open()` function. In this program the array has just a single pathname, `argv[1]`, but in general it can have more. If the open succeeds, the program repeatedly calls `fts_read()`. The order in which files are visited is determined by the comparison function. For each file, it uses the `fts_info` field ❷ to determine if there were errors processing the file, and if not, it gets its size from the `fts_statp` pointer to the file's `stat` structure ❸. If the file is larger than the current largest file, it updates the variables that record this information. Since the `fts_level` field is the level of the file relative to the root of the tree, it indents the file name by `4*fts_level` spaces to emphasize its depth in the tree.

The `printf()`❹ function's format string, `"%12ld\t%*s%s\n"` has something new. The specifier `%*s` expects a number followed by a string. The number is the minimum width of the field in which to print the string. Therefore,

```
printf("%*s%s\n", 4*(file->fts_level), " ", file->fts_path);
```

prints a space in a field of width `4*(file->fts_level)` followed by the string stored in `file->fts_path`.

I built the executable (`fts_demo`) and ran it on the test directory to see how it behaves:

```
$ fts_demo testdir
        4096  testdir
           7      testdir/linktosubdir1
       18893      testdir/newfile
        4096      testdir/subdir1
        4096          testdir/subdir1/subsubdir1
          13              testdir/subdir1/subsubdir1/link2tosubdir1
        4096              testdir/subdir1/subsubdir1/subsubsubdir1
           0                  testdir/subdir1/subsubdir1/subsubsubdir1/testfile1
           0              testdir/subdir1/subsubdir1/testfile1.lnk
        4096          testdir/subdir1/subsubdir2
        4096      testdir/subdir2
Largest file: testdir/newfile; Size=18893
```

Although the program is a simple one, you can see the potential that these `fts` functions have for implementing a variety of applications that need to walk a directory tree. For example, we could implement `du` with it, or a recursive `ls`, or even the more complex `find` command. In fact, the GNU versions of commands such as `grep`, `chmod`, `chown`, `rm`, `cp`, and `chgrp` are implemented with the `fts` functions to perform their recursive tree traversals. On BSD systems, the `find` command is based on the `fts` functions.

## The pwd Command

The `pwd` command prints the absolute pathname of the current working directory. Implementing it will expand our understanding of the structure of directories, but will also present a different set of challenges than we've encountered so far. To see this, we'll work through a small exercise.

### *An Exercise in Constructing a Directory Tree*

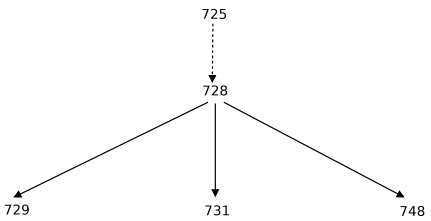
We'll try to reconstruct a portion of a file hierarchy from the inode numbers in a set of directories. Let's suppose that we're given a directory named *scratch* that contains subdirectories and ordinary files and that the subdirectories also have subdirectories, and so on. Each of these subdirectories may have regular files as well. Let's assume for this example that all files are in the same filesystem, so that inode numbers uniquely identify files. The command

```
$ ls -iaR scratch
```

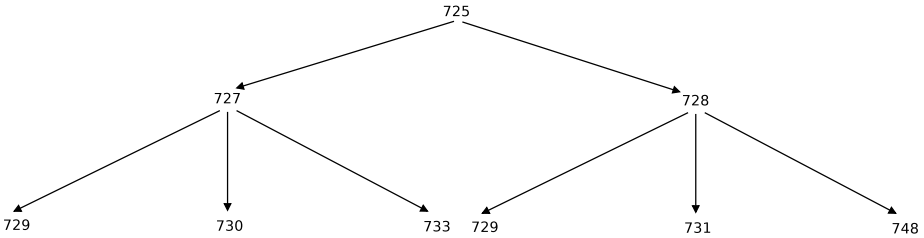
can be used to recursively display the inode numbers and filenames of all files in the directory tree rooted at *scratch*, including the entries for dot and dot-dot in each directory. These dot and dot-dot entries play a critical role in navigating the directories. Suppose that entering **`ls -iaR scratch`** produces the following output, in which the actual directory names have been omitted.

```
725 .
449 ..
753 README
727 work
728 misc

731 .
728 ..
733 TODOLIST
732 tests

727 .
725 ..
729 info
730 prog1
733 TODO

728 .
725 ..
731 tests
748 prog2
729 docs
```

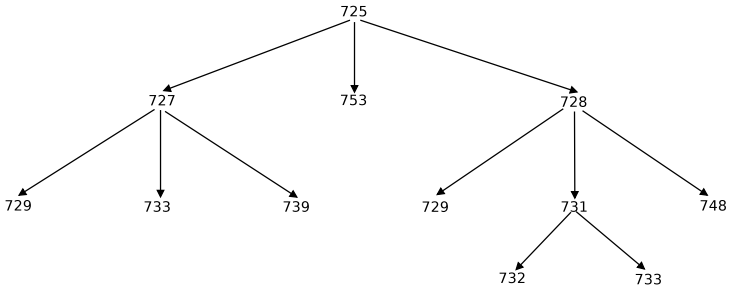
The contents of these directories have enough information to reconstruct the file tree rooted at *scratch*. First, we'll draw a tree whose nodes are just inode numbers, after which we can use the directory listing to assign filenames to those numbers. For example, from the last five lines of output, we see that node 728 has three children: 731, 748, and 729, and that its parent is 725, so our first subtree looks like this:



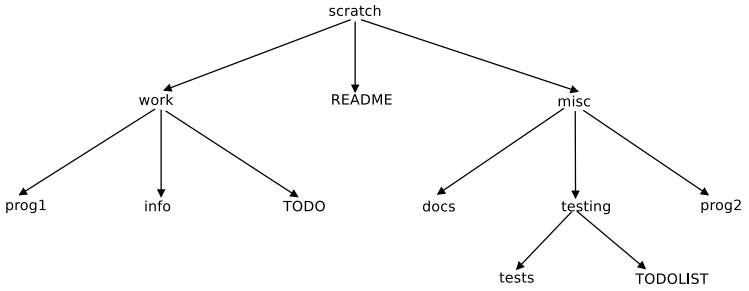
Similarly, the preceding five lines show that node 727 has children with inode numbers 729, 739, and 733, and its parent is 725, which we can use to construct the combined subtree so far:




The lines above the last group show that the inode with number 731 has two child nodes with numbers 732 and 733, and that its parent is 728. This is a subtree of the first tree we constructed. We'll attached this subtree to our growing tree, and simultaneously, since inode 725 has children with numbers 728, 729, and 753, we'll create the new node, 753, and attach it as a child of node 725:




Looking at the resulting tree, we see that two of the inodes, 729 and 733, have links in two different directories; as we replace inode numbers with filenames, we'll use the filenames for those inodes that are contained in the respective directories:



This exercise demonstrates that the parent entries in a given directory are the only means to obtain that directory's name, and that the only way to do this, when all files are in the same filesystem, is to:

1. Save the inode number of the current directory, which is the one associated with the dot entry.
2. Find the inode number for the parent entry in the current directory, meaning the dot-dot entry.
3. Get the list of child inodes of the parent directory.
4. Find the inode number in this list that matches the inode number for dot in the current directory.
5. For the matching inode number, get the filename associated with it. This is the name of the current working directory.

In short, the parent directory entries in a directory play a vital role in the hierarchy, because they're essentially *back links* — they're the only practical way to know the name of the current directory and the only way to ascend the tree.

This strategy will not work in all cases, because it doesn't account for the effect of mount points on this problem. When a part of this directory tree is in a different filesystem because one of the directories is a mount point, it isn't enough to use inode numbers alone. We need to use (inode number, device ID) pairs to represent files. In the preceding sequence of steps, each reference to an inode number must be replaced by a (inode number, device ID) pair. To make this convenient, we'll define a structure that represents such a pair:

```
typedef struct device_inode_pair
{
    dev_t  dev;
    ino_t  ino;
} dev_ino;
```

All preceding references to inode numbers should now be thought of as references to elements of type `dev_ino`.

## A Strategy for Implementing the pwd Command

Suppose our current working directory is *chap_dir_hierarchy*, which is located in the directory *demos*, which is in *lsp_book*, which is in *teaching*, which is in *home/snw*, which is in the root directory, *"/"*. Then entering **`pwd`** will print the absolute pathname:

```
$ pwd
/home/snw/teaching/lsp_book/demos/chap_dir_hierarchy
```

Initially, `pwd` doesn't know where the working directory is with respect to the potentially very large directory tree; it doesn't have the path to it. From the

preceding exercise it should be clear that, to construct this path, it has to work upward, using the parent entries of each new directory as it ascends the tree.

In fact, there's a system call as well as a library function, both named `getcwd()`, that return the absolute pathname of the current working directory. If we wanted to, we could call either of them to solve this problem and we'd be finished, however, since the objective of this exercise is to learn how to climb the tree, that is not an option. Instead, we will try to write those functions from scratch.

The exercise we did gives us part of the strategy for implementing the `pwd` command. We find the name of the current directory using the steps we described on page 410, substituting `dev_ino` pairs for the inode numbers. We then step up to the parent directory and repeat these steps. We do this repeatedly until we've reached the root of the directory hierarchy. This strategy raises several questions:

- How can a program change its working directory to that of its parent?
- How can a program get the list of children of its parent directory?
- How can a program determine which directory entry in the parent matches the child representing the current directory?
- How can we tell when we've reached the root of the tree?
- How do we construct the string that stores the absolute pathname to the directory from right to left, since that's the order in which we'll discover the ancestor directories?
- Do we need to be concerned about the maximum length of a pathname? If so, what would we do if the pathname exceeded it?

To answer the first question, let's see what a man page search will give us:

```
$ apropos -s2,3 -a change directory
chdir (2)            - change working directory
chdir (3posix)       - change working directory
chroot (2)           - change root directory
fchdir (2)           - change working directory
fchdir (3posix)      - change working directory
--snip--
```

The `chdir(2)` and `fchdir(2)` functions share the same man page. The corresponding POSIX pages describe the POSIX requirements for these functions. The synopsis for them is:

```
#include <unistd.h>

int chdir(const char *path);
int fchdir(int fd);
```

The first function changes to the directory specified by the string `path` whereas the second, `fchdir()` changes to the directory specified by the file descriptor, `fd`. The second function requires opening the directory and obtaining its descriptor, whereas, the first doesn't require this. As a system call, the second is faster, but for this program, speed is not an important factor. Both can fail for a variety of reasons, such as not having permission on the directory, or encountering an I/O error. Although unlikely to fail in this case, we still need to error-check the return value. Therefore, changing the working directory to the parent is accomplished with

```
errno = 0;
if ( -1 == chdir("..") )
   fatal_error(errno, "chdir");
```

To answer the second question, we can use any of the methods from earlier in this chapter to open a directory and retrieve its child entries. One choice is to use `opendir()` to open it and then repeatedly call `readdir()` to get its entries until we find the matching inode number. Another choice is to call `scandir()`, passing a filter function designed to select only the single entry whose (inode number, device ID) matches that of the current directory. Performance-wise, it's a toss-up: repeated calls to `readdir()` may be faster in the case that we quickly find the match, but each call adds time, whereas the single call to `scandir()` may be a slower call, and we can't control which order it searches the directory. I'll choose to use repeated calls to `readdir()`.

To determine which directory entry in the parent matches the child, before stepping up to the parent directory, the program can call `stat()` to get the inode number and device ID of `"."`, which is the entry for the current working directory, and store it into a `dev_ino` structure named `dir_dev_ino`. When it's changed the working directory to the parent directory using `chdir("..")`, and it's searching through all of its entries, for each entry returned by `readdir()`, it would call `lstat()` on the `d_name` member of the `dirent` structure to retrieve the inode number and device ID of the entry, storing the pair into a `dev_ino` structure, say named `this_dev_ino`. If `this_dev_ino` and `dir_dev_ino` match then the `d_name` in the `dirent` structure is the name of the directory. It's important that we use `lstat()` rather than `stat()`; the latter reports on the targets of symbolic links and not the links themselves and will lead to errors.

To make the code more readable, I'll define a Boolean-valued function that compares `dev_ino` structures:

```
BOOL  are_samefile(dev_ino f1, dev_ino f2)
{
   return ( f1.ino == f2.ino && f1.dev == f2.dev );
}
```

I'll also define a function, which I've named `get_dev_ino()`, that, given a filename relative to the current directory, gets its inode number and device ID and stores them in a `dev_ino` structure:

```
void  get_dev_ino( const char *fname, dev_ino *dev_inode)
```

```
{
    struct stat sb;
    errno = 0;
    if ( -1 == lstat( fname , &sb )  )
        fatal_error(errno, "Cannot stat ");
    dev_inode->dev  = sb.st_dev;
    dev_inode->ino  = sb.st_ino;
}
```

Assume that `dir_dev_ino` is the `dev_ino` pair for the directory whose name we're trying to find and that the program has changed directory into the parent directory. The code, missing its error checking, would then be roughly:

```
dirp = opendir("."); /* Open parent directory */
while ( ( direntp = readdir( dir_ptr ) ) != NULL ) {
    get_dev_ino(direntp->d_name, &this_entry);
    if ( are_samefile( this_entry, dir_dev_ino) ) {
        /* found the matching entry */
    else
        /* keep searching */
}
```

The next question is how we'll know when we've reached the root. There are two ways. One is to call `stat("/", &sb)` at the start of the program, to get a `stat` structure for the root, extract its inode number and device ID, and store them. Each time we're about to change directory to the parent, we'd compare the inode and device against that of the root. If they match, we know we're at the root. A second method is to use the fact that in the root directory, and only in the root directory, the dot and dot-dot entries point to the same inode. Each time we're about to change directory to the parent, we'd check whether the parent and the current directory have the same `dev_ino` pair. If so, we'd stop, otherwise we'd continue. I'll use the first method, because it's simpler.

The last question is how to build the pathname. Because we're ascending the tree rather than descending it, we're discovering the names of the directories in the pathname in a right to left order. We can't just append the most recently discovered directory name to the pathname because the path would be backwards if we did. Instead we have to *prepend* it to the existing partially-constructed pathname. Figure 8-9 visualizes the steps to prepend the first directory name to an initially empty buffer of size `PATH_MAX` bytes storing the pathname. We've used the `PATH_MAX` macro before; it's the largest allowable size of a pathname on the system, typically 4096 bytes.

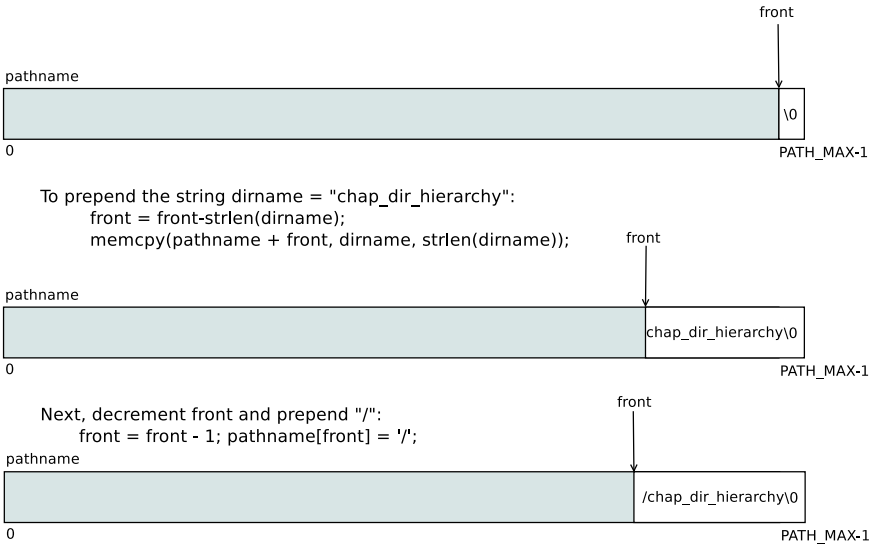



*Figure 8-9: Steps to build the pathname in a right to left direction by prepending each new parent directory*

The first step is to write a `NULL` byte to the rightmost position in the buffer, in `pathname[PATH_MAX-1]`. At all times, `front` will be the index of the leftmost character of the partially-built pathname. Initially, `front = PATH_MAX-1`. For each name that we want to prepend, we get the string length of the name:

```
len = strlength{dirname};
```

and subtract it from `front` so that `front` points to where this name will start in the buffer. We then copy the directory name to this position. The C `memcpy()` function can copy it efficiently. It won't add a `NULL` byte to the end, but we already put it there. Lastly, we prepend a `/` to the string, decrementing `front`. This process is then repeated until the entire pathname has been constructed.

The only problem that this last algorithm does not consider is what will happen if the pathname is larger than the buffer. This can happen if directory names are very long and the working directory is very deep in the directory tree.

The easiest acceptable solution is to write a message that the full path was not constructed and replace the initial part of the pathname by a sequence of ellipsis characters. If we name the executable `spl_pwd`, with our previous example, the output would be something like

```
$ spl_pwd
Error: File name too long
..../teaching/lsp_book/demos/chap_dir_hierarchy
```

It's not that hard to detect when this error would occur. In the listings that follow, which we're ready to assemble, we'll call out the code that handles it.

The first function is the one that searches in the parent directory to find the name of the current directory. When this has been called, the working directory is already the parent.

*get_filename()*

```
char* get_filename( dev_ino child_entry)
{
    DIR           *dir_ptr;     /* The directory to be opened         */
    struct dirent *direntp;     /* The dirent for each entry          */
    dev_ino       this_entry;   /* The dev_ino pair for the entry     */
    char          errmssge[256];/* To store error messages            */
    int           len;          /* Length of a string                 */
    char*         fname;        /* malloc-ed name to return to caller */

    errno = 0;
    dir_ptr = opendir( "." );
    if ( dir_ptr == NULL )
        fatal_error(errno, "opendir" );

    /* Search through the current working directory for a file whose
       inode number and device ID are that of entry.*/
    while ( ( direntp = readdir( dir_ptr ) ) != NULL ) {
        errno = 0;
        get_dev_ino(direntp->d_name, &this_entry);

        /* If this entry matches, we found the file. */
        if ( are_samefile( this_entry, child_entry) ) {
            /* Copy the entry's d_name into a malloc-ed fname. */
            len = strlen(direntp->d_name);
            errno = 0;
            if ( NULL == (fname = malloc(len+1)) )
                fatal_error(errno, "malloc");
            strncpy(fname, direntp->d_name, len);
            closedir( dir_ptr );
            return fname;
        }
    }
    /* If we reach here, there is no matching entry in this directory. */
    sprintf(errmssge, "i-number %lu not found.\n", child_entry.ino);
    error_mssge(-1, errmssge);
    return NULL;
}
```

A call to `get_filename(dir_dev_ino)` in the parent directory of the current working directory returns the name of the current working directory by searching through the entries in the parent using the algorithm we described. If for some reason, no entry is found that matches, an error is reported and a `NULL` pointer is returned instead.

Because the function is returning a pointer to the directory name, that name can't be on the stack. Instead, the function allocates memory for the name on the heap, using `malloc()`. It has to be freed by the caller when it is no longer needed.

The last listing is of the main program. To save space, prototypes for the preceding functions replace their code. The complete program is in the book's source code distribution.

*spl_pwd.c*

```
#include "common_hdrs.h"
#include <sys/stat.h>
#include <dirent.h>

/* The following two functions are in preceding listings */
void  get_dev_ino( const char *fname, dev_ino *dev_inode);
char* get_filename( dev_ino child_entry);

int main(int argc, char* argv[])
{
    dev_ino     current;
    dev_ino     root;
    char        pathname[PATH_MAX];
    char        *dirname;
    ssize_t     front = PATH_MAX-1;
    ssize_t     namelength;

    get_dev_ino("/", &root );
    get_dev_ino( ".", &current);
    pathname[PATH_MAX-1] = '\0';
    while ( !are_samefile(current, root)) {
        /* go up to parent directory */
        chdir( ".." );
        /* Search in the parent directory for the fileame of this_inode. */
        if ( NULL == (dirname = get_filename(current)) )
            fatal_error(-1, "Could not find entry in .. for current directory.");
        /* If successful, write this name to the left of the current path. */
        namelength = strlen(dirname);

        /* Check if the new path is too long. If so, fill with dots instead
           and report the error. */
      ❶ if ( front - namelength <= 0 ) {
            memset(&(pathname[0]), '.', front);
            front = 0;
            error_mssge(ENAMETOOLONG, "Error");
            break;
        }
        else {
            front = front - namelength;
            memcpy(pathname+front, dirname, namelength);
```

```
        }
        /* Free the memory allocated by get_filename() for this string. */
        free(dirname);
        front--;
        pathname[front] = '/';
        get_dev_ino("..", &current); /* To start next level */
    }
    printf("%s\n", &(pathname[front]));
    return 0;
}
```

*Listing 8-10: A program that displays the absolute pathname of the current working directory*

Just before prepending the next component of the pathname, the program checks ❶ whether the buffer would overflow if it did. If so, instead of prepending the component, it puts a string of dots there instead, to indicate that something's missing. It also prints an error message.

We'll build the executable, naming it `showpwd`, and run it and `pwd` in the same directory. I'll pick one whose path crosses two mount points:

```
$ spl_pwd
/data/research_resources/physics/articles/more_articles/quantumstuff
$ pwd
/data/research_resources/physics/articles/more_articles/quantumstuff
```

The *more_articles* directory is the mount point for */dev/sdd4* and *data* is the mount point for */dev/sdb4*:

```
$ df --output=source,target .  /data
Filesystem     Mounted on
/dev/sdd4      /data/research_resources/physics/articles/more_articles
/dev/sdb4      /data
```

which implies that the program crossed both mount points on its way up the tree.

The actual `pwd` command is usually implemented in a more complex way than the one we developed here. The implementations vary from one distribution to another. We could have used the `getcwd()` function to do most of the work. The GNU/Linux implementation avoids calling `getcwd()` because that function fails for pathnames that exceed `PATH_MAX` bytes and their version is designed to be more robust. Their version handles pathnames of umlimited length. Our version fails if it doesn't have permission to open a directory whereas the GNU version displays a more useful diagnostic. Finally, the GNU version will fall back to reading the `PWD` environment variable if things go awry, and we didn't consider that option.

## Summary

The structure of a directory file in Unix is quite simple. A directory is just a sequence of entries, called links, of the form (*inode number, filename*), among which are two entries present even in empty directories for `"."`, called *dot*, and `".."`, called *dot-dot*, which refer respectively to the directory itself and to the parent directory. These two entries provide the means to connect the directories and files into a tree-like structure called the directory hierarchy.

There are a few different methods of processing the contents of directories. One approach uses an API that requires opening a directory using `opendir()`, getting a directory stream as a result, using successive calls to `readdir()` to read the entries in the directory, which are delivered in sequence, and closing the stream with `closedir()`. This API also contains a few other useful functions for saving a position in the directory stream (`telldir()`), returning to a saved position (`seekdir()`), and starting all over from the beginning (`rewinddir()`).

Another method is to use the `scandir()` function, which does not require the directory to be opened and which collects all of its entries into an array that can be accessed after the call returns. This option gives us the ability to filter the entries and also to order them, by passing it a filtering function and/or a comparison function. We developed a few different programs for listing directory contents in the chapter, to demonstrate how both of these methods could be used.

The directory hierarchy is a single tree-like structure whose nodes are directories and files. Even though it isn't technically a tree, it's convenient to call it one. Distinct filesystems can be attached to this single tree by a process called mounting. In mounting, the root of the filesystem to be mounted is attached to an existing directory in the tree, called its mount point. Mounting allows different types of filesystems to be a part of a single hierarchy.

There are many different ways to *walk* through a directory tree, such as a depth-first traversal and a breadth-first traversal. One can do pre-order or post-order processing while walking the tree. The `nftw()` function allows us to walk a tree in various ways, as does the `fts` set of functions. The former is a POSIX standardized tree walk function whereas the latter is a BSD-derived set of functions that may not be present on all systems. These functions can be used to implement a wide range of directory tree tools, such as the `find` command and many others.

In the chapter we showed how to walk a directory tree using a recursive algorithms based upon the `readdir()` and the `scandir()` functions respectively. We also implemented a simple version of the `du` command based upon the `nftw()` function and demonstrated with smaller programs how to use some of the `fts` functions. Lastly, we solved a different problem, that of walking up the tree, to implement the `pwd` command.

## Exercises

1. Modify the *spl_ls1.c* program so that it does not display the `.` and `..` entries and sorts the entries in the collating order of the current locale.
2. Modify the *spl_ls1.c* program so that it omits the `.` and `..` entries and sorts the entries by their times of last modification, with the more recent files preceding the less recent ones.
3. In "A Simple ls Program", we purposely gave the `listdir()` function a `flags` parameter that it didn't need so that it was extensible. With that in mind, write a version of `ls` that accepts one or more of the following options:

```
-l    # Display a listing for each file similar to the real ls
-F    # Add one of the characters */=>@| to the end of the file
              to indicate its type.
-g    # Display each entry's group
```

4. The filter function passed as the third argument to `scandir()` is used to select which entries are copied into the returned array of filenames. Can it be used to limit the entries to those whose filenames match a pattern, such as a file glob? Read the man page for `fnmatch()`. Implement a command `findmatches` that can recursively find all files in a directory tree whose names match the file glob specified on the command line as follows:

```
$ findmatches  fileglob  directory_name
```

   Remember that the filter function has no hooks, so you'll need to use globally scoped variables that it can access.
5. Exercise 3 asks you to use `scandir()`. Solve this same problem using the `fts` set of functions instead.
6. Write a version of `spl_du` based on a recursive algorithm using `scandir()`.
7. Write a version of `spl_du` with an option `-d`*`maxlevel`* that will not process any files whose level in the tree is greater than *`maxlevel`*.
8. The `find` command is a very powerful command. If you read its man page you'll see ow much you can do with it. Write limited version of `find` named `findlinks` that when run as follows

```
$ findlinks  dirpath  pathname&&
```

   searches in the directory tree rooted at *`dirpath`* for all filenames that are links to the same file as *`pathname`* and prints out their pathnames relative to *`dirpath`*.
   with an option `-d`*`maxlevel`* that will not process any files whose level in the tree is greater than *`maxlevel`*.

# 9

# INTRODUCTION TO SIGNALS

In the past few chapters, we implemented various commands as a way to learn different parts of the Unix/Linux API. The programs that we wrote in those chapters were relatively short running programs that had no interaction with users or other programs. Many programs don't behave that way. For example, some must run until a particular event takes place, and others run for a long time and need to respond to events that can happen any time while they're running. Programs such as text editors, games, and servers of various kinds fall into this category. In these types of programs, segments of code are executed only as a result of some external event, such as a user's keypress. Such programs must be designed to deal with signals, and since they're now on our programming horizon, this is a good point at which to cover them.

This chapter differs from the previous few, in which we developed programs similar to existing commands, because it primarily presents and ex-

plores a core concept, much as Chapter 3 did. In particular, it introduces and discusses signals in Unix, including what they are, and how, when, and why they're created and sent to processes. It explains how a process can control when signals are delivered to it and how it can respond when it does receive signals. In the course of presenting these ideas, it introduces the basic system calls and commands related to signals. To make all of this more concrete, we'll work through the development of a few simple programs that demonstrate how we can use these system calls in our programs. There is much more to signals than we cover in this chapter, which covers just the basics.

## The Role of Signals

Signals serve the same purpose in Unix as they do in the outside world; they're a form of notification about some event or condition of importance that is sent to a recipient. In the outside world, signals can be visual, such as the change in color of a traffic light at an intersection, they can be audible, such as the sounding of an alarm clock or smoke alarm, and they can even be mechanical, such as the vibration of a mobile phone. In Unix systems, signals are essentially software interrupts; they're empty messages delivered to a process that interrupt its normal instruction cycle. They're usually sent to report exceptional situations to a process, such as references to invalid memory addresses, or a terminal's being disconnected.

Many signals are like hardware interrupts in that they can occur at any time, independent of what a process is doing when they arrive. The kernel is almost always the source of the signal. Sometimes, under the right conditions, one process can send a signal to another, and it's even possible for a process to send a signal to itself, in which case it is delivered to that process immediately.

Examples of events that cause signals to be sent to a process include a user's entering CTRL-C in the terminal of the process, or resizing the terminal window in which the process is running. These can happen at any time with respect to the process's execution, and therefore we say they're sent *asynchronously*. In contrast, when a process performs an arithmetic instruction that results in a divide-by-zero error, it will also be sent a signal, but this signal will always arrive whenever the process executes that same sequence of instructions. Because the delivery of the signal to the process always happens at the same time in its instruction sequence, this type of signal is said to be a *synchronous* signal.

Let's make this a bit more concrete. Every signal in Unix has a unique integer value that has a (unique) symbolic name. These symbolic names are of the form `SIG` followed by short strings that are descriptive of the purpose of the signal. For example, `SIGINT`, commonly called the *interrupt signal*, is the signal usually sent to a process because the user entered the interrupt character on the keyboard while the process was running, and `SIGWINCH` is the signal sent to it when the window in which the process is running was resized. On most architectures, the numeric value of `SIGINT` is 2, but because

the numeric value of a signal is not standardized, it may vary from one architecture to another. Therefore programs only use their symbolic names. You can see a list of their names in the `signal` man page in Section 7. Later in this chapter I'll present a list of them as well, with more details about them.

A signal has no other information associated with it besides its name. It's like an email message with a subject and no body, the subject being the numeric value of the signal. The value of a signal by itself provides its information; if a process receives a `SIGWINCH` signal, for example, it knows that the terminal window was resized. For this reason, the value of a signal is called the *signal type*.

## A Signal Delivery Example

Let's start with an example that describes the complete sequence of steps that results in a signal being delivered to a process. Entering CTRL-C in a terminal while a process is running usually terminates the process. Let's see how that happens and how signals are involved in it. Figure 9-1 illustrates the sequence we now describe.

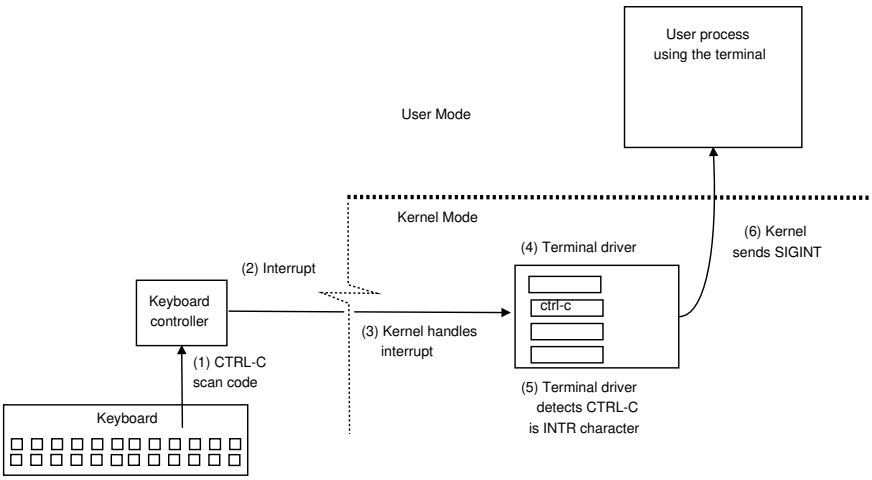



*Figure 9-1: The steps taken to transform CTRL-C entered on a terminal's keyboard to a `SIGINT` delivered to processes using that terminal.*

1. In general, as you type on a keyboard, *scan codes* that represent the entered key combinations are generated by the circuits in the keyboard. When you enter CTRL-C, the *scan code* for that key combination is sent by the keyboard through the port to which it's connected to the keyboard controller.
2. When the controller receives the scan code, it causes a *hardware interrupt*. Every keystroke results a hardware interrupt as you're typing, so that each individual character can be processed.

3. The interrupt causes the kernel to run briefly. The kernel determines with which terminal the keyboard is associated, and transfers control to that terminal's terminal driver.
4. Inside the terminal driver, there's an array of character codes representing the terminal's *special characters*. The special characters are characters that cause actions other than simply being displayed, such as backspace and end-of-file. The terminal driver determines that the entered code matches the code in the entry in this array (at index `VINTR`) whose special meaning is, *send the interrupt signal* (`SIGINT`).
5. The terminal driver checks whether a particular flag (`isig`) is set in the terminal's configuration settings. If `isig` is set, it calls the signal subsystem of the kernel to tell it to send the `SIGINT` signal to all processes whose control terminal is the one that received the `CTRL-C`.
6. The kernel sends `SIGINT` signal to all of those processes.
7. Your process receives the signal. In general, unless a process has specified explicitly what it does when it receives this signal, which I'll explain later, it will terminate upon receiving it, because by default, processes are killed by `SIGINT`. Most programs do not take such explicit steps to *handle* `SIGINT`, and so they terminate abruptly. In this case, your program terminates.

All of these steps take place so fast that it's hard to believe that they all happen in that short amount of time, but they do!

### Sources of Signals

The *source* of a signal is the component of the computer system in which the event or condition occurs, whether it is hardware or software. Regardless of the source of the event, it is only the kernel that sends signals to processes. The kernel is like a central signal processing station inside the machine. It sends a signal to one or more processes if it detects a condition requiring it or if a process issued a system call requesting that a signal be sent. Following is a summary of the different types of sources.

**User** A user can type a key combination that causes the terminal driver to ask the kernel to send a signal. This is an asynchronous signal, since it can arrive at a process at any time, independent of what the process might be doing. Examples include CTRL-C, CTRL-Z, and CTRL-S. The user can also issue the `kill` command, which can send a signal to one or more processes.

**Kernel** Events such as the completion of an I/O operation, the loss of power, a network becoming disconnected, or a timer expiring, are sources of signals that the kernel directly detects. The kernel sends the appropriate signal to the affected processes when these types of events occur. These are asynchronous signals, because they are unpredictable and can arrive at any time with respect to a process's execution.

**Hardware Exceptions** A running process can cause an exception, an error condition, that is trapped by the hardware. These include floating-point exceptions, illegal instructions, addressing exceptions (such as attempts to access addresses outside of the process's address space), and other events generally caused by the process itself. The kernel runs as a result of these traps and sends a signal to the offending process. These are synchronous signals because if the process is run again, they will occur at the same point in the process's execution.

**Other Processes** Processes themselves can request the kernel to send signals to other processes to which it has permission to send signals. For ordinary user processes, these are any processes with the same real and effective user ID. A process can even issue a system call to have a signal sent to itself.

Signals, as a mechanism, were originally designed as a means for a process to be notified of errors and exceptional conditions [15]. They made their first appearance in the early Unix versions. The BSD distributions extended the use of signals so that processes to send signals to each other. However, the most common source of signals is still either the hardware or the kernel.

## Signal Concepts

Here I introduce the terminology and concepts associated with signals in the context of the sequence of events described in the example in the preceding section, "A Signal Delivery Example."

### *The Lifetime of a Signal*

A signal has a lifetime that starts with some event or condition arising either in hardware or software, and ends when it's delivered to the destination process. Some sources state that this causal event *generates* the signal. For example, the POSIX.1-2017 specification states that, "a signal is said to be *generated* for (or sent to) a process or thread when the event that causes the signal first occurs."

From the kernel's point of view, a signal isn't generated until the kernel performs an action to create it. It isn't the occurrence of the event itself that generates the signal, but the action taken by the kernel. In Linux, when the kernel detects an event for which a signal should be sent to one or more processes, or is requested by a process to send a signal to one or more processes, it updates a few data structures associated with each destination process to indicate that a new signal has been sent to that process. Among these data structures is a queue of signals that have been generated for, but not yet delivered to, that process. In Chapter 11, we'll examine these and other data structures associated with a process.

A process that's been sent a signal may not be executing at the time the signal was sent. For example, it might be waiting for an I/O operation to complete, or it may be ready to run but not currently running on a proces-

sor. Until it resumes execution and the signal is actually delivered to it, we say that the signal is *pending* for that process.

**NOTE** *At any given time, there's at most one pending signal of a given type for each process. In other words, the kernel does not generate a signal of that same type for the process if one is already pending; if an event occurs for which it should send that same signal again while it's pending, it just discards it. It's easiest to remember this if you picture the set of pending signals as a set of bits, with one bit for each signal type.*

Processes also have the ability to temporarily *block* certain types of signals by defining a *signal mask*. We'll cover this aspect of signals later in the chapter, in the section "Blocking Signals." If a process has blocked a signal that's been sent to it, the signal remains pending until the process unblocks it.

A signal is *delivered* to a process when it responds to the signal in one of the following ways:

- The process explicitly *ignores* the signal. Some signals can't be ignored though. Even if a process chooses to ignore the signal, it is still considered to be delivered to it.
- The process executes a *signal handler*. A signal handler is a function that the process executes when the signal is delivered. When the process's response is to execute a signal handler, we say that the signal has been *caught*. Signal handling is a large and complex topic that we'll explore in the sections "Basic Signal Handling" and "The sigaction() System Call" later in this chapter.
- The process accepts the *default action* associated with the signal. The default action can be one of the following:

  **Terminate** The process is terminated.

  **Ignore** The process ignores the signal and continues to execute.

  **Stop** The process's execution is *suspended*; it can resume execution at a later time.

  **Core Dump** The kernel writes the contents of a process's logical memory and its context into a *core dump file* and then terminates the process. The core dump can be opened by a debugger such as `gdb` to be inspected.

  **Continue** If the process was stopped, the process can resume execution when it receives certain signals.

  A signal handler is e function of a specific form. A program makes a system call to tell the kernel that this function is to be run when specific signal is sent to the program. This is called *registering* or *installing* the handler. If a program has not registered a signal handler for a particular signal, and doesn't have the explicit instructions that tell the kernel it wants to ignore that signal, its fate is determined by the default action of that signal. Most signals cause a process to terminate by default.

A process's *disposition* of a signal is the action that it takes when the signal is delivered. When you design a program, if you create and register a signal handler for a specific signal, you've set its disposition. By not registering a signal handler, you've also set its disposition to accept the default action.

### Signal Types

Signals first appeared in Fourth Edition Unix in 1973 [**?** ]. Initially, there were nine different signal types, but over the years, the number of signal types increased. The way that signals were sent and delivered also changed over time, since the early methods were unreliable. The BSD systems developed one solution, and System V, another. The first POSIX standard, adopted in 1990, defined a single reliable model of signals with 19 different signals. POSIX.1-2001 added nine more signals, and some Unix distributions added others that aren't standardized. As a consequence, the exact set of signal types varies from one system to another, but those standardized by POSIX are universally found in all Unix systems. There are typically about 30 to 35 different signal types in any Unix system.

The two most important resources for learning how to program with signals are the `signal (7)` man page and the `signal.h(7POSIX)` man page. The `signal.h(7POSIX)` man page contains the POSIX requirements for everything related to signals, including the definitions of all data types, macro constants, functions that work with them, and the signal types themselves. The `signal (7)` man page describes the different types of signals, how a process can send a signal, how signal handlers can be set up, when and how signals are delivered to processes, how processes can temporarily delay their delivery, and much more. Together, these two pages provide almost everything we need to know.

The `signal (7)` man page contains two tables of signals. The first lists the different types of signals, and for each one, which standard introduced it, what the default action is for it, and a brief summary of what causes it. The second table contains the numeric values of the symbolic constants. Because these aren't standardized, for some signals, it lists several values, depending on the processor architecture. Table 9-1 is a nearly complete list of all of the signals listed in the man page, with a few non-standard signals omitted, sorted by their standard numeric value. In the table, the default action *Term* is short for *terminate*, and *Core* is short for *core dump and exit*. A *Yes* in the *POSIX* column means that it's a POSIX standardized signal; a *No* means it isn't a standard signal.

**Table 9-1:** Signal names and default actions

| Name | Number | POSIX | Default Action | Comment |
|---|---|---|---|---|
| SIGHUP | 1 | Yes | Term | Hangup detected on controlling terminal or death of controlling process |
| SIGINT | 2 | Yes | Term | Interrupt from keyboard |
| SIGQUIT | 3 | Yes | Core | Quit from keyboard |
| SIGILL | 4 | Yes | Core | Illegal Instruction |
| SIGTRAP | 5 | Yes | Core | Trace/breakpoint trap |
| SIGABRT | 6 | Yes | Core | Abort signal from abort(3) |
| SIGBUS | 7 | Yes | Core | Bus error (bad memory access) |
| SIGFPE | 8 | Yes | Core | Floating-point exception |
| SIGKILL | 9 | Yes | Term | Kill signal |
| SIGUSR1 | 10 | Yes | Term | User-defined signal 1 |
| SIGSEGV | 11 | Yes | Core | Invalid memory reference |
| SIGUSR2 | 12 | Yes | Term | User-defined signal 2 |
| SIGPIPE | 13 | Yes | Term | Broken pipe: write to pipe with no readers |
| SIGALRM | 14 | Yes | Term | Timer signal from alarm(2) |
| SIGTERM | 15 | Yes | Term | Termination signal |
| SIGCHLD | 17 | Yes | Ignore | Child stopped or terminated |
| SIGCONT | 18 | Yes | Cont | Continue if stopped |
| SIGSTOP | 19 | Yes | Stop | Stop process |
| SIGTSTP | 20 | Yes | Stop | Stop typed at terminal |
| SIGTTIN | 21 | Yes | Stop | Terminal input for background process |
| SIGTTOU | 22 | Yes | Stop | Terminal output for background process |
| SIGURG | 23 | Yes | Ignore | Urgent condition on socket (4.2BSD) |
| SIGXCPU | 24 | Yes | Core | CPU time limit exceeded (4.2BSD); |
| SIGXFSZ | 25 | Yes | Core | File size limit exceeded (4.2BSD); |
| SIGVTALRM | 26 | Yes | Term | Virtual alarm clock (4.2BSD) |
| SIGPROF | 27 | Yes | Term | Profiling timer expired |
| SIGWINCH | 28 | No | Ignore | Window resize signal (4.3BSD, Sun) |
| SIGIO | 29 | No | Term | I/O now possible (4.2BSD) |
| SIGPOLL | 29 | Yes | Term | Pollable event (Sys V); synonym for SIGIO |
| SIGPWR | 30 | No | Term | Power failure (System V) |
| SIGSYS | 31 | Yes | Core | Bad system call (SVR4); |

You can list all signals with their numeric values on your host computer from the command line by entering **`kill -l`**. In bash, you can enter the built-in command, **`trap -l`**, which displays the same output.

Note that the spelling of the symbolic signal names is correct; many correspond to English words, but have missing letters. We can categorize these signals based on their source. We'll describe many, but not all, of the signals listed in Table 9-1 in the following summaries. Because there are circumstances in which we need to know the total number of signals, the *signal.h* header file defines a symbolic constant, NSIG which is the total number of signals defined on the given system. Since signal numbers are assigned consecutively, NSIG equals the largest defined signal number plus one.

### Signals from Program Errors

When a program receives any of the following errors, it should usually terminate. These signals are sent when the program has made a serious enough error that it can't feasibly continue. The purpose of the signal is to give the program a chance to clean up before exiting. For example, it might have changed the state of the terminal window and needs to restore it. After it cleans up, it can safely terminate.

**SIGABRT** A process can call `abort()` to have this signal sent to itself. Since this signal causes a core dump by default, this is how a process can terminate with a core dump.

**SIGSEGV** This is sent when your program causes a segmentation fault, an attempt to access a part of memory for which it doesn't have permission. It indicates an invalid access to valid memory. This can happen when the program dereferences an uninitialized pointer, or when it uses a pointer to step through an array, but doesn't check for the end of the array.

**SIGBUS** This is like a `SIGSEGV` except that it is generated when the the program tries to access an invalid memory address. This causes a *bus error*.

**SIGFPE** This signal reports a fatal arithmetic error of any kind, not just floating-point errors, but errors such as division by zero and overflow.

**SIGILL** The *ILL* is short for *illegal instruction*. This is usually sent when the program tries to execute data, typically because of a bad pointer dereference, or tries to execute a privileged instruction.

**SIGEMT** This name is short for *emulator trap*, the signal sent when an instruction doesn't exist in hardware and must be emulated by software.

**SIGSYS** This signal is sent when the program makes a bad system call, such as by passing the wrong number to the `syscall()` function (see Chapter 3.)

**SIGTRAP** This signal is used to implement debuggers and tracing programs such as `strace` and `ptrace`.

The next class of signals are those that are intended to terminate the program, for one reason or another.

### Termination Signals

These signals are sent, in general, to tell the process to terminate. The intention is to give it a chance to clean up, such as closing open files, saving its state information, restoring the state of the terminal, and so on.

**SIGINT, SIGQUIT** These are the signals that are sent as a result of keyboard input. `SIGINT` is sent when a user enters a CTRL-C. It's a common way to abruptly terminate a process. If a process has a signal handler for it, it can perform clean-up before exiting, or even ignore it. `SIGQUIT` is sent when the user enters CTRL-\, but this can be changed in the ter-

minal settings. Unlike SIGINT, this produces a core dump and terminates the process.

**SIGKILL, SIGSTOP** This are the two signals that will terminate a process without exception. Neither can be ignored, deferred, or caught by a signal handler. The other is SIGKILL. terminates the process whereas SIGSTOP stops it.

**SIGTERM** This signal also terminates a process by default, but unlike SIGKILL, it can be caught by a handler. When you enter the kill command to kill a process, this is the signal that's sent to it.

**SIGHUP** This signal is sent to a process when its controlling terminal is closed or a remote connection is broken. By default it terminates a process. Installing a signal handler for it is a way to do clean-up before the process exits.

#### Timer Expiration Signals

These signals are sent when a timer of some type expires. There are two different types of timers, those that measure real or clock time, and those that measure processor time. We make use of timers when we cover interactive programs in Chapter 15

**SIGALRM** This signal is sent to a process by the kernel when a timer set by either the alarm() or setitimer() system call expires. These timers measure real time.

**SIGVTALRM** This signal is sent to a process by the kernel when a timer that measures how much time the process has spent in the processor, called *virtual time*, expires.

We won't see much use for the SIGVTALRM signal, but we'll find the first to play an important role in game programs.

#### Asynchronous I/O Signals

These signals are related to a type of I/O called *asynchronous I/O*, which we'll cover in Chapter **??**.

**SIGIO** This signal is sent to a process that has arranged in advance to be notified when data is available from a read operation from a terminal device. In Chapter 15, where we cover the design of interactive programs such as games, we'll make use of this signal.

**SIGURG** This signal is related to network programming. It is sent when out-of-band data is received on a *socket*. We'll briefly cover sockets in Chapter 19.

#### Process and Job Control Signals

This category of signals are those that are sent to a process for job control related activities, such as the termination of a child process (Chapter 12) or a request to temporarily stop the process.

**SIGCHLD** In Chapter 12, we'll learn about how one process can create new processes, called *child processes*. This signal is sent to a process that has created one or more children, when one of those child processes terminates.

**SIGTSTP** This is short for *terminal stop*, and is typically sent when the user enters CTRL-Z. It suspends the process, which can be resumed at a later time with the `fg` command.

**SIGCONT** This signal is sent to resume a process that was previously stopped. If a running process receives it, it ignores it by default. It's useful because a signal handler for it can take specific actions when the process resumes.

While a process is suspended, it can't receive any signals until it is continued, except for the `SIGKILL` and `SIGCONT` signals.

#### Miscellaneous Signals

These signals are lumped together but are caused by very different events.

**SIGPWR** This is an asynchronous signal sent when a power loss is detected.

**SIGWINCH** This is sent when the window in which the process is running is resized.

**SIGUSR1,SIGUSR2** These two signals have no pre-defined meaning. They're intended to be used by ordinary user-level programs for synchronization or other notifications. We'll make use of these signals in Chapter 15.

There are a few signals that we didn't describe. It's unlikely you'll ever need to handle them.

#### Signal Definitions

If you're curious to see the definitions of these symbolic names in the header file on your own machine, you'll need to rummage around a bit. The *signal.h* header file includes the definitions of all symbolic signal types indirectly, but does not usually contain them. Typically, the header file <*signal.h*> is just a thin wrapper that includes other header files, including the one that contains the actual signal definitions. In Linux, the files <*bits/signum-generic.h*> and <*bits/signum-arch.h*> together define the signal names. User level programs should never include these; they should just include <*signal.h*>.

Now that we know what the various signals are and what types of events cause them to be sent, we'll consider how we can control what our program does when it receives a given signal. This is called *setting the signal disposition*.

## Basic Signal Handling

As noted earlier, a program doesn't have to accept the default action caused by a signal. It can be designed to respond in its own way to any signal except `SIGKILL` and `SIGSTOP`, which cannot be caught. In short, we can change the dis-

position of our programs with respect to every possible signal except those two. This involves two steps:

1. Defining a function called a *signal handler*, to be executed on receipt of a specific signal. A signal handler has the prototype

```
void  sighandler ( int signum );
```

   in which the integer argument is the number of the signal it has caught. By default, this signal handler is executed, like any other function, on the process's run-time stack, but it's possible to have it run on an alternate stack. There are many limitations on what kind of code can be put into a signal handler; we'll address this in the section "Guidance on Designing Signal Handlers" later in this chapter.

2. Informing the kernel that this function is to be executed when the specific signal is delivered to the process. This is called *registering* or *installing* the signal handler.

Figure 9-2 depicts how the handler is run. When a handler has been registered for a signal and that signal is delivered to a process, the kernel runs briefly, arranges for the signal handler to be run, and returns control to the user process starting inside the handler code.

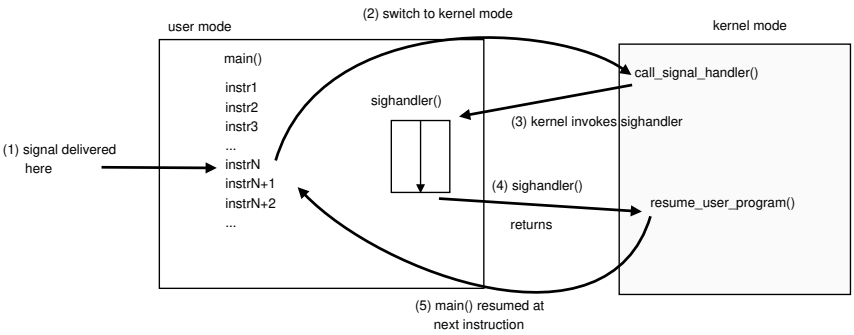



*Figure 9-2: A schematic representation of the steps that occur when a process receives a signal for which it has registered a signal handler named `sighandler()`*

When the signal handler finishes executing, the kernel runs briefly again, this time to ensure that the main program is resumed in the instruction right after the one that was executed just before the signal was delivered. Registering a handler requires making a system call. The original system call designed for this purpose was `signal()`. Although it isn't the preferred way to do this, it is simple to use and easy to understand, and is therefore worth the effort to learn. We'll examine it first.

## The signal() System Call

The `signal()` function was first designed only as a way to notify processes of exceptional events; signals weren't intended as a general purpose notification system [15]. When a signal was delivered to a process and its signal

handler ran, its disposition was reset to the default action, which meant that, if a second signal of the same type arrived, rather than its handler running again, the default action would be taken. If the default action was process termination, the process would die. Programmers worked around this problem by calling `signal()` within the handler to catch the second signal, as in

```
void sig_handler( int sig )
{
    sig_handler( sig );
    /* Take actions in response to signal. */
}
```

but this wasn't a real solution, because the second signal could arrive before the handler was set up again, thereby terminating the process, or after, in which case the handler might be re-entered a second time, like a recursive function call. Because of its unpredictable nature, the `signal()` call and the consequent signal handling were deemed *unreliable*.

Later versions of the `signal()` function in 4.4BSD and in System V corrected this problem in different ways, the consequence being that its semantics depended on which Unix system was being run. POSIX adopted the 4.4BSD model, incorporating a slightly modified specification of it in its first standard, whereas System V continued to use the original semantics. Although current versions of Linux and BSD combine the semantics of each, POSIX and most documentation recommend not using this function anymore. It has been replaced by a reliable method of signal handling based on the `sigaction()` system call, which we'll examine in the section "The sigaction() System Call" later in this chapter.

The `signal()` function's man page is in Section 2. Its prototype, from the `SYNOPSIS`, is:

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Programs must include *signal.h* to use this function. The synopsis contains a declaration of the `sighandler_t` type, which is a pointer to a function whose prototype is `void` *`sighandler`*`(int signum)`. This is both the return type of `signal()` as well as the type of its second argument.

On success, the return value of `signal()` is the old disposition of the signal passed to it in its first argument, which is the number of the signal to be handled. For this argument, it's best to use its symbolic name, such as `SIGINT` or `SIGQUIT`. rather than an actual number. The second argument is the disposition we want this signal to have. It need not be the address of a signal handler function; it can also be either of the two constants, `SIG_DFL` and `SIG_IGN`. Both of these are defined indirectly in *signal.h* as *fake* signal functions, along with the return value `SIG_ERR`, which indicates an error:

```
/* Fake signal functions.  */
```

```
#define SIG_ERR  ((__sighandler_t) -1) /* Error return   */
#define SIG_DFL  ((__sighandler_t)  0) /* Default action */
#define SIG_IGN  ((__sighandler_t)  1) /* Ignore signal  */
```

If SIG_DFL is supplied, the default action will be taken, and if SIG_IGN, the signal will be ignored. If instead, it's the name of a signal handler, then that handler will be invoked. Let's take a look at a simple example that uses signal().

The following program, *signal_demo1.c*, shows how a program can install signal handlers to catch the SIGINT and SIGQUIT signals generated when the user enters CTRL-C and CTRL-\ respectively. When compiling on Linux, using the typical default compiler options, signal() will have the BSD-style semantics in which the disposition is not reset to the default action when the signal is delivered. It's as if _BSD_SOURCE were defined when compiling.

*signal_demo1.c*

```
#include "common_hdrs.h"
#include <signal.h>

void catch_sigint(int signum)
{
    printf("I'm not terminated by CTRL-C!\n");
}

void catch_sigquit(int signum)
{
    printf("I'm not terminated by CTRL-\\!\n");
}

int main()
{
    int i;
    if ( SIG_ERR == signal( SIGINT,  catch_sigint) )
        fatal_error(errno, "signal()");
    if ( SIG_ERR == signal( SIGQUIT, catch_sigquit) )
        fatal_error(errno, "signal()");
    for(i = 20; i > 0; i-- ) {
        printf("Try to terminate me with ^C or ^\\.\n");
        sleep(1);
    }
    return 0;
}
```

*Listing 9-1: A simple program that catches SIGINT and SIGQUIT*

The two functions, catch_sigint() and catch_sigquit(), are signal handlers for SIGINT and SIGQUIT respectively. Observe that their prototypes match the definition of sighandler_t. In the main() function, the two calls to signal() install catch_sigint() as the signal handler for SIGINT and catch_sigquit() as the signal handler for SIGQUIT. Until signal() is executed, the program is subject to the default action for each. Like previous programs we've written, this one

checks for an error from the system call and exits if it's detected. First let's compile the program with the command

```
$ gcc -o signal_demo1 -I../include signal_demo1.c -L../lib -lspl
```

When we run `signal_demo1` and enter **CTRL-C** in the same terminal, the `SIGINT` signal is sent to the process executing `signal_demo1`; as a result, `catch_sigint()` runs, and when it returns, the program resumes execution. In *signal_demo1.c*, the only action taken by either handler is to print a message on the screen, simply to show that the function was executed.

**NOTE** *Normally, we shouldn't call `printf()` in a signal handler because it isn't an* async-signal-safe *function. Some functions are not safe to call in signal handlers; those that can be called safely are async-signal-safe. Because the point of many of the small programs presented here is to show when handlers are called, I put calls to `printf()` in them; otherwise I'd avoid it. I discuss this issue in more detail in "Guidance on Designing Signal Handlers" and describe how a handler can safely print messages.*

You can enter CTRL-C many times and each time the program will just print a message; the disposition is not reset. This is the BSD-style semantics.

### The System V signal() Semantics

On some systems, if you enter CTRL-C a second time, the program will terminate, because signalling is based on the System V Interface Definition (SVID) model, in which the disposition is reset to the default action when the signal is delivered. As I mentioned before, with the default compiler options in Linux, this won't happen, because the version of `signal()` that's called doesn't reset the disposition after the handler runs. However, we can change the behavior of `signal_demo1` by defining the macro `_XOPEN_SOURCE` when we compile it. Doing so exposes a different version of the `signal()` function with the semantics defined in the SVID. If we compile with the command

```
$ gcc -D_XOPEN_SOURCE -o signal_demo1 -I../include signal_demo1.c -L../lib -lspl
```

and rerun the program, we'll see different behavior, as the following run shows. The `^C` is what appears on the terminal when we enter CTRL-C.

```
$ signal_demo1
Try to terminate me with ^C or ^\.
^CI'm not terminated by CTRL-C!
Try to terminate me with ^C or ^\.
^C
$
```

The first CTRL-C is caught but the second terminates the program. Linux provides an explicit way to obtain the SVID behavior with the `sysv_signal()` system call. We use it in the exact same way as `signal()`, but we need to define `_GNU_SOURCE` to employ it, since it's a GNU extension.

*sysv_signal_demo.c*

```
#define _GNU_SOURCE
#include  "common_hdrs.h"
#include  <signal.h>

void catch_sigint(int signum)
{    /* REMINDER: printf() is not safe in the signal handler. */
    printf("I'm not terminated by the first CTRL-C!\n");
}

void catch_sigquit(int signum)
{    /* REMINDER: printf() is not safe in the signal handler. */
    printf("I'm not terminated by the first CTRL-\\!\n");
}

int main()
{
    int i;
    if ( SIG_ERR == sysv_signal( SIGINT,  catch_sigint ) )
        fatal_error(errno, "sysv_signal()");
    if ( SIG_ERR == sysv_signal( SIGQUIT, catch_sigquit ) )
        fatal_error(errno, "sysv_signal()");
    for (i = 20; i > 0; i-- ) {
        printf("Try to terminate me with ^C or ^\\.\n");
        sleep(1);
    }
    return (0);
}
```

*Listing 9-2: A program that catches SIGINT and SIGQUIT using unreliable signalling*

When you compile and run this program and enter CTRL-C or CTRL-\ more than once, the program terminates. The point of these examples is to witness how this unreliable signalling mechanism behaves. Unless I state otherwise, the executables of all remaining programs that use the `signal()` system call for setting the disposition are assumed to be built using the default, BSD-style, semantics.

The next program, *signal_demo2.c*, is almost the same as *signal_demo1.c* with one exception: the dispositions of `SIGINT` and `SIGQUIT` are set to ignore them by the calls to `signal()` with `SIG_IGN` as the second argument.

*signal_demo2.c*

```
#include  "common_hdrs.h"
#include  <signal.h>

int main()
{
    if ( SIG_ERR == signal( SIGINT,  SIG_IGN ))   /* ignore Ctrl-C */
        fatal_error(errno, "signal()");
    if ( SIG_ERR == signal( SIGQUIT, SIG_IGN ))  /* ignore Ctrl-\ */
```

```
        fatal_error(errno, "signal()");
    for( i = 10; i > 0; i-- ) {
        printf("Try to kill me with ^C or ^\\. "
               "Seconds remaining: %2d\n", i);
        sleep(1);
    }
    return 0;
}
```

*Listing 9-3: A simple program that sets the dispositions of `SIGINT` and `SIGQUIT` to be ignored*

When you run this program and enter CTRL-C and CTRL-\ repeatedly, nothing happens. The program runs as if you didn't enter any keyboard input. The program calls `sleep()`, a system call that suspends the calling process for the given number of seconds. Because the process spends almost all of its time suspended in the call to `sleep()`, the signals are most likely being sent to the process while it's suspended in that call. When a signal is sent to a process that's not running, the kernel records that signal's arrival and marks it as pending, if it wasn't already marked as pending. The process is then scheduled to run. As soon as it runs, its signal handler is executed. When it returns, the interrupted function, in this case the `sleep()` call, may or may not be restarted, depending on the particular Unix distribution on which it's run. In Linux, the call is not restarted; it's one of many functions that aren't restarted when they're interrupted by a signal handler's execution. This implies that in our example program, at most one signal can be sent to the process during its `sleep()` call. The `signal (7)` man page has a section detailing the responses of system calls to interruptions by signal handlers.

Before we explore some of the more advanced topics, including the `sigaction()` based method of setting the signal disposition, let's find some user-level commands that we can run for sending signals to processes and some system calls and library functions that programs can call to do the same, so that we have a way to test out some of the signal handling programs that we write.

## Sending Signals

Let's search in the man pages for commands that we can use to send signals:

```
$ apropos -s1 -a send signal
kill (1)             - send a signal to a process
skill (1)            - send a signal or report process status
snice (1)            - send a signal or report process status
```

Among these is the `kill` command. We've used the `kill` command before to list signal types; `kill -l` displays the list of all signals. In spite of its name, the `kill` command sends signals. If we enter

```
$ kill pid
```

where ***pid*** is the process ID of a process, it will send the SIGTERM signal to that process. For example, **kill 1234** sends the signal to the process whose PID is 1234. If we replace the PID with -1, as in

```
$ kill -1
```

then the SIGTERM signal will be sent to every process that we're allowed to kill, which, since we are non-privileged users, are, roughly speaking, all of our currently running processes, but not those of other users. By default, kill sends the SIGTERM signal, but it can send any other signal as well.

The more general form of the command is

```
$ kill -s signal-val <pid> [<pid>] ...
```

where *signal-val* can be one of the following:

- The full signal macro name, such as SIGKILL.
- The part of the signal macro name after SIG, such as KILL.
- The numeric value of the symbol, such as, for SIGKILL, 9.

The kill command requires that you know the PID of the process you want to signal. You can always get it using ps, but there's a faster method mentioned in the SEE ALSO section of kill's man page, namely the pkill command, which doesn't need the PID. The pkill command's syntax is essentially the same as kill's, but you can give it a pattern to match the command that you'd like to signal rather than its PID, and it will send the signal to all commands that match the pattern. For example,

```
$ pkill signal_demo
```

will send the SIGTERM signal to any process that was run by entering **signal_demo...**, such as signal_demo1 or signal_demo2.

Let's turn to the programming interface. We'll search the man pages in Sections 2 and 3 for system calls or library functions for sending signals:

```
$ apropos -s2,3 -a send signal
kill (2)              - send signal to a process
kill (3posix)         - send a signal to a process or a group of processes
killpg (3)            - send signal to a process group
killpg (3posix)       - send a signal to a process group
pidfd_send_signal (2) - send a signal to a process specified by a file desc...
pthread_kill (3)      - send a signal to a thread
pthread_kill (3posix) - send a signal to a thread
raise (3)             - send a signal to the caller
raise (3posix)        - send a signal to the executing process
tgkill (2)            - send a signal to a thread
tkill (2)             - send a signal to a thread
```

This list includes functions that send signals to threads. Ignoring them for now, we see that there are a few functions for sending signals to either a single process or to a process group. Process groups are covered in Chapter 11. Let's look at the man page for the `kill()` system call.

This call allows a process to send a signal to one or more processes or even itself. Its synopsis is

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

int kill(pid_t pid, int sig);
```

The first parameter can be used to specify the PID of the process to receive the signal. The second parameter is the type of signal to send. In the simplest case, a call such as

```
kill(942, SIGTERM);
```

sends the `SIGTERM` signal to the process whose PID is 942. One process cannot send a signal to another, target process if the sender's real or effective user ID doesn't equal either the real or saved set-user-ID of the target. If a process doesn't have permission to send a signal to the specified process, `kill()` returns -1.

The first argument can also be 0, -1, or another negative number, and it means something different in each case:

**0** The signal will be sent to all processes in the same process group.

**-1** If the sender is not the superuser, it's sent to all processes for which it has permission to send signals, which are all those processes whose real or saved set-user-ID is the same as the real or effective user ID of the sending process.

***n* < -1** It's sent to all processes in the process group whose process group ID is the absolute value of *n*. When we run a command such as

```
$ last | grep pts/0 | sort | uniq
```

a process is created for each separate program in the command, and all of those are placed into a single process group. Being able to send a signal to all of the processes involved in the command with a single call is convenient.

One application of the `kill()` system call is to enable related processes that have created new processes to coordinate and synchronize their behavior with these *child* processes. We'll explore how to write programs that can create new processes in Chapter 12. Here we'll use an artificial example to show how `kill()` works. We begin by writing a small program that sends two consecutive signals to the process whose PID is passed to the program on the command line. That program is displayed in Listing 9-4.

*kill_demo.c*

```
#include  "common_hdrs.h"
```

```
#include  <signal.h>

int main(int argc, char* argv[])
{
    int res, pid;
    char message[128];
    if ( argc < 2 )
        usage_error("kill_demo <PID of a process to signal>");

    if ( VALID_NUMBER != (res = get_int(argv[1], NO_TRAILING, &pid, message)))
        fatal_error(res, message);

    printf("Sending SIGINT to  process %d.\n", pid);
    if ( -1 == kill(pid, SIGINT))
        fatal_error(errno, "kill() sending SIGINT");
    sleep(1); /* Give a chance for signal to be sent. */
    printf("Now sending SIGTERM to  process %d.\n", pid);
    if ( -1 == kill(pid, SIGTERM))
         fatal_error(errno, "kill() sending SIGTERM");
    return 0;
}
```

*Listing 9-4: A program that sends a SIGINT and then a SIGTERM to the process whose PID is given on its command line*

The program sends a SIGINT followed by a SIGTERM, the idea being that the target process is designed to catch the SIGINT but not the SIGTERM, and will terminate after receiving it.

We then demonstrate how it works by performing the following steps:

1. We write a small program named signal_demo3, displayed in Listing 9-5, with a handler only for SIGINT, that prints its PID when it starts up, produces no input prompts, and runs until it receives any other terminating signal such as SIGTERM, so that we have a chance to send it the signals from a different terminal. Printing the PID allows us to record the PID and pass it to kill_demo.
2. We run signal_demo3 in the background and record the PID that it displays.
3. While signal_demo3 is running, we run kill_demo, giving it the PID displayed by signal_demo3.

*signal_demo3.c*

```
#include  "common_hdrs.h"
#include  <signal.h>

static char* progname;

void catch_sigint(int signum)
{    /* REMINDER: printf() is not safe here. */
```

```
    printf("%s caught CTRL-C!\n", progname);
}

int main(int argc, char* argv[])
{
    progname = argv[0];
    printf("PID=%d\n", getpid());
    if ( SIG_ERR == signal( SIGINT,  catch_sigint ) )
        fatal_error(errno, "signal()");
    while (TRUE) continue; /* Wait for a signal to be received. */
    return 0;
}
```

*Listing 9-5: A small program that catches SIGINT and no other signal and runs idly until it is sent a terminating signal*

When we perform this small experiment, we'll see that `kill_demo` sent the two signals to `signal_demo3`, because `signal_demo3` prints a message when it receives `SIGINT` and it terminates when it receives the `SIGTERM`:

```
$ signal_demo3 &
[1] 18268
PID=18268
$ kill_demo 18268
kill_demo sending SIGINT to  process 18268.
signal_demo3 caught CTRL-C!
kill_demo sending SIGTERM to terminate process 18268.
[1]+  Terminated              signal_demo3
```

If we try to pass the PID of a process that isn't our own to `kill_demo`, we'll see a message like

```
kill() sending SIGINT: Operation not permitted
```

as proof that a process can send signals only to those processes whose real or saved set-user-IDs are the same as either the real or the effective user ID of the process.

A process can send a signal to itself using the `raise()` library function:

```
int raise( int signal);
```

which returns 0 on success and -1 on failure. Since the only possible error is passing a bad signal number, I don't check the return value in any of the programs in the listings. If the process has installed a handler for the signal that it sends to itself, the handler will run and, only after it terminates, will `raise()` return from the call. A process can also send a signal to itself by calling `getpid()` and using `kill()`, as in

```
kill(getpid(), signal);
```

Why would a process ever need to send a signal to itself? Here's one common reason. Suppose that your program has modified the terminal settings, or created temporary files, or taken other actions that might require immediate cleaning up if a user tries to stop or terminate it. Proper behavior would be to perform all clean-up and then terminate or stop, depending on the signal sent by the user. Therefore, in the handler for a job control signal, it can raise a signal to terminate the program after it performs the cleanup. Here's a small program, based on an example in the GNU C Library manual, that shows how to do this:

*raise_demo.c*

```
#include  <stdio.h>
#include  <unistd.h>
#include  <signal.h>

void sigtstp_handler(int signum)
{
    if ( SIG_ERR == signal( SIGTSTP, SIG_DFL ) )
        fatal_error(errno, "signal()");
    printf("\ncleaning up in progress...\ndone\n");
    raise( SIGTSTP );
    printf("raise() called to stop process.\n");
}

void sigcont_handler(int signum)
{
    /* When the process is resumed, reset the sigtstp handler so that it
       cleans up again before stopping. */
    if ( SIG_ERR == signal(SIGTSTP, sigtstp_handler) )
        fatal_error(errno, "signal()");
}

int main()
{
    int i;
    if ( SIG_ERR == signal( SIGTSTP, sigtstp_handler ) )
        fatal_error(errno, "signal()");
    if ( SIG_ERR == signal( SIGCONT, sigcont_handler ) )
        fatal_error(errno, "signal()");
    for( i = 20; i > 0; i-- ) {
        printf("Enter CTRL-Z to stop the process, or CTRL-C to end it.\n");
        sleep(2);
    }
    return 0;
}
```

*Listing 9-6: A program that shows how to clean-up when receiving a job control signal and then comply with the signal's default action*

The main program installs handlers for SIGTSTP and SIGCONT. The signal handler for SIGTSTP first sets the disposition back to the default, which is to stop the process, then performs all clean-up, and then calls raise(SIGTSTP) to send itself the signal and force itself to stop. Once the program has stopped, if the user resumes it by entering fg on the command line, the sigcont_handler() will run, setting the disposition of SIGTSTP back to calling sigtstp_handler(), so that it can clean up again if it's stopped another time. We'll compile and run this program, and enter a CTRL-Z, then resume it after it stops and enter a CTRL-C:

```
$ raise_demo
Enter CTRL-Z to stop the process, or CTRL-C to end it.
^Z
cleaning up in progress...
done
raise() called to stop process.

[2]+  Stopped                 raise_demo1
$ fg
raise_demo
Enter CTRL-Z to stop the process, or CTRL-C to end it.
^C
$
```

You should try this a few times to convince yourself that it's doing what I've just described, each time entering CTRL-\ a few times before terminating it with the CTRL-C.

## Blocking Signals

Another way in which we have control over how our programs can respond to signals is by *blocking* them. Blocking a signal means informing the kernel to hold onto that signal for a short time until we're ready for it to be delivered. If the kernel generates a signal that a process has blocked, the kernel won't send it to the process until it unblocks it. Blocking a signal is best viewed as putting a short term hold on its delivery while our program performs some actions that we don't want to be interrupted. If we wanted it to be blocked for a long time, it would be better to just set its disposition to SIG_IGN. There are various circumstances under which short term blocking is useful:

- A program might be executing a *critical region* of code, which is a short section of code that updates shared variables or data structures. In particular, if the signal handler for a given signal or the main program modifies some variable that they both modify, then we don't want the program to receive that signal while the main program is modifying that data, otherwise the signal handler will run

and possibly corrupt the data that the main program was updating. Therefore, we'd block it before and unblock it after the access.

- The program might need to execute some code before a particular signal has been delivered to it. It could block that signal until it executes the code and unblock it when it's done.
- When a program is in the midst of handling one signal, it may need to block the delivery of other signals until it finishes what it's doing.

The only two signals that cannot be blocked are `SIGKILL` and `SIGSTOP`. It isn't an error to try to block them; the attempt will just be ignored.

**NOTE** *Blocking a signal is not the same as ignoring it. An ignored signal is actually delivered to a process, which then ignores it, whereas a blocked signal is not delivered to the process, but might be later, when it may no longer be ignored.*

The kernel manages blocking of signals by maintaining a *signal mask* for every process. When we study threads in Chapter 13, we'll see that a signal mask is also maintained for every thread of a multi-threaded process. The signal mask is the set of signals that are currently blocked for that process (or thread.) We can think of it as a bit mask with a bit for every signal type; a signal is blocked if and only if its corresponding bit in the mask is set, as depicted in Figure 9-3. The mask may not be implemented like this, but it's an easy way to conceptualize it.

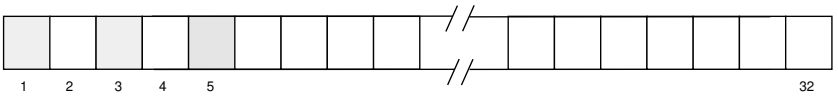



*Figure 9-3: A conceptualization of the signal mask for a process in which shaded cells represent blocked signals*

Let's search the man pages to try to find functions related to the blocking and unblocking of signals:

```
$ apropos -s2,3 -a signal block
--snip--
rt_sigprocmask (2)   - examine and change blocked signals
sigblock (3)         - BSD signal API
sigpause (3)         - atomically release blocked signals and wait for inte...
sigprocmask (2)      - examine and change blocked signals
sigprocmask (3posix) - examine and change blocked signals
```

Reading their man pages we learn that the `sigblock()` function is obsolete and `sigpause()`'s man page tells us not to use it. The preferred function is `sigprocmask()`, which is a system call described in the same man page as `rt_sigprocmask()`. The `sigprocmask()` function can be used to block or unblock one or more signals anywhere in a program. It isn't the only means of blocking signals; when we examine the `sigaction()` system call, we'll see that we can use it to specify which signals are blocked during execution of an installed signal handler.

The synopsis for sigprocmask() is

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

where the first parameter (how) takes one of three symbolic integer values: SIG_BLOCK, SIG_UNBLOCK, or SIG_SETMASK. We'll explain them shortly. On success, this call returns 0 and on failure, -1. The remaining two parameters are of type sigset_t. The sigprocmask() man page suggests reading the Section 3 sigsetops man page for more information about it.

That page presents the general concept of signal sets as well as a collection of functions that act on them.

### Signal Sets

A *signal set* is a data structure that specifies a set of signals. The system data type sigset_t represents a signal set. The sigsetops man page lists five functions for working with signal sets:

**int sigemptyset(sigset_t *set)** This initializes the parameter (set) to be empty, or in other words, to exclude every defined signal type. It always returns 0.

**int sigfillset(sigset_t *set)** This initializes the parameter (set) to be full, or in other words, to include every defined signal type. It always returns 0.

**sigaddset(sigset_t *set, int signum)** Given a particular signal (signum), this adds that signal type to the given signal set, set. It returns 0 on success, and -1 on failure. The only possible failure occurs if signum isn't a valid signal number.

**sigdelset(sigset_t *set, int signum)** Given a particular signal (signum), this deletes that signal type from the given signal set, set. It returns 0 on success, and -1 on failure. The only possible failure occurs if signum isn't a valid signal number.

**int sigismember(const sigset_t *set, int signum)** This function tests whether signum is a member of the signal set set. It returns 1 if the signal is in the set, 0 if not, and -1 on error, which only occurs if signum isn't a valid signal number.

The first two functions create empty and full signal sets respectively. The next two add or delete individual signals from the specified sets. This gives us two ways to build a set of signals. We can either create an empty set and add signals to it, or create a full set and delete from it. If we want a set with fewer than half of the possible signals in it, then it makes sense to do the former, otherwise the latter.

Although the sigset_t data type may be implemented as a bit mask, we can't count on that and we need to use these functions exclusively for defining sets of signals.

The sigsetops man page also lists three other support functions in *signal.h* that can be exposed by defining the _GNU_SOURCE feature test macro:

```
int sigisemptyset(const sigset_t *set);
int sigorset(sigset_t *dest, const sigset_t *left, const sigset_t *right);
int sigandset(sigset_t *dest, const sigset_t *left, const sigset_t *right);
```

The first is useful for determining whether a signal set is empty, returning 1 if the set has no signals, and 0 if it does. The next two fill the signal set dest respectively with the union of their last two arguments (sigorset()), or their intersection (sigandset()). These are not POSIX functions, implying that if we use them, our programs may not be portable.

### The sigprocmask() Function

Now that we know how to create and modify signal sets, we can return to the sigprocmask() function. Its first parameter (how) should be one of the following three values:

**SIG_BLOCK** With this value, the signal set passed as the second parameter (set) will be added to the set of currently blocked signals in the signal mask. In effect, the new mask is the union of the old mask and the supplied set. If a signal is already blocked and is part of the set, it has no effect.

**SIG_UNBLOCK** With this value, the signal set passed as the second parameter (set) will be subtracted from the set of currently blocked signals in the signal mask. If a signal in set is not currently blocked, it has no effect.

**SIG_SETMASK** With this value, the signal set passed as the second parameter (set) replaces the entire signal mask by the signals in set, effectively ignoring the old mask.

The last parameter is the old value of the mask. If we don't want to use it later in the program, we can just pass a NULL to it; otherwise we'd pass the address of a signal set in which to save it. A common reason for saving it is being able to restore the previous mask after temporarily change the mask of blocked signals.

Let's look at a code fragment that illustrates the general paradigm. The following code snippet blocks delivery of SIGINT during a section of code.

```
sigset_t blocked_signals, old_mask;
--snip--
sigemptyset(&blocked_signals);
sigaddset(&blocked_signals, SIGINT);

/* Add SIGINT to set of blocked signals in mask. */
if ( -1 == sigprocmask(SIG_BLOCK, &blocked_signals, &old_mask) )
    fatal_error(errno, "Error trying to change signal mask");
```

```
/* Do critical work here, and then unblock the signal. */

❶ if ( -1 == sigprocmask(SIG_SETMASK, &old_mask, NULL) )
    fatal_error(errno, "Error trying to restore old mask");
```

This code prevents delivery of `SIGINT` while it's executing critical code. Other signals that aren't currently blocked can still be delivered, and this fragment doesn't show their dispositions. The unblocking method ❶ used in this code fragment does not necessarily unblock `SIGINT`. If `SIGINT` had been blocked before this code fragment ran, it would still be blocked afterward, because when we restore the old mask using `SIG_SETMASK`, we're replacing the entire current mask with the old one, and if it was blocked in the old mask, it will still be blocked. If the intent is to unblock `SIGINT`, it's better to explicitly do so, with the call

```
sigprocmask(SIG_UNBLOCK, &blocked_signals, NULL);
```

which removes it from the mask without blocking or unblocking any other signals.

This raises another question, which is how we can tell whether one or more signals are currently blocked. If a program calls `sigprocmask()` with `NULL` as its second argument, as in

```
if ( -1 != sigprocmask(SIG_BLOCK, NULL, &old_mask) )
    fatal_error(errno, "Error calling sigprocmask()");
```

then the mask is unaffected but its current state is saved in `old_mask`. To test whether a particular signal, say `SIGINT`, is in the returned mask, we can write

```
if ( sigismember(&old_mask, SIGINT) )
    printf("SIGINT is currently blocked.");
```

We'll put some of these ideas together to write a small program named *sigprocmask_demo1.c* that blocks `SIGINT` while the program sleeps a bit, then unblocks it and sleeps intermittently again. It introduces the use of a new system call, `usleep()`, which has finer granularity than `sleep()`. This call suspends the process for the given number of *microseconds* rather than seconds. Therefore, calling `usleep(5000)` suspends the process for 5000 microseconds, which is 5 milliseconds, and `usleep(500000)`, for 0.5 seconds.

*sigprocmask_demo1.c*

```
#include  "common_hdrs.h"
#include  "common_hdrs.h"
#include  <signal.h>

void catch_sigint( int signum )     /* Signal handler for SIGINT */
{
    printf("  Caught SIGINT\n");   /* UNSAFE */
}

int main(int argc, char* argv[])
```

```
{
    int i;
    sigset_t blocked_set;

    if ( SIG_ERR == signal(SIGINT, catch_sigint) )
        fatal_error(errno, "signal()");
    /* Create a signal set with just SIGINT, and block with it. */
    sigemptyset(&blocked_set);
    sigaddset(&blocked_set, SIGINT);
    if ( -1 == sigprocmask(SIG_BLOCK, &blocked_set, NULL) )
        fatal_error(errno, "sigprocmask()");

    printf("SIGINT is blocked; sleeping for 5 seconds."
           " Try entering a few CTRL-Cs.\n");
    for ( i = 1; i <= 1000; i++ )
        usleep(5000);

    if ( -1 == sigprocmask(SIG_UNBLOCK, &blocked_set, NULL) )
        fatal_error(errno, "sigprocmask()");
    printf("SIGINT is no longer blocked. Enter a few CTRL-Cs.\n");
    for ( i = 1; i <= 5; i++ )
        usleep(800000);
    return 0;
}
```

*Listing 9-7: A program that shows the effect of blocking and unblocking a signal*

This program is designed to demonstrate a few different properties of signals and blocking. After you build this executable, run it, enter several CTRL-Cs immediately, and observe what happens. After you see the message, "`SIGINT is no longer blocked. Enter a few CTRL-Cs`", enter them again and observe the difference. For example, I ran this program following these instructions and the session looked like this:

```
$ sigprocmask_demo1
SIGINT is blocked; sleeping for 5 seconds. Try entering a few CTRL-Cs.
^C^C^C^C^C^C  Caught SIGINT
SIGINT is no longer blocked. Enter a few CTRL-Cs.
^C  Caught SIGINT
^C  Caught SIGINT
^C  Caught SIGINT
^C  Caught SIGINT
^C  Caught SIGINT
```

The following rules explain its output and behavior.

- Blocked signals are not queued. While a signal is blocked, if it is generated multiple times, only one instance of it will be delivered when that signal is unblocked. If we enter CTRL-C six times while it is

blocked, when it becomes unblocked, the handler will run just once, not ten times. That's why you'll see just one message, "`Caught SIGINT`".

- POSIX requires that when a signal is unblocked by a call to `sigprocmask()`, if it is pending, the signal should be delivered to the process immediately, before the `sigprocmask()` call returns. That's why the handler's message, "`Caught SIGINT`", appears before the message printed by the following `printf()`, namely
  `SIGINT is no longer blocked. Enter a few CTRL-Cs.`
- The second `for`-loop suspends the process 0.8 seconds five times and if you observe the output, the number of signals caught, no matter how many you enter, will be five. If a signal is delivered during a system call, it interrupts that call. Some systems calls that are interrupted by signals automatically restart and others don't. Whether or not the call restarts after the handler runs depends on which call it is. The `signal(7)` man page has a detailed list describing how most system calls respond to the interruption. The `usleep()` system call is not restarted after the handler runs; it just terminates. This is why the number of messages printed by the handler is equal to the number of iterations of the `for`-loop.

In the displayed session, you can see that only one CTRL-C was delivered when it was unblocked, and that `usleep()` returned and did not resume whenever it was interrupted, otherwise we'd be able to enter more than five CTRL-Cs during that `for`-loop.

Here's another very small program that shows how to block all signals that user programs are allowed to block while it executes a small fragment of code.

*sigprocmask_demo2.c*

```
#include  "common_hdrs.h"
#include  <signal.h>

int main(int argc, char* argv[])
{
    sigset_t signals, prevsignals;

    printf("PID=%d\n",getpid());
    sigfillset(&signals);
    if ( -1 == sigprocmask(SIG_BLOCK, &signals, &prevsignals) )
        fatal_error(errno, "sigprocmask()");

    while ( TRUE ) {
        printf("Try sending signals to me. "
               "Use SIGKILL to terminate me, SIGSTOP to stop me.\n");
        sleep(5);
    }
    if ( -1 == sigprocmask(SIG_SETMASK, &prevsignals, NULL) )
        fatal_error(errno, "sigprocmask()");
    return 0;
```

```
}
```

*Listing 9-8: A program that blocks all signals*

Build and run this program in one terminal window, copy its PID, and in a second terminal window, send lots of signals to the program with the `kill` command, such as **`kill -s SIGABRT pid`**, where *`pid`* is the PID you copied. This is an easy way to temporarily prevent a portion of code from being interrupted by almost any signal.

Because a signal handler has just a single parameter, which is just the signal number, the only way that it can share data with the program is through file-scoped, or global, variables. At the start of the discussion about blocking signals on page 443, I mentioned that when a signal handler modifies variables that are shared with the rest of the program, in order to access them safely outside of the handler, the program should prevent the handler from running by blocking the signal while it accesses those variables. This next program demonstrates how we can do this.

In this program, the program counts how many times the signal handler was called and prints the count. If we didn't block the signal from arriving while the `main()` function updated and printed the count, the program might fail to count some delivered signals. This program declares a global variable, `sig_received`, as `volatile sig_atomic_t`, which is shared by the handler and `main()`.

### THE SIG_ATOMIC_T TYPE

The `signal.h(7POSIX)` man page defines `sig_atomic_t` as a "possibly volatile-qualified integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts." Let's break this down.

Variables that are declared to be `sig_atomic_t` can be read or written with a single, uninterruptible machine instruction. They are *atoms*, in the sense that they are moved around as single chunks in the machine. A data type consisting of larger pieces, such as a `struct`, is not an atom. Standard `int` types are usually atomic, but this is not guaranteed. A 64-bit integer might be moved as two 32-bit chunks on some architectures.

When compilers optimize code, they sometimes put variables into registers temporarily. If a variable is in a register, and another part of the program updates the in-memory copy, the value in the register is no longer valid. The `volatile` qualifier tells the compiler that it's not safe to do this, because the variable might be updated asynchronously by other parts of the same program. Therefore, it's common to see variables declared as `volatile sig_atomic_t` in code intended to access them atomically and possibly asynchronously.

The signal handler in this program sets this `sig_received` variable and does nothing else. The main program tests this variable and if it's set, increments a counter. The program has to block delivery of `SIGINT` while it tests the shared variable.

*sigprocmask_demo3.c*

```
#include  "common_hdrs.h"
```

```
#include  "common_hdrs.h"
#include  <signal.h>
static volatile sig_atomic_t sig_received = 0;

void catch_sigint(int signum)
{
    sig_received = 1;
}

int main (int argc, char* argv[])
{
    sigset_t blockedset;
    int i;
    int count = 0;

    /* Initialize the signal mask and install the handler. */
    sigemptyset (&blockedset);
    sigaddset (&blockedset, SIGINT);
    if ( SIG_ERR == signal(SIGINT, catch_sigint) )
        fatal_error(errno, "signal()");
    printf("PID=%d\n Enter CTRL-\\ to end this program.\n", getpid());
    while (TRUE)    {
        /* Block the signal while we print the count. */
        if ( -1 == sigprocmask (SIG_BLOCK, &blockedset, NULL) )
            fatal_error(errno, "sigprocmask()");
        if ( sig_received ) {
           count++;
           sig_received = 0;
        }
        printf("\n%d SIGINTs received so far\n", count);
        /* Unblock the signal, allowing handler to run. */
        if ( -1 == sigprocmask (SIG_UNBLOCK, &blockedset, NULL) )
            fatal_error(errno, "sigprocmask()");
      ❶ pause();
    }
}
```

*Listing 9-9: A program in which the signal handler updates an atomic variable accessed by the `main()` function*

This program introduces a new system call ❶, `pause()`,which suspends the calling process until it receives a signal that either terminates the process or causes a signal handling function to run. If the program doesn't have a signal handler for the signal and the default action is to ignore it, `pause()` does not return. Using `pause()` here serves two purposes. The first is that we have as much time as we need to send a signal, either by entering the `kill -s SIGINT` command in another terminal window, or by entering CTRL-C in the process's terminal. The second is that it makes it easy to verify that the reported count is correct, because each time we send a signal, it is only

delivered while the process is suspended in the `pause()`, causing the handler to run and the process to wake up, block signals again and update and print the count. The `pause()` system call is one method of waiting for a signal. There are several system calls designed for this purpose that we don't cover here, but which you can investigate on your own, including `sigsuspend()`, `sigwait()`, and `sigwaitinfo()`.

When you run this program, enter sequences of CTRL-C and notice that the number is counted correctly by the main program. You'll need to terminate it with a signal other than CTRL-C, such as CTRL-\.

## The sigaction() System Call

The `sigaction()` system call was introduced to replace the use of `signal()` for installing signal handlers and controlling their behavior. It overcomes the deficiencies of `signal()` that we described in "The signal() System Call" and it allows a programmer to specify how the handler will respond when multiple signals are sent to a program while it's executing a signal handler. It also provides a way for a program to obtain detailed information about the source and cause of delivered signals. However, this increased functionality comes with a cost, because it's harder to learn and understand, and it raises new questions we've yet to consider. We'll start by reading its man page. Its prototype is:

```
#include <signal.h>

int sigaction (int signum, const struct sigaction *act,
               struct sigaction *oldaction;
```

where

- `signum` is the value of the signal to be handled
- `act` is a pointer to a `sigaction` structure that specifies the handler, masks, and flags for the signal
- `oldact` is a pointer to a structure to hold the currently active `sigaction` data.

When called, it sets the disposition of signal (`signum`) based on the contents of the `sigaction` structure `act` and saves its current disposition in `oldact`. If successful, it returns 0, otherwise it returns -1 and sets `errno` accordingly. The man page tells us that we need to define the `_POSIX_C_SOURCE` feature test macro to use this function. Notice that the function name is the same as the name of the structure whose address is passed to it, like the `stat()` function and the `stat` structure.

Let's examine the `sigaction` structure first to learn what roles its various members play.

### The sigaction Structure

The `sigaction` structure is declared in the *signal.h* header file. The man page states that it's "something like":

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

The ambiguity is intentional, because the definition is more complicated than this. The page warns us that the two members `sa_handler` and `sa_sigaction` on some machines might be defined as a C `union`. A `union` is like a `struct` in which the members can have overlapping storage and therefore cannot have different values simultaneously. Figure 9-4 depicts a small `union`.

```
union sigval {
    int number;
    char w[4];
}
```

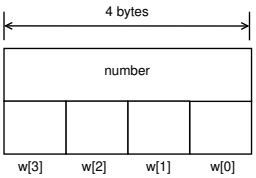



*Figure 9-4: A C union with a four-byte integer and a four-character string*

In the `sigaction` structure, the two members are both pointers to functions. The fact that the functions have different prototypes is not a problem, since they're both pointers, which have a fixed number of bytes regardless of what they point to.

The `sigaction` structure allows us to install either the old-style signal handler that we've been using, whose prototype has a single integer parameter, or, if we include the appropriate flag in the `sa_flags` member, the newer, POSIX-compliant type of signal handler whose prototype is:

```
void  (*sa_sigaction)(int signum, siginfo_t *info, void *ucontext);
```

but we must choose one or the other. Since we already know enough about the older method, we'll concentrate on the newer `sa_sigaction` type of signal handler. If the `SA_SIGINFO` flag is bitwise-or-ed into the `sa_flags` member, then the `sa_sigaction` member of the structure will be installed as the signal handler, not the `sa_handler` member, and the pointer must point to a function whose prototype matches it.

The remaining members of the `sigaction` structure are as follows.

**`sa_mask`** By default, the signal that caused the handler to run will be blocked during execution of the handler. This integer bitmask defines which other signals should also be blocked while the handler is running.

**sa_flags** This is an integer that encodes a set of flags that control how subsequent signals of the same type as the one that caused the handler to run are handled. For example, if a handler has caught a SIGINT signal and another SIGINT arrives while the handler is executing, then the flags in sa_flags will determine how to dispose of the second SIGINT, in effect overriding the default behavior of blocking it. The sa_flags member has no effect on arriving signals of other types. This sa_flags field is a bitwise-or of several flags, the most important of which are:

**SA_NODEFER** If set, the kernel will not automatically block signals of the same type while it's being handled, which it does by default. This implies that an arriving signal of the same type will cause the handler to be interrupted and a second instance of it re-entered with the second signal. A stack frame for the second instance is pushed on top of the stack frame for the first instance.

**SA_RESETHAND** When set, the signal action is reset to SIG_DFL. This means that as soon as the signal is delivered, the default action will take place. This flag implies the SA_NODEFER flag, because signals are not blocked. The difference is that instead of a second handler instance running, the process takes the default action for the signal. The intention is to make the handler behave like the old-style, mouse-trap-like signal() handler, since any signal of the same type arriving after the first will cause the default behavior.

**SA_RESTART** When set, certain system calls that would otherwise be terminated if a signal were delivered during their execution, will be restarted automatically. The signal(7) man page lists and describes the system calls that would be restarted if this flag were set.

**SA_SIGINFO** When set, the newer style sa_sigaction handler is installed, with three arguments passed to it. The first is the signal number. If the second argument is not NULL, it points to a siginfo_t structure containing the reason why the signal was generated; the third argument points to a ucontext_t structure containing the receiving process's context when the signal was delivered.

Two other flags that we'll make use of in Chapter 12 are SA_NOCLDSTOP and SA_NOCLDWAIT.

**sa_restorer** This function pointer is not used by any application. It is strictly for the use of the C libraries and we can safely ignore it.

We'll explore how sigaction() works by way of some examples. We've got several different aspects of its behavior to study. In particular,

- What information can a signal handler obtain when the SA_SIGINFO flag is enabled in the call to sigaction()?
- When a synchronous signal such as a SIGFPE is delivered to a process and a handler for it runs, when the handler returns, is the instruction with the error executed from the point after the error, or will the error occur again?

- How do the various combinations of the SA_NODEFER, SA_RESETHAND, and SA_RESTART flags that can be bitwise-ored into sa_flags affect how a program responds when signals of the same type as the one currently being handled are delivered to the process?
- When a signal handler is running and a signal of the same type is delivered because it's not blocked, the handler is interrupted and a second instance of it runs. How do we make the handler re-entrant so that data is not corrupted when this happens?

### Signal Information Passed to the Handler

When SA_SIGINFO is enabled in sa_flags, the signal handler that sigaction() installs is expected to have a prototype with three parameters, which are:

**int signum** The number of the signal causing the handler to run.

**siginfo_t* info** A pointer to a siginfo_t structure containing information about the signal such as what caused it, who the sender is, and so on.

**void* ucontext** A pointer to a ucontext_t structure, cast to void*. This structure contains information about the context of the program at the time the signal was delivered, such as what signals were blocked at the time and the location of the process stack. It is rarely used.

The first parameter is just the signal number, and the last is a structure that is rarely needed by the handler, so we'll concentrate on the second parameter, the siginfo_t structure.

The sigaction() man page on Linux contains a definition of this structure that shows all possible members it can have, partially reproduced here:

```
siginfo_t {
    int      si_signo;     /* Signal number */
    int      si_errno;     /* An errno value */
    int      si_code;      /* Signal code */
    int      si_trapno;    /* Trap num that caused hardware-generated signal */
    pid_t    si_pid;       /* Sending process ID */
    uid_t    si_uid;       /* Real user ID of sending process */
    int      si_status;    /* Exit value or signal */
    union sigval si_value; /* Signal value */❶
--snip--
    int      si_syscall;   /* Number of attempted system call */
    unsigned int si_arch;  /* Architecture of attempted system call */
}
```

Although the definition makes it appear as though all of these members are present in this structure, they aren't. The narrative following the definition explains that the structure is essentially a union and that the set of members actually present when the handler runs depends on which signal the handler caught. Most of these members are filled in only by a few signals but not

others. In Linux, the only three members that are guaranteed to be part of the structure regardless of the signal type are `si_signo`, `si_errno`, and `si_code`. In contrast, POSIX.1-2017 specifies a different set of mandatory members, namely:

```
int      si_signo;     /* Signal number */
int      si_code;      /* Signal code */
pid_t    si_pid;       /* Sending process ID */
uid_t    si_uid;       /* Real user ID of sending process */
int      si_status;    /* Exit value or signal */
union sigval si_value; /* Signal value */
void    *si_addr;      /* Memory location which caused fault */
```

The `si_value` field ❶ has type `union sigval`, which is defined in *siginfo.h*.

Which fields are filled in depends on the manner by which the signal is sent, the source of the signal, and the actual signal type. The idea is that when a particular signal is sent to a process and the handler catches it, some of this information is stored into selected fields that have meaning for that particular signal. For example, if a signal is sent by the `kill()` system call or the `kill` command, regardless of the signal type, then the `si_pid` and `si_uid` are filled in. In contrast, if a hardware-generated signal such as `SIGILL` or `SIGSEGV` is caught, then `si_addr` is filled with the address of the instruction causing the trap.

The simple program in Listing 9-10 is an example that demonstrates the first case.

*sigact_demo1.c*

```
#include "common_hdrs.h"
#include <signal.h>

void sig_handler (int signo, siginfo_t *info,  void *context)
{
    printf ("Signal number: %d\n", info->si_signo);      /* UNSAFE */
    printf ("PID of sender: %d\n", info->si_pid);        /* UNSAFE */
    printf ("UID of sender: %d\n\n", info->si_uid);      /* UNSAFE */
    /* Force the process to terminate by raising SIGTERM,
       for which we have no handler. */
    if ( signo == SIGINT)
        raise(SIGQUIT);
    else
        raise(SIGTERM);
}

int main (int argc, char* argv[])
{
    struct sigaction the_action;

    the_action.sa_flags     = SA_SIGINFO;
    the_action.sa_sigaction = sig_handler;
```

```
    if ( -1 == sigaction(SIGINT, &the_action, NULL))
        fatal_error(errno, "sigaction()");
    if ( -1 == sigaction(SIGQUIT, &the_action, NULL))
        fatal_error(errno, "sigaction()");
    printf ("Open a second terminal window and send SIGINT "
            "by entering kill -s SIGINT %d\n", getpid());
    pause();
    return 0;
}
```

*Listing 9-10: A program in which the signal handler displays information about the source of a signal*

The `main()` function sets the handler for both `SIGINT` and `SIGQUIT` to be the `sig_handler()` function.

**NOTE** *Although I may occasionally use a single function to catch more than one signal, in general it's not a good idea to do so. It's better to install separate handlers for each signal type. I only use a shared handler here to save space.*

After installing the handlers, the program prints instructions to open a second terminal window and then pauses, so that we can take our time in setting up the terminal.

The handler begins by printing out the values of the three members of the `siginfo_t` structure that are guaranteed to have data. Then, if it received a `SIGINT`, it raises a `SIGQUIT` so that it will run a second time. When it runs the second time, it will have received a `SIGQUIT` and will raise `SIGTERM`, which is unhandled and will terminate the program. This design allows us to compare the information delivered when the `kill()` system call sent the signal (through the `kill` command) as opposed to when it was sent by the process itself through `raise()`. Since `raise(signo)` is equivalent to `kill(getpid(), signo)`, the same fields are filled in by the two calls but the values will not be the same. A sample run of the program will look something like this:

```
$ sigact_demo1
Open a second terminal window and send SIGINT by entering kill -s SIGINT 12461
Signal number: 2
PID of sender: 4978
UID of sender: 500

Signal number: 3
PID of sender: 12461
UID of sender: 500

Terminated
```

Notice that the PID listed for the received `SIGQUIT` (signal number 3) is the same as the process's PID whereas the one listed for the `SIGINT` is different, because it's that of the `kill` command entered in a `bash` shell.

Let's look at another example in which the signal is caused by hardware. We can force a SIGFPE signal to be sent to a process by intentionally dividing by zero in our program and then examine the information in the signal handler. The only field filled in when a SIGFPE is received is the si_code field. The possible values for si_code and their meanings, are described in the POSIX.1-2017 standard (*https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/signal.h.html*). The POSIX specification of the header file *signal.h*, which we can read by entering **man signal.h**, also contains the codes.

Because the compiler might optimize our intentional arithmetic errors out of the code, we'll turn off optimization when we compile this program, which appears in Listing 9-11. Running this program let's us see some of the codes generated by floating-point exceptions.

*sigact_demo2.c*

```
#define _GNU_SOURCE
#include "common_hdrs.h"
#include <signal.h>
#include <math.h>
#include <fenv.h>❶

void fpe_handler (int signo, siginfo_t *info, void *context)
{   /* These calls to printf() are UNSAFE. */
    printf ("Signal: %s\n", strsignal(info->si_signo));
    switch ( info->si_code ) {
        case FPE_INTDIV :
        printf("Code: FPE_INTDIV (Integer divide by zero)\n");          break;
        case FPE_FLTDIV :
        printf("Code: FPE_FLTDIV (Floating-point divide by zero)\n");   break;
        case FPE_FLTOVF :
        printf("Code: FPE_FLTOVF (Floating-point overflow)\n");         break;
        --snip--    }
  ❷ raise(SIGTERM);
}

int main(int argc, char* argv[])
{
    struct sigaction action;
    sigset_t     blocked;
    float        y=2.0, z = 0.0;
    signed int   n=1, m=2;

    action.sa_sigaction = fpe_handler;
    action.sa_flags = SA_SIGINFO;
    sigemptyset(&blocked);
    action.sa_mask = blocked;
  ❸ int excepts = FE_DIVBYZERO|FE_INEXACT|FE_INVALID|
                  FE_OVERFLOW|FE_UNDERFLOW;
    feenableexcept(excepts);
    if ( sigaction(SIGFPE, &action, NULL) == -1 )
```

```
        fatal_error("sigaction");
    m = 2*n - m;  /* m == 0 but compiler doesn't detect it. */
    if ( argc > 1 )
        if ( argv[1][0] =='f' )
            n = (int) y/z;
        else if ( argv[1][0] =='o' )
            feraiseexcept(FE_OVERFLOW);
    else
        n = n/m;
    return n; /* Prevent compiler from warning about unused n. */
}
```

*Listing 9-11: A program in which the signal handler displays information about a hardware generated signal*

Because the program is using functions from C's floating-point exception library, it must include the *fenv.h* ❶ header file and the math library must be linked into the code (with `-lm`). This program is designed to produce three different types of floating-point errors. If an 'f' is given as the program argument, it will divide execute the statement, `n = (int) y/z`. Since `z` is really zero, this results in a floating-point divide-by-zero. If an 'o' is given, it will raise a floating-point overflow artificially, by calling `feraiseexcept(FE_OVERFLOW)`. Otherwise it will evaluate `n/m`, in which `m` is exactly zero, resulting in an integer-divide-by-zero. In order to force the floating-point traps to occur, the program enables them by calling `feenableexcept()`, passing a mask ❸ containing all allowable traps. The *env.h* ❶ exposes the various floating-point exception-related functions and constants. To save space, only the relevant parts of the signal handler's `switch` statement are displayed.

The handler's call to `raise(SIGTERM)` ❷ forces the program to terminate. Without it, the program would enter an infinite loop. To experience this yourself, comment out this call and recompile and run the program. You'll see that it repeatedly outputs the following two lines

```
Code: FPE_INTDIV (Integer divide by zero)
Signal: Floating point exception
```

This is because, when a signal handler returns from execution, the program normally resumes execution in the instruction that was interrupted. In the case of hardware-detected errors such as floating-point exceptions, the very code that caused the trap will be re-executed, causing an infinite cycle of traps. A signal-handler must either terminate the program explicitly or raise an exception such as `SIGTERM` that causes it to terminate.

We compile and run this program, first without any arguments and then with 'f' followed by 'o':

```
$ sigact_demo2
Signal: Floating point exception
Code: FPE_INTDIV (Integer divide by zero)
Terminated
$ sigact_demo2  f
```

```
Signal: Floating point exception
Code: FPE_FLTDIV (Floating-point divide by zero)
Terminated
$ sigact_demo2  o
Signal: Floating point exception
Code: FPE_FLTOVF (Floating-point overflow)
Terminated
```

This output confirms that the `si_code` member of the `info` parameter can be used to determine the exact type of error that was trapped when the program ran.

To obtain similar information about the causes of other signals, we need to consult the `sigaction` man page, which has the remaining details about exactly which signals populate the various members of the `siginfo_t` structure. As an alternative, the POSIX.1-2017 specification has tables that show the possible values assigned to `si_code` as well as the remaining fields, for each signal type. We can refer to them as needed when writing code that needs this type of information.

### *Effect of sa_flags on Signal Handler Execution*

Let's turn our attention to studying the effects of the different possible `sa_flags` on a program's signal handling. The three flags we consider are `SA_RESETHAND`, `SA_NODEFER`, and `SA_RESTART`. The best way to understand what these flags do is to write a program that let's us see these effects, both in isolation and in combination with each other. We'll develop a small program that does exactly this.

First let's review the difference between how a handler behaves with and without the `SA_NODEFER` flag being enabled. Suppose that a signal handler is running because it received a signal of some hypothetical type `SIGX`. If `SA_NODEFER` was not enabled when the handler was installed, signals of type `SIGX` will be blocked by default. In this case, if a second `SIGX` is sent, it will not be delivered to the process until the signal handler terminates. If more than one `SIGX` arrives while it's blocked, only a single instance is delivered. On the other hand, if the `SA_NODEFER` flag is enabled, a signal of type `SIGX` will be delivered while the handler is running, interrupting the handler. A second instance of the handler will run. If another `SIGX` arrives, it will interrupt the second instance of the handler, and so on. It behaves like a recursive function.

With this in mind we can design a handler. So that we can tell whether a second call of the handler interrupted a first, as opposed to the second call starting after the first finished executing, we'll have the handler generate a unique number based on the time it was called, accurate to the millisecond, and print a message containing that number as soon as it starts running and just before it exits. The printed messages will show us the ordering of the calls. Therefore, our handler's logic, step by step should be:

1. On entry, the handler gets the current time with at least millisecond accuracy and uses that time to generate a unique number, which we'll name `call_id`.
2. It prints a short message that the handler was entered, along with its `call_id` and the type of signal it received.
3. To allow enough time for multiple signals to be delivered while the handler is running, it then spins in a loop that does nothing, just to prolong its running time.
4. When it is about to exit, it prints a second message that it is exiting, along with its `call_id` and signal type.

Suppose the entry and exit messages are of the form

```
Entered handler for signal SIGINT, ID=1234567
Leaving handler for signal SIGINT, ID=1234567
```

Then, with this design, if signals aren't blocked, meaning `SA_NODEFER` is set, and they're sent quickly enough so that the handler is running when they're sent, the sequence of printed messages should look like matching bookends:

```
Entered handler for signal SIGINT, ID=521400
Entered handler for signal SIGINT, ID=521500
Entered handler for signal SIGINT, ID=521600
--snip--
Leaving handler for signal SIGINT, ID=521600
Leaving handler for signal SIGINT, ID=521500
Leaving handler for signal SIGINT, ID=521400
```

On the other hand, if they're blocked, (`SA_NODEFER` not set) the sequence will instead be a sequence of interleaved entrance and exit messages, such as this:

```
Entered handler for signal SIGINT, ID=521400
Leaving handler for signal SIGINT, ID=521400
Entered handler for signal SIGINT, ID=521500
Leaving handler for signal SIGINT, ID=521500
Entered handler for signal SIGINT, ID=521600
Leaving handler for signal SIGINT, ID=521600
--snip--
```

When we researched functions for working with time in Chapter 4, we came across one named `clock_gettime()`. We decided we didn't need it for implementing the `date` command because it was accurate to the nanosecond, but we'll use it now. Its prototype is:

```
#include <time.h>
int clock_gettime(clockid_t clockid, struct timespec *tp);
```

The `clockid` is a constant indicating which clock to use. In our case we'll use the one named `CLOCK_REALTIME`, which is like a wall clock's time. The `struct timespec` is defined by

```
struct timespec {
    time_t   tv_sec;        /* seconds */
    long     tv_nsec;       /* nanoseconds */
};
```

The handler will call `clock_gettime()` to get the current time, accurate to the nanosecond, storing it into a `timespec` structure (`t`). To generate the `call_id`, it will multiply the number of seconds ( `tv_sec`) by 1000 and add the number of nanoseconds (`tv_nsec`) divided by 1,000,000 to get a number of milliseconds. To make `call_id` shorter, we'll drop the high-order digits in the number of seconds. The instruction is therefore:

```
call_id = 1000 * (t.tv_sec & 0xFFF)  + (t.tv_nsec / 1000000 );
```

We're ready to assemble the handler function, which is in Listing 9-12.

*sig_handler()*

```
 void sig_handler (int signo, siginfo_t *info, void *context)
{
    int      call_id;  /* Number to uniquely identify signal handler run. */
    int      i, j = 0;
    struct timespec t;  /* Time handler starts */

    /* Get current time in nanoseconds. */
    if ( -1 == clock_gettime(CLOCK_REALTIME, &t) )
        raise(SIGTERM);
    /* Create an id to uniquely identify this call to handler. */
    call_id = (t.tv_sec & 0xFFF)*1000 + (t.tv_nsec / 1000000 );
    printf("Entered handler for SIG%s ID=%d\n",
           sigabbrev_np(info->si_signo), call_id );
    /* Artificially delay handler to allow time for signals to arrive. */
    for ( i = 0; i < 200000000; i++ ) { j++ ; }
    printf("Leaving handler for SIG%s ID=%d\n",
           sigabbrev_np(info->si_signo), call_id );
}
```

*Listing 9-12: The signal handler for* sigact_demo3.c

Let's turn to the main program now. Because we're also interested in the effect of the `SA_RESTART` flag, which determines whether or not system calls are restarted if a signal arrives while they're executing, the program needs to make a system call, not just any call, but one that blocks waiting for the user to do something. The ideal candidate is the `read()` call. The `signal(7)` man page listed this call as one that can be restarted if it's waiting for a *slow* device. Reading from the terminal is considered slow; therefore our main program will have a `while`-loop that repeatedly calls `read()` to read a small number of characters from the terminal.

The program will prompt the user to enter a few characters and provide a way for the user to terminate the program by entering `quit`. It'll check the return value of `read()` each time. If it's -1, we'll see if the `errorno` value is `EINTR`, which indicates it was interrupted by a signal. If so, we'll print a message, otherwise we'll print whatever the user entered.

There are a few complications. First, we've never used `read()` to read from a terminal. Unlike a read from a file, a read from a terminal does not return until the user presses **`ENTER`**. When we study terminals, we'll see how to prevent that, but for now we have to work with this limitation. Also, it's inadvisable to mix calls to the I/O library functions such as `printf()` with system calls such as `read()` in the same program; the library functions can interfere with the reads. Therefore, we'll use `write()` to write our messages to the terminal.

The last problem is what happens if the user enters too many characters. For example, suppose the program asks the user to enter 12 characters but they enter 20. Where are those characters stored? Are they discarded? Are they saved for the next `read()`. These questions all pertain to the subject of terminals, which we'll study in Chapter 14. A simplified answer, for now, is that there's a hidden queue that contains the characters that the user enters, and that, if a call to `read()` requested *N* characters, as soon as *N* characters are in the queue, `read()` returns, leaving any other entered characters in the queue for the next call to `read()`.

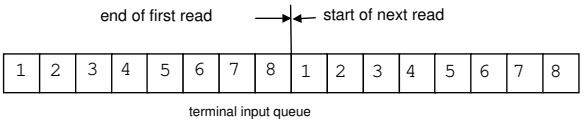



*Figure 9-5: A read of eight characters from the terminal, showing where the next read operation will start.*

If we don't want those extra characters in the queue, we can discard them by calling `tcflush()`, which is a function in the `TERMIOS` library. We give this function the file descriptor for the terminal and an operation code:

```
tcflush{STDIN_FILENO, TCIFLUSH); /* Flush all input from terminal. */
```

We also need to empty the input buffer before each read operation, otherwise if the user enters, say ten characters the first time, and only six the second, when we print out what they entered, we don't see the four characters from the first iteration.

The pseudocode loop body is therefore

1. Zero the buffer into which `read()` will store the user's text, using `memset(buffer, 0, buffer_size)`.
2. Flush the terminal queue in case there's anything there, with `tcflush(STDIN_FILENO,TCIFLUSH)`.
3. Display a prompt string, with `write(STDOUT_FILENO, prompt, strlen(prompt))`.

4. Read from the terminal:
   `chars_entered = read(STDIN_FILENO, &buffer, buffer_size)`.
5. If the read was interrupted display a message to that effect, otherwise display what the user entered.

We put all of this together into the program shown in Listing 9-13, omitting the handler code. The complete program is available in the source code distribution for the book.

*sigact_demo3.c*

```
#include  "common_hdrs.h"
#include  <signal.h>
#include  <termios.h> /* Needed for tcflush */

/* Prototype for handler, shown in previous listing */
void sig_handler (int signo, siginfo_t *info, void *context);

int main(int argc, char* argv[])
{
    const  int  maxsize = INPUTLEN; /* Maximum input size */
    const char  intr_message[] = "    read() was interrupted.\n";
    const char  out_label[]    = "Entered text:";
    char        buffer[maxsize+2]; /* INPUTLEN plus newline and null byte  */
    struct sigaction action;
    sigset_t    blocked;             /* Set of blocked sigs */
    int         flags = 0;
    int         n, i = 1;
    char        prompt[128];
    int         reply_len = strlen(out_label);
    int         intr_message_len = strlen(intr_message);
    int         prompt_len;

    sprintf(prompt, "Type at most %d characters, then <ENTER>"
                            "(or 'quit' to quit):", maxsize);
    prompt_len = strlen(prompt);

    /* Get command line arguments and check which ones user entered.       */
    while ( i < argc ) {
        if (0 == strncmp("reset", argv[i], strlen(argv[i])) )
            flags |= SA_RESETHAND;
        else if (0 == strncmp("nodefer", argv[i], strlen(argv[i])) )
            flags |= SA_NODEFER;
        else if (0 == strncmp("restart", argv[i], strlen(argv[i])) )
            flags |= SA_RESTART;
        i++;
    }

    /* Set up sigaction.                                                   */
    action.sa_sigaction = sig_handler;       /* SIGINT handler             */
```

```
    action.sa_flags = SA_SIGINFO | flags;    /* Add the entered flags.     */
    sigemptyset(&blocked);                   /* Clear all bits of mask.    */
    action.sa_mask = blocked;                /* Set blocked mask.          */

    /* Install sig_handler as the SIGINT handler */
    if ( sigaction(SIGINT, &action, NULL) == -1 )
        fatal_error(errno, "sigaction");

    while( TRUE ) {
        memset((void*)buffer, 0, maxsize+2);   /* Zero input buffer.        */
        tcflush(STDIN_FILENO,TCIFLUSH);        /* Remove bytes never sent.  */
        write(STDOUT_FILENO, prompt, prompt_len); /* Write prompt string.   */
        n = read(STDIN_FILENO, &buffer, maxsize); /* Read user input.       */
        if (-1 == n  &&  EINTR == errno )      /* If interrupted by signal  */
            write(STDOUT_FILENO, intr_message, intr_message_len);
        else {
            if ( strncmp("quit", buffer, 4) == 0 )  /* User wants to quit.  */
                break;
            else {             /* Write the entered characters to terminal. */
                write(STDOUT_FILENO, &out_label, reply_len);
                if ( n >= maxsize ) /* In this case, need to add a newline. */
                    buffer[n] = '\n';
                write(STDOUT_FILENO,&buffer,n+1);
            }
        }
    }
    return 0;
}
```

*Listing 9-13: A program that can be used to test the effects of several different sa_flags*

We can run this program without any arguments, or with any combination of the words, `reset`, `nodefer`, and `restart`. First, run it with just `reset` and don't even enter any characters. Just enter two CTRL-Cs, slowly:

```
$ sigact_demo3 reset
Type at most 12 characters, then <ENTER>(or 'quit' to quit):^CEntered handler
for SIGINT ID=53264325
Leaving handler for SIGINT ID=53264325
    read() was interrupted.
Type at most 12 characters, then <ENTER>(or 'quit' to quit):^C
$
```

The flag puts the handler into mouse-trap mode, so that the second CTRL-C terminates the program. It never reaches the second call to `read()`. If you run it again but enter the CTRL-Cs faster, you won't even see the first message that the read was interrupted.

Now try running it with the `restart` argument and nothing else. This time enter a few CTRL-Cs rapidly:

```
$ sigact_demo3 restart
Type at most 12 characters, then <ENTER>(or 'quit' to quit):^CEntered handler
for SIGINT ID=53551971
^C^CLeaving handler for SIGINT ID=53551971
Entered handler for SIGINT ID=53552390
^CLeaving handler for SIGINT ID=53552390
Entered handler for SIGINT ID=53552785
Leaving handler for SIGINT ID=53552785
```

The display does not show the prompt character because the program is still in the `read()` system call, waiting for input, evidence that the `read()` was restarted. You can enter **`quit`** or terminate it with CTRL-\ or continue by pressing **`ENTER`**, in which case you'll get the prompt back. Notice that the signals are not blocked; all of them were delivered to the handler, but they were queued, so that the handler got them one after another. Try this again but enter some text to see that it outputs the text.

The next test is to run it with the `nodefer` argument. It's best to try it by itself first, without entering text:

```
$ sigact_demo3 nodefer
Type at most 12 characters, then <ENTER>(or 'quit' to quit):^CEntered handler
for SIGINT ID=53972397
^CEntered handler for SIGINT ID=53972556
^CEntered handler for SIGINT ID=53972732
^CEntered handler for SIGINT ID=53972868
Leaving handler for SIGINT ID=53972868
Leaving handler for SIGINT ID=53972732
Leaving handler for SIGINT ID=53972556
Leaving handler for SIGINT ID=53972397
    read() was interrupted.
Type at most 12 characters, then <ENTER>(or 'quit' to quit):quit
$
```

The signals were all delivered, and each interrupted the previous one.

If you run this program and enter more than 12 characters at the prompt, the excess will be discarded. But you should try the following experiment: comment out the call to flush the input queue, recompile the program, and enter dozens of characters at the prompt. What do you see?

The preceding program showed how to detect when a `read()` from the terminal was interrupted by a signal. If we want to design a handler for a signal such as `SIGINT`, enabling restarting of interrupted system calls, the handler should print a suitable message to the terminal when it runs, telling the user to re-enter the text. The following simple, old-style handler demonstrates this idea:

```
/* File-scoped variables */
volatile sig_atomic_t  got_interrupt= 0;
char alert[] = "\nSignal caught; re-enter input.\n>";
```

```
int  alertlen; /* In main(), assign with alertlen = strlen(alert). */

void on_interrupt(int signo)
{
    /* Any other handling code would be here. */
    got_interrupt = 1;
    write(1, alert, alertlen);
}
```

Notice that in this example, we call `write()` instead of `printf()`, because it's signal safe. If this handler is installed in a program to catch `SIGINT`, and that signal is delivered, the user will see a message such as

```
Signal caught; re-enter input.
>
```

with the prompt (>) indicating that it is waiting for more input.

## Guidance on Designing Signal Handlers

This section is a short list of dos and don'ts in the design of signal handlers. We've already mentioned a few of these.

- Most of the time, it's best to do as little as possible inside a signal handler. If receiving a particular signal requires that a significant amount of work needs to be done, the handler should set a `sig_atomic_t` flag that the main program can monitor periodically. The main program should then do the work. The exception to this rule is when the main program does essentially nothing and all of the work is performed by signal handlers. When we explore the design of interactive programs, we'll see how this works.
- Many functions are considered to be unsafe when called from inside a signal handler. The complete list of them, as well as an explanation and guidance on signal safety, is in the `signal-safety (7)` man page. Your programs should not call any of these functions from within a signal handler. Since `printf()` isn't safe, to print messages from within a handler, we can try to use the `write()` system call. If the message requires the kind of formatting only available with `printf()`, then if it's possible, we can create a formatted string and pass it to `write()`. Sometimes it's possible to just set a flag in the handler and write outside of it. The following code snippet suggests how to do this.

  ```
  static volatile sig_atomic_t flag = 0;

  void catch_sigint(int signum)
  {
      flag = 1;
  }
  ```

```
int main (void)
{
--snip--
    sigprocmask (SIG_BLOCK, &blockedset, NULL);
    if ( flag ) {
            printf("SIGINT received\n");
            flag = 0;
    }
    else
        printf("SIGINT not received\n");
    sigprocmask (SIG_UNBLOCK, &blockedset, NULL);
}
```

- Signal handlers are usually meant to be the last code executed when a signal is delivered to a process. Usually, the program should exit after the handler runs, but sometimes it needs to perform a few short tasks first. There are advanced techniques for jumping to a different part of a program's code in this case, but we don't discuss them here. You can read about `sigsetjmp()` and `siglongjmp()` in their man pages.
- It is not a good idea to use the same handler for more than one signal type; it makes it all the harder to design the handler to be reentrant.

You can find more extensive guidance on the design of signal handlers in the GNU C Library Reference Manual (*https://www.gnu.org/software/libc/manual/*).

In short, signal handlers are usually called asynchronously, at unpredictable times, to do as little as possible. The call can happen between the beginning and the end of a C operator that requires multiple instructions. Even copying one integer variable into another can take two instructions on most machines. If a handler uses global or static variables that are not `sig_atomic_t`, results can be unpredictable. If it spends too much time, there's chance another signal might arrive. Keep them short and simple whenever possible.

## Summary

Signals are essentially empty messages that are sent to processes to notify them of events requiring their attention. They were originally designed for exceptional events such as arithmetic errors or attempts by users to terminate a process, but now they're used more extensively and they also serve as a simple means of inter-process communication. There are many different types of signals, with each distinct type represented by a unique number that has a symbolic name beginning with `SIG`, such as `SIGINT` or `SIGTERM`.

Signals can be sent to a process by the kernel, by users with the `kill` command, and by other processes with the `kill()` system call. Users and processes require appropriate permission to send signals to a process. A process can also send a signal to itself with the `raise()` system call. Technically, the kernel does all of this sending; users and processes only make requests to the kernel to send signals. Despite this, we usually say that users and processes send the signals.

When a request is made to send a signal to a process, or an event occurs that requires that a signal be sent to that process, the kernel generates the signal by updating some data structures representing the state of that process. The signal is considered to be delivered to a process when the process receives and responds to it. Until it's delivered, it's called a pending signal.

Processes can temporarily block the delivery of most types of signals by creating signal masks. Blocked signals are delivered when the process unblocks them. If multiple signals of the same type are delivered to a process that has blocked them, all but one of them will be discarded.

Signal delivery is usually asynchronous with respect to process execution, which means that the time at which the process receives it is independent of where the process is in its computation. Signals sent by users, other processes, and the kernel are typically asynchronous. In contrast, a signal that's due to a hardware exception caused by the process itself is delivered synchronously — each time the process runs, the signal is delivered at the exact same time in the process's execution.

A process's disposition of a signal is what it does in response to it. There are three possible responses. One is to accept the default action associated with the signal's type, which is typically termination of the process. Another is to ignore the signal explicitly. The third is to execute a function that the process previously designated to be invoked whenever a signal of that type is delivered to it. This type of function is called a signal handler, and when it's run, we say the signal's been caught by it. Designating a signal handler function to be invoked on receipt of a signal is called registering or installing it.

The `signal()` system call was the original method of installing signal handlers in early UNIX systems and it's still available in most systems, but because it isn't standardized, it should not be used. Modern applications should use the POSIX-conforming `sigaction()` system call. This latter call provides much finer and greater control over how signals are handled. For example, it lets us control what a signal handler does when signals arrive during its execution, and what types of information are available for the handler to access.

## Exercises

1. Write a function with prototype

```
int printsigset(sigset_t set);
```

that prints, on a single line of standard output, a list of the numbers of all signals in the set `set` or prints `empty set` if it has none. It should return -1 on error and 0 on success. Use the functions from glibc and the appropriate feature test macro.

2. One function that we didn't explore is `sigpending()`. Read its man page and then write a function that prints on a single line of standard output, a list of the numbers of all pending signals or prints `no pending signals` if there are none. It should return -1 on error and 0 on success.
3. The `abort()` library function terminates the calling process by sending it a `SIGABRT` signal. However, because the caller might have a signal handler for `SIGABRT`, it has to do more than simply raising this signal. Read its man page and then write an implementation of it.
4. In the chapter we didn't explore method of waiting for signals other than the `pause()` system call. That call suspends a process until a signal arrives and its signal handler runs, or it is unhandled and terminates the process. Why would a process want to suspend itself until a signal arrives? Can you think of applications for which this is useful? List a few of them.
5. The `sigwait()` function suspends the calling process until a signal of a specified type arrives. Read its man page and write a program that:
    - Prints its process id and then blocks all signals.
    - Sleeps for 30 seconds.
    - Suspends itself waiting for signals to become pending.
    - Prints a list of all signals that are pending.

    It should be designed so that a `SIGINT` will terminate it after the pending signals are printed. You can open a second terminal to send signals to this process.

# A

# CREATING LIBRARIES

Here, I'll show you how to create and use static and shared libraries in a Unix environment. This summary is tutorial and elementary. I begin by explaining a bit about software libraries in general, then proceed to describe the differences between static and shared libraries. I then describe the how-to's about creating and using both types of libraries using the tools available in a GNU-based Unix system such as Linux. The discussion here is limited to executables and libraries in the *Executable and Link Format* (*ELF*), which is the format used by Linux and most Unix systems at the time of this writing. If you don't know what this means or why it might be important, that's fine; you may safely ignore this.

If you think you don't need the conceptual discussions, you can just "cut to the chase" and jump directly to the appropriate section below, either "Creating a Static Library" and "Using (Linking to) a Static Library" for static libraries or "Creating a Shared Library" and "Using a Shared Library" for shared libraries. For a more advanced explanation about cre-

ating and using library files, I recommend that you read David Wheeler's *http://tldp.org/HOWTO/Program-Library-HOWTO/index.html*Program Library HOWTO.

## About Libraries

A *software library*, also called a *program library*, is a file containing compiled code and possibly data that can be used by other programs. Libraries are not stand-alone executables – you can not "run" a library. They contain things like functions, type definitions, and useful constants that other programs can use. You have been using software libraries since your very first "Hello World" program, whether you knew it or not. Whatever function that you used to print those words on the screen was contained in a library, most likely either the C standard I/O library (if you used `printf`, for instance), or the C++ iostreams library (if you used the insertion operator of the `cout` ostream object.)

Perhaps you might have reached the point where you realize that you are writing useful code, code that you might want to use in more than one project, and that while you could continue to copy those functions into each new project, perhaps you would like to reuse that code in a more efficient way by creating a library file that contains it. If so, read on.

## Static Versus Shared Libraries in Unix

In Unix, there are two kinds of library files, static and shared. The term '*static library* is short for statically linked library. A *static library* is a library that can be linked to the program statically, after the program is compiled, as part of the program executable file. In other words, it is incorporated into the program executable file as part of the build of that executable. A *shared library* is a library that is linked dynamically, either at *loadtime* or at *runtime*, depending on the particular system. Loadtime is when the program is loaded into memory in order for it to execute. Runtime is the interval of time during which it is actually running. If linking is delayed until runtime, then a symbol such as a function in the library is linked to the program only when the program calls that function or otherwise references that symbol. The fact that a shared library is a dynamically linked library is not to be confused with the use of that term by Microsoft in what they call a DLL. While *DLL* is short for "dynamically linked library", DLLs are different from Unix shared libraries. In these notes, I use the term in the more general sense of a library that is linked to a program either at loadtime or at runtime.

*Static linking*, which was the original form of linking, resolves references to externally-defined symbols such as functions, by copying the library code directly into the executable file when the executable (file) is built. The *linkage editor*, also called the *link editor*, or just the *linker*, performs static linking. The term "linker" is a bit ambiguous, so I will avoid using it. The primary advantage of static linking, perhaps now the only advantage, is that the executable is self-contained and can run on multiple platforms. For exam-

ple, a program might use a graphical toolkit such as GTK that may not be present on all systems. With the toolkit's libraries statically linked into the executable, the executable can run on other systems (with the same machine architecture) without requiring the users on those systems to install those library files. Once upon a time, static linking resulted in faster code as well, but the gain is negligible today.

*Dynamic linking* can be done either when the program is loaded into memory, or while it is running and references an unresolved symbol. In the former case, the start-up time of the program is slightly longer than if it had been statically linked, since the libraries have to be located in memory (and possibly loaded into memory if they were not already there) and then linked to the program before it can actually begin execution. In the latter case, the program will experience slightly longer running time, because whenever an unresolved symbol is found and must be resolved, there is a bit of overhead in locating the library and linking to it. This latter approach is the more common approach because it only links symbols that are actually used. For example, if a function from a shared library is not called during execution, it will not be linked to the library at all, saving time.

There are several advantages of linking dynamically over linking statically. One is that, because the executable program file does not contain the code of the libraries that must be linked to it, the executable file is smaller. This means that it loads into memory faster and that it uses less space on disk. Another advantage is that it makes possible the sharing of memory resources. Instead of multiple copies of a library being physically incorporated into multiple programs, a single memory-resident copy of the library can be linked to each program, provided that it is a shared library. Shared libraries are dynamically-linked libraries that are designed so that they are not modified when a process uses them. This is why they have the `.so` extension — short for shared object.

Another advantage of linking to shared libraries is that this makes it possible to update the libraries without recompiling the programs which use them, provided the interfaces to the libraries do not change. If bugs are discovered and fixed in these libraries, all that is necessary is to obtain the modified libraries. If they were statically linked, then all programs that use them would have to be recompiled.

Still other advantages are related to security issues. Hackers often try to attack applications through knowledge of specific addresses in the executable code. Methods of deterring such types of attacks involve randomizing the locations of various relocatable segments in the code. With statically linked executables, only the stack and heap address can be randomized: all instructions have a fixed address in all invocations. With dynamically linked executables, the kernel has the ability to load the libraries at arbitrary addresses, independent of each other. This makes such attacks much harder.

## Identifying Libraries

Static libraries can be recognized by their ending: they end in "`.a`" . Shared libraries have a ".so" extension, possibly with a version number following, such as *librt.so.1*. Both types of libraries start with the prefix "*lib*" and then have a unique name that identifies that library. So, for example, the standard C++ static library is *libstdc++.a*, and the shared real-time library is *librt.so.1*. The "*rt*" in the name is short for real-time.

## Creating a Static Library

The steps to create a static library are fairly simple. Suppose that you have one or more source code files containing useful functions or perhaps definitions of useful types. For the sake of precision, suppose that *timestuff.c* and *errors.c* are two such files.

1. Create a header file that contains the prototypes of the functions defined in *timestuff.c* and *errors.c*. Suppose that file is called *utils.h*.
2. Compile the C source files into object files using the command

   ```
   $ gcc -c timestuff.c gcc -c errors.c
   ```

   This will create the two files, *timestuff.o* and *errors.o*.
3. Run the GNU archiver, `ar`, to create a new archive and insert the two object files into it:

   ```
   $ ar rcs libutils.a timestuff.o errors.o
   ```

   The **`rcs`** following the command name consists of a one-letter operation code followed by two modifiers. The "`r`" is the operation code that tells `ar` to insert the object files into the archive. The ``c'' and ``s'' are modifiers; `c` means "create the archive if it did not exist" and `s` means "create an index," like a table of contents, in the archive file. The name of the archive is given after the options but before the list of files to insert in the archive. In this case, our library will be named `libutils.a`.
   This same command can be used to add new object files to the library, so if you later decide to add the file `datestuff.o` to your library, you would use the command

   ```
   $ ar rcs libutils.a datestuff.o
   ```

4. Install the library into some appropriate directory, and put the header file into an appropriate directory as well. I use the principle of " most-closely enclosing ancestral directory" for installing my custom libraries. For example, a library that will be used only for programs that I write for my Unix System Programming class will be in a directory under the directory containing all of those programs, such as:

```
~/unix_demos/lib/libutils.a
```

and its header will be

```
~/unix_demos/include/utils.h
```

If I have a library, say *libgoodstuff.a*, that is generally useful to me for any programming task, I will put it in my *~/lib* directory:

```
~/lib/libgoodstuff.a
```

with its header in my *~/include* directory:

```
~/include/goodstuff.h
```

5. Make sure that your `LIBRARY_PATH` environment variable contains paths to all of the directories in which you might put your *static* library files. Your *.bashrc* file should have lines of the form:

```
LIBRARY_PATH=$LIBRARY_PATH:~/lib:
export LIBRARY_PATH
```

so that `gcc` will know " where to look" for your custom static libraries. If you want your libraries to be searched before the standard ones, then reverse the order:

```
LIBRARY_PATH=~/lib:\$LIBRARY_PATH
export LIBRARY_PATH
```

6. Make sure that your `CPATH` or `C\_INCLUDE_PATH` (or if using C++, your `CPLUS\_INCLUDE_PATH`) contains the path to the directory in which you put the header file. My *.bashrc* file has the lines

```
CPATH=~/include
export CPATH
```

**NOTE** *Do not put your static libraries into the same directories as your shared libraries. Keep them separate.*

## Using (Linking to) a Static Library

To use the library in a program, (1) you have to tell the compiler to include its interface, i.e., its header file, and (2) you have to tell the linkage editor to link to the library itself. The first task is accomplished by putting an include directive in the program. The second task is achieved by using the `-l` option to `gcc` to specify the *name* of the library. Remember that the name is everything between `lib` and the "`.`". The `-l` option must follow the list of files that refer to that library. For example, to link to the *libutils.a* library you would do two things:

1. In the program you would include the header file for the library:

```
\#include "utils.h"
```

2. To build the executable, you would issue the command

```
gcc -o myprogram myprogram.c -lutils
```

or the following if you did not modify your `CPATH`:

```
gcc -o myprogram myprogram.c -lutils -I~/unix_demos/include
```

but in either case, only if you are certain that there is not a shared library with the same name in a directory that will be searched ahead of the one in which *libutils.a* is located, or in the same directory as *libutils.a*. This is because `gcc`, by default, will always choose to link to a shared library of the same name rather than a static library of that name. This is one reason why you should not put static libraries in the same directory as shared libraries.
If you get the error message

```
/usr/bin/ld: cannot find -lutils
collect2: ld returned 1 exit status
```

it means that you did not set up the `LIBRARY_PATH` properly. (Did you export it? Did you type it correctly?)
If you want to be safe, you can use the `-L`*dir* option to the compiler. This option adds *dir* to the list of directories that will be searched when looking for libraries specified with the `-l` option, as in

```
gcc -o myprogram myprogram.c -L~/unix\_demos/lib -lutils
```

Directories specified with `-L` will be searched before those contained in the `LIBRARY_PATH` environment variable.

If you do a web search on this topic, you may see instructions for building your program of the form

```
gcc -static myprogram.c  -o myprogram -lutils
```

This will probably fail with the error message

```
/usr/bin/ld: cannot find -lc

collect2: ld returned 1 exit status
```

because the `-static` option tells `gcc` to statically link `myprogram.c` to all libraries, not just `libutils.a`. Since the C standard library no longer ships as a static library with most operating systems, the link editor, `ld`, will not find `libc.a` anywhere. Do not try to use the `-static option`. Follow my instructions instead.

## Creating a Shared Library

The `ar` command does not build shared libraries. You need to use `gcc` for that purpose. Before diving into the details though, you need to understand a few things about shared libraries in Unix to make sense out of the options to be passed to `gcc` to create the library.

### *Shared Library Names*

Every shared library has a special name called its *soname*. The *soname* is constructed from the prefix `lib`, followed by the name of the library, then the string `.so`, and finally, a period and a version number that is incremented whenever the interface changes. So, for example, the soname of the math library, `m`, might be `libm.so.1`.

Every shared library also has a *real name*, which is the name of the actual file in which the library resides. The real name is longer than the soname; it must be formed by appending to the soname a period, and a minor number, and optionally, another period and a release number. The minor number and release number are used for configuration control.

Lastly, the library has a name that is used by the compiler, which is the soname without the version number.

#### Example 1

The *utils* library will have three names:

*libutils.so.1* – This will be its soname.

*libutils.so.1.0.1* – This will be the name of the file. I will use a minor number of 0 and a release number 1

*libutils.so* – This is the name the compiler will use, which we will called the *linker name*.

#### Example 2

If you look in the */lib* directory, you will see that links are created in a specific way; for each shared library there are often at least three entries, such as

```
lrwxrwxrwx 1 root root 11 Aug 12 18:52 libacl.so -> libacl.so.1
lrwxrwxrwx 1 root root 15 Aug 12 18:51 libacl.so.1 -> libacl.so.1.1.0
-rwxr-xr-x 1 root root 31380 Aug 3 18:42 libacl.so.1.1.0
```

Notice that the compiler's name (without the version number) is a soft link to the soname, which is a soft link to the actual library file. When we set up our *libutils* library, we need to do the same thing. Every library will have three files in the directory where it is placed: the soname will be a soft link to the actual library file, and a soft link to the soname file named with the linker name.

### Steps to Create the Library

1. For each source code file that you intend to put into a shared library, say *stuff.c*, compile it with *position independent code* using the following command:

```
gcc -fPIC -g -Wall -c stuff.c
```

   This will produce an object file, *stuff.o*, with debugging information included (the `-g` option), with all warnings enabled ( the -Wall option), which is always a safe thing to do. The `-fPIC` option is what generates the *position independent code* (hence `PIC`). Position independent code is code that can be executed regardless of where it is placed in memory. This is not the same thing as relocatable code. Relocatable code is code that can be placed anywhere into memory with help from a linkage editor or loader. Instructions such as those that specify addresses relative to the program counter are position independent.

2. Suppose that *stuff.o* and *tools.o* are two object files generated in accordance with the first step. To create a shared library containing just those files with soname *libgoodstuff.so.1*, and real file name *libgoodstuff.so.1.0.1*, use the following command:

```
gcc -shared -Wl,-soname,libgoodstuff.so.1 -o libgoodstuff.so.1.0.1  \
     stuff.o  tools.o
```

   This will create the file *libgoodstuff.so.1.0.1* with the soname *libgoodstuff.so.1* stored internally. Note that there cannot be any white space before or after the commas. The `-Wl` option tells `gcc` to pass the remaining comma-separated list to the link editor as options. You might be advised by someone else to use "`-fpic`" instead of "`-fPIC`" because it generates faster code. Do not do so. It is not guaranteed to work in all cases. `-fPIC` generates bigger code but it never fails to work.

3. It is time to install the library in the appropriate place. Unless you have superuser privileges, you will not be able to install your nifty library in a standard location such as */usr/local/lib*. Instead, you will most likely put it in your own *lib* directory, such as *~/lib*. Just copy the file into the directory.

4. After you copy the file into the directory, you should run `ldconfig` on that directory, with a `-n` option, e.g.

```
ldconfig -n ~/lib
```

   `ldconfig`, with the `-n` option, creates the necessary links and cache to the most recent shared libraries found in the given directory. In particular, it will create a symbolic link from a file named with the soname to the actual library file. If there are multiple minor versions or releases, `ldconfig` will link the soname file to the highest-numbered

minor version and release combination. The `-n` option tells `ldconfig` not to make any changes to the standard set of library directories. After `ldconfig` runs in our example, we would have the link

```
libgoodstuff.so.1 -> libgoodstuff.so.1.0.1
```

After running `ldconfig`, you should manually create a link from a file with the linker name to the highest-numbered soname link. In our example, we would type

```
ln -s libgoodstuff.so.1 libgoodstuff.so
```

to create the link

```
libgoodstuff.so -> libgoodstuff.so.1
```

5. If at some future time, you revise the *goodstuff* library, you would increment either the minor version number or the release number, or perhaps even the major version number, if the interface to the library changed. If you just change an algorithm internally or fixed a few bugs, you would not change the major number, only the minor one or the release number. Suppose that you create a new release, *libgoodstuff.so.1.0.2*, with soname *libgoodstuff.1*. You would copy the file into the same directory as the older release and run `ldconfig` again. `ldconfig` would change the link from the soname to the later release. A listing of that directory would then look like

```
libutils.so -> libutils.so.1
libutils.so.1 -> libutils.so.0.2
libutils.so.1.0.1
libutils.so.1.0.2
```

## Using a Shared Library

What you need to understand about how to use shared libraries is that it is a two-step linking process. In the first step, the linkage editor will create some static information in your executable file that will be used later by the dynamic linker at runtime. So both the linkage editor and a dynamic linker participate in creating a working executable.

You link your program to a shared library in the same way that you link it to a static library, using the `-l` option to `gcc`, to name the library to which you want your program linked, and using the `-L`*dir* option to tell it which directory it is in if it is not in a standard location. For example:

```
gcc -o myprogram myprogram.c -L~/lib  -lgoodstuff
```

will create the executable `myprogram`, to be linked dynamically to the library *~/lib/libgoodstuff.so*. We can also write

```
gcc -o myprogram myprogram.c ~/lib/libgoodstuff.so
```

skipping the options `-l` and `-L`. The two methods are equivalent. If *~/lib* is in the `LIBRARY_PATH` environment variable, then you can also write

```
gcc -o myprogram myprogram.c   -lgoodstuff
```

and this will be equivalent as well. All of the above assume that the directory containing the header file is in your `CPATH` or is in a standard location. Otherwise remember to add the option `-I` *`includedir`* to this command.

This is just the first step. Your executable will not run correctly unless the dynamic linker can find you shared library file. One way to tell whether it will run correctly is with the `ldd` command. The `ldd` command prints shared dependencies in a file. Translation: it displays a list of shared libraries upon which your program depends. If `ldd` does not display the path to *~/lib/libgoodstuff.so*, then `myprogram` will fail to find the file and will not run. If the dynamic linker will be able to find my library, the output of `ldd` would look something like:

```
linux-gate.so.1 =>  (0x00a31000)
libgoodstuff.so.1 => ~/lib/libgoodstuff.so.1 (0x00caa000)
libc.so.6 => /lib/libc.so.6 (0x00110000)
/lib/ld-linux.so.2 (0x00bd5000)
```

If it will not be able to find it, I will see

```
linux-gate.so.1 =>  (0x00a31000)
libgoodstuff.so.1 => not found
libc.so.6 => /lib/libc.so.6 (0x00110000)
/lib/ld-linux.so.2 (0x00bd5000)
```

If you had the means to put your shared library file in a standard directory, this problem would be solved easily. Unfortunately, with just user privileges and not superuser privileges, you cannot do this. The easiest solution to this problem is one that is not recommended for various reasons: you can modify the environment variable `LD_LIBRARY_PATH`, which the dynamic linker uses at loadtime and runtime to locate shared libraries. To be precise, the dynamic linker searches the directories in this variable before any in the standard locations. Therefore, you can put the line

```
LD_LIBRARY_PATH=~/lib
export LD_LIBRARY_PATH
```

in your `.bashrc` file to have the dynamic linker search that directory at run time. The alternative is to modify the variable every time you run the program, which is a nuisance I think, or to hard code the path to the libraries into the executable using the `-rpath` option to the linkage editor,which is described in the `ld` man page.

There is one other option. You can define the `LD_RUN_PATH` variable to contain the directory in which you put your libraries, in your `.bashrc file`:

```
LD_RUN_PATH=~/lib
export LD_RUN_PATH
```

If this variable is defined when you compile the executable, then the run path will be hard-coded into the executable and the dynamic linker will find your libraries at run time.

# B

# SYSTEM LIMITS

## About System Limits

We can do a man page search to discover how to find this and other resource limits on processes:

```
$ apropos -a resource limit
--snip--
prlimit (1)          - get and set process resource limits
prlimit (2)          - get/set resource limits
prlimit64 (2)        - get/set resource limits
--snip--
```

The `prlimit` command will display these limits. Its man page states that `prlimit -n` displays the maximum number of open files:

```
$ prlimit -n
RESOURCE DESCRIPTION              SOFT    HARD UNITS
NOFILE   max number of open files 1024 1048576 files
```

The *soft limit* is always smaller than the *hard limit*. The soft limit for a resource is the value enforced by the kernel for a process. When a process tries to exceed its soft limit for a resource, the kernel will prevent it from acquiring it. A process can raise its soft limit up to the value of its hard limit. A process can irreversibly lower its hard limit but not raise it. The Linux-specific `prlimit()` system call can be used for getting and setting these limits. A process can also use the more portable `getrlimit()` and `setrlimit()` calls.

# C

# DATE AND TIME FORMAT SPECIFIERS

The examples listed in this table are based on the date of January 19, 2038 at 03:14:07 UTC, the time at which the 32-bit Unix time representation overflows.

**Table C-1:** Format specifiers for date and time formatting, with examples based on the date January 19, 2038 at 03:14:07 UTC, in the time zone EDT with current locale en_US.UTF-8

| FORMAT | EXAMPLE | MEANING |
|---|---|---|
| %a | Mon | locale's abbreviated weekday name |
| %A | Monday | locale's full weekday name |
| %b | Jan | locale's abbreviated month name |
| %B | January | locale's full month name |
| %c | Mon 18 Jan 2038 10:14:07 PM EST | locale's date and time |
| %C | 20 | century; like %Y, except omit last two digits |
| %d | 18 | day of month |
| %D | 01/18/38 | date; same as %m/%d/%y |
| %e | 18 | day of month, space padded; same as %_d |
| %F | 2038-01-18 | full date; like %+4Y-%m-%d |
| %g | 38 | last two digits of year of ISO week number (see %G) |
| %G | 2038 | year of ISO week number (see %V); normally useful only with %V |
| %h | Jan | same as %b |
| %H | 22 | hour (00..23) |
| %I | 10 | hour (01..12) |
| %j | 18 | day of year (001..366) |
| %k | 22 | hour, space padded ( 0..23); same as %_H |
| %l | 10 | hour, space padded ( 1..12); same as %_I |
| %m | 1 | month (01..12) |
| %M | 14 | minute (00..59) |
| %n | | a newline |
| %N | 0 | nanoseconds (000000000..999999999) |
| %p | PM | locale's equivalent of either AM or PM; blank if not known |
| %P | pm | like %p, but lower case |
| %q | 1 | quarter of year (1..4) |
| %r | 10:14:07 PM | locale's 12-hour clock time |
| %R | 22:14 | 24-hour hour and minute; same as %H:%M |
| %s | 2147483647 | seconds since 1970-01-01 00:00:00 UTC |
| %S | 7 | second (00..60) |
| %T | 22:14:07 | time; same as %H:%M:%S |
| %u | 1 | day of week (1..7); 1 is Monday |
| %U | 3 | week number of year, with Sunday as first day of week (00..53) |
| %V | 3 | ISO week number, with Monday as first day of week (01..53) |
| %w | 1 | day of week (0..6); 0 is Sunday |
| %W | 3 | week number of year, with Monday as first day of week (00..53) |
| %x | 01/18/2038 | locale's date representation |
| %X | 10:14:07 PM | locale's time representation |
| %y | 38 | last two digits of year (00..99) |
| %Y | 2038 | year |
| %z | -500 | +hhmm numeric time zone |
| %Z | EST | alphabetic time zone abbreviation |

# E

# UNICODE AND UTF-8

Computers store all data as sequences of bits that are essentially numbers. Numbers are used torepresent all of the visual symbols that we think of as characters, such as the letters of our alphabets, the digits in our numerals, the various punctuation symbols, and the control codes that affect how and where other symbols are printed. Informally, *characters* are the smallest representable symbols used in a written language, such as the letters of the Roman alphabet, and a *character encoding* is an assignment of numbers to a set of characters.

The *ASCII* character encoding was the most prevalent encoding for more than forty years. ASCII is the acronym for *American Standard Code for Information Interchange*. The ASCII encoding maps characters to 7-bit integers, using the range from 0 to 127 to represent 94 printing characters, 33 control characters, and the space. Since a byte is usually used to store a character, the eighth bit of the byte is filled with a 0. Well before the ASCII encoding was defined, IBM defined a different encoding named *EBCDIC*, the acronym for *Extended Binary Coded Decimal Interchange*. That encoding as-

signed an entirely different set of eight-bit numbers to the same characters assigned by the ASCII encoding. The existence of two different encodings of the same set of characters required programs to be aware of which encoding was used and to convert from one to the other.

One problem with both the ASCII and EBCDIC codes is that they do not provide a way to encode characters from other scripts, such as Cyrillic or Greek. It does not even have encodings of Roman characters with diacritical marks, such as e, a, s, or ó. Over time, as computer usage extended worldwide, other encodings for different alphabets and scripts were developed, usually with overlapping codes. These encoding systems conflicted with one another. That is, two encodings could use the same number for two different characters, or use different numbers for the same character. A program transferring text from one computer to another would run the risk that the text would be corrupted in the transfer.

## Background

In 1989, to overcome these problems, the International Standards Organization (ISO) started work on a universal, all-encompassing character code standard, and in 1990 they published a draft standard (*ISO 10646*) called the *Universal Character Set* (*UCS*). UCS was designed as a superset of all other character set standards, providing round-trip compatibility to other character sets. *Round-trip compatibility* asserts that no information is lost if a text string is converted to UCS and then back to its original encoding.

Simultaneously, the Unicode Project, which was a consortium of private industrial partners, was working on its own, independent universal character encoding. In 1991, the Unicode Project and ISO decided to work cooperatively to avoid creating two different character encodings. The result was that the code table created by the Unicode Consortium, as they are now called, satisfied the original *ISO 10646* standard. Over time, the two groups continued to modify the respective standards, but they always remain compatible. Unicode adds new characters over time, but it always contains the character set defined by *ISO 10646-x*. The latest Unicode standard as of this writing is *Unicode 15.0.0*.

## Terminology

The Unicode Consortium defines a *character* as an abstract representation of the smallest element of a written language that has semantic value. The actual appearance or form of a character is called a *glyph*. The letter 'a', for example, is drawn using one of many possible fonts and so its actual form can vary, but it's still an 'a'. Each different way to render the letter 'a' is a different glyph. Glyphs are the shapes that characters take. Character encodings assign numbers to characters, not to glyphs.

The set of all characters used together in a written, natural language is called a *script*, not to be confused with the use of the term *script* as a type of program. For example, the characters in the Greek language constitute the

Greek script, and the characters used in most of Western Europe are part of the Latin script.

The set of numbers that are assigned to all of the characters in a script is called its *codespace*. The codespace for Greek, for example, is the set of integers from decimal 880 through 1023, or hexadecimal 0370 through 03*FF*. An individual number in a codespace is called a *code point*.

In Unicode, a code point is denoted by "U+" following by a hexadecimal number from 4 to 8 digits long. For example, the code point assigned to the Greek character $\psi$ is U+03C8 and the one assigned to $\phi$ is U+03C6. Most of the code points in use are 4 digits long. When a character has been assigned a code point, it's called an *encoded character*.

## Unicode

Unicode contains the alphabets of almost all known languages, as diverse as Japanese, Chinese, Greek, Cyrillic, Canadian Aboriginal, and Arabic. It was originally a 16-bit character set, but in 1995, with Unicode 2.0, it became 32 bits. The Unicode Standard encodes characters in the range U+0000..U+10FFFF, which is roughly a 21-bit code space. The code reserves the remaining values for future use.

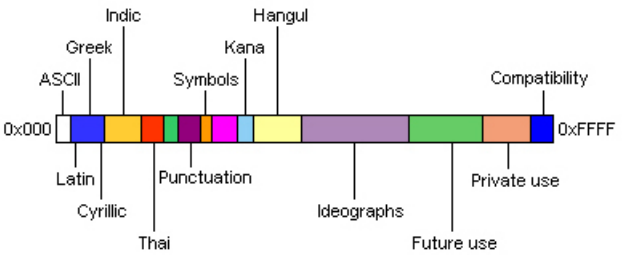



*Figure E-1: Unicode layout*

## UTF-8

Unicode code points are just numeric values assigned to characters. They are not representations of characters as sequences of bytes. For example, the code point U+03C6 is not a sequence of two bytes containing 0x03 and 0xC6. If we were to just use the number's ordinary byte representation, to encode the character, there would be no way to distinguish the sequence of two characters `\f$` (form feed followed by $) from the Greek character $\phi$.

The mapping of code points to sequences of bytes is called a *character encoding form*. Because the ordering of bytes in a particular computer system can vary, such as whether it is big-endian or little-endian, the Unicode Consortium defines a *character encoding scheme* as a character encoding form together with a specification of the way in which the bytes are sequenced.

There are Unicode several character encoding schemes, including UCS-2, UCS-4, UTF-2, UTF-4, UTF-8, UTF-16, and UTF-32. UCS-2 and UCS-4

encode Unicode text as sequences of either two or four bytes, but these cannot work in a Unix system because strings with these encodings can contain bytes that match ASCII characters and in particular, `\0` or `/`, which have a special meaning in filenames and other C library function parameters. Unix file systems and tools expect ASCII characters and would fail if they were given two-byte encodings.

The most prevalent encoding of Unicode as sequences of bytes is UTF-8, invented by Ken Thompson in 1992. In UTF-8, characters are encoded with anywhere from one to six bytes. In other words, the number of bytes varies with the character. In UTF-8, all ASCII characters are encoded within the 7 least significant bits of a byte whose most significant bit is 0.

UTF-8 uses the following scheme for encoding Unicode code points:

1. Characters U+0000 to U+007F ( i.e., the ASCII characters) are encoded simply as bytes 0x00 to 0x7F. This implies that files and strings that contain only 7-bit ASCII characters have the same encoding under both ASCII and UTF-8.
2. All characters larger than U+007F are encoded as a sequence of two or more bytes, each of which has the most significant bit set. This means that no ASCII byte can appear as part of any other character, because ASCII characters are the only characters whose leading bit is 0.
3. The first byte of a multibyte sequence that represents a non-ASCII character is always in the range 0xC0 to 0xFD and it indicates how many bytes follow for this character. Specifically it is one of `110xxxxx`, `1110xxxx`, `11110xxx`, `111110xx`, and `1111110x`, where the x's may be 0's or 1's. The number of 1-bits following the first 1-bit up until the next 0-bit is the number of bytes in the rest of the sequence. Thus, `1110xxxx` indicates that two bytes follow.
   All further bytes in a multibyte sequence start with the two bits 10 and are in the range 0x80 to 0xBF. This implies that UTF-8 sequences must be of the following forms in binary, where the x's represent the bits from the code point, with the leftmost x-bit being its most significant bit:

```
0xxxxxxx
110xxxxx  10xxxxxx
1110xxxx  10xxxxxx  10xxxxxx
11110xxx  10xxxxxx  10xxxxxx  10xxxxxx
111110xx  10xxxxxx  10xxxxxx  10xxxxxx  10xxxxxx
1111110x  10xxxxxx  10xxxxxx  10xxxxxx  10xxxxxx  10xxxxxx
```

4. The bytes 0xFE and 0xFF are never used in the UTF-8 encoding.

A few things can be concluded from the above rules. First, the number of x's in a sequence is the maxiumum number of bits that a code point can have to be to be representable in that many bytes. For example, there are 11 x-bits in a two-byte UTF-8 sequence, so all code points whose 16-bit binary value

is at least 0000000010000000 but at most 0000011111111111 can be encoded using two bytes. In hexadecimal, these lie between 0080 and 07FF. Table E-1 shows the ranges of Unicode code points that map to the different UTF-8 sequence lengths.

**Table E-1:** Code point ranges in Unicode 15.0.0

| Number of Bytes | Number of bits in Code Point | Range |
|---|---|---|
| 1 | 7 | 00000000 - 0000007F |
| 2 | 11 | 00000080 - 000007FF |
| 3 | 16 | 00000800 - 0000FFFF |
| 4 | 21 | 00001000 - 001FFFFF |
| 5 | 26 | 00200000 - 03FFFFFF |
| 6 | 31 | 04000000 - FFFFFFFF |

You can see that, although UTF-8 encoded characters may be up to six bytes long in theory, code points up to U+FFFF, having at most 16 bits, can be encoded in sequences of at most three bytes.

Converting a Unicode code point to UTF-8 by hand is straightforward using the above table.

1. From the range, determine how many bytes are needed.
2. Starting with the least significant bit, copy bits from the code point from right to left into the least significant byte.
3. When the current byte has reached eight bits, continue filling the next most significant byte with successively more significant bits from the code point.
4. Repeat until all bits have been copied into the byte sequence, filling with leading zeros as required.

### *Example 1*

To convert U+05E7 to UTF-8, we first observe that it is in the interval 0080 to 07FF, which requires two bytes. We write it in binary as

```
0000 0101 1110 0111
```

The rightmost six bits 100111 are placed into the rightmost byte after a leading two-bit sequence 10:

```
10 100111
```

and the next least significant five bits 10111 are placed into the next byte after a leading three-bit sequence 110:

```
110 10111
```

Therefore, the two-byte sequence is

```
11010111 10100111 = 0xD7 0xA7
```

which is the decimal 215 in the upper byte and 167 in the lower byte.

### *Example 2*

To convert U+0ABC to UTF-8, we observe that it is greater than U+07FF and therefore it requires a three-byte code. In binary, its value is

```
0000 1010 1011 1100
```

Following the procedure, the rightmost six bits are placed into the rightmost byte after a leading `01`. The next six bits are placed into the middle byte after a leading `01`. The remaining bits are all zeros, so the leftmost byte is filled with zeros after a leading `1110`. The resulting bytes are, from most significant to least:

```
11100000
10101010
10111100
```

The sequence `11100000 10101010 10111100` in hexadecimal is `0xE0 0xAA 0xBC`, which in decimal is 224 170 188, the Gujarati sign Nukta.

# G

# SOLUTIONS TO SELECTED EXERCISES

Following are solutions to selected exercises with brief explanations.

## Chapter 1

1. From the AUTHORS section of the man page for `bash` the two authors are Brian Fox and Chet Ramey.

5. The `stat` command displays most of a file's properties, among which is its "birth date".

## Chapter 2

1. The `shuf` command has a `-i<input range>` option. For example, **`shuf -i1-10`** outputs the numbers 1 through 10 in random order. Therefore the answer is **`shuf -i1-100 > permutation100`** or even better **`shuf -i1-100 >| permutation100`** in case the file exists.

4. **`touch shopping_list ; chmod 600 shopping_list`**

7. The following script solves the problem.

```
#!/bin/bash
for name in * ; do
```

```
    cp $name ${name}.copy
done
```

# BIBLIOGRAPHY


[1] Stephen R. Bourne. The UNIX shell. *The Bell System Technical Journal*, 57(6, Part 2):1971–1990, July 1978.

[2] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., Sebastopol, CA, USA, 2006.

[3] Rémy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the Second Extended Filesystem. In *Proceedings of the First Dutch International Symposium on Linux*, University of Groningen, Netherlands, 1995.

[4] The C Standards Committee. Iso/iec 9899:2018, also known as c17, 2017.

[5] C. DiBona and S. Ockman. *Open Sources: Voices from the Open Source Revolution*. O'Reilly Media, 1999.

[6] Robert. Grudin. *Time and the Art of Living*. Harper & Row Cambridge [Mass], 1st edition, 1982.

[7] G. Haff. *How Open Source Ate Software: Understand the Open Source Movement and So Much More*. Apress, 2018.

[8] G. Irlam. Unix file size survey-1993. *http://www.base.com/gordoni/ufs93.html*, 1994.

[9] S. C. Johnson and D. M. Ritchie. unix time-sharing system: Portability of C programs and the unix system. *Bell Sys. Tech. J.*, 57(6):2021–2048, 1978.

[10] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[11] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall Software Series, Englewood Cliffs, N.J., USA, 1984.

[12] B.W. Kernighan. *Unix: A History and a Memoir*. Independently Published, 2019.

[13] Michael S. Mahoney. An oral history of UNIX.

[14] Jim Mauro and Richard McDougall. *Solaris Internals*. Sun Microsystems Press, Palo Alto, CA, USA, 1st edition, 2001.

[15] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quartermain. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, Inc., New York, NY, USA, 1996.

[16] Bruce Molay. *Understanding UNIX/LINUX Programming: A Guide to Theory and Practice*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2002.

[17] Institute of Electrical and Electronics Engineers and The Open Group. IEEE Std 1003.1-2017, 2018 edition, 2018.

[18] Steve D. Pate. *UNIX Internals: A Practical Approach*. Addison-Wesley Longman Publishing Co., Inc., USA, 1996.

[19] Chet Ramey and Brian Fox. The GNU Bash reference manual.

[20] D. M. Ritchie. Unix time-sharing system: A retrospective. *Bell Sys. Tech. J.*, 57(6):1947–1969, 1978. Also in *Proc. Hawaii International Conference on Systems Science,* Honolulu, Hawaii, Jan. 1977.

[21] D. M. Ritchie. The development of the C language. In *Proceedings, ACM History of Programming Languages II*, Cambridge, MA, April 1993.

[22] D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan. Unix time-sharing system: The C programming language. *Bell Sys. Tech. J.*, 57(6):1991–2019, 1978.

[23] Dennis Ritchie. The evolution of the Unix time-sharing system. In *Proceedings of a Symposium on Language Design and Programming Methodology*, page 25‑36, Berlin, Heidelberg, 1979. Springer-Verlag.

[24] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.

[25] Peter H. Salus. *A Quarter Century of UNIX*. Addison-Wesley, Reading, MA, 1994.

[26] Peter H. Salus and Jeremy C. Reed. *The Daemon, the Gnu, and the Penguin*. Reed Media Services, 2008.

[27] Abraham Silberschatz, Greg Gagne, and Peter B. Galvin. *Operating System Concepts*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 10th edition, 2018.

[28] Richard M. Stallman. Linux and the GNU system.

[29] Andrew S. Tanenbaum. *Modern Operating Systems*. Pearson, LLC., London, England, 4th edition, 2014.

[30] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. Prentice-Hall, Inc., Englewood Cliffs, NJ, USA, 3rd edition, 2006.

[31] K. Thompson and D. M. Ritchie. *The UNIX Programmer's Manual*. Bell Telephone Laboratories, Short Hills, N.J., USA, 2nd edition, june 1972.

[32] Ken Thompson. UNIX implementation. *The Bell System Technical Journal*, 57(6, Part 2):1931–1945, July 1978.

[33] Linus Torvalds and David M. Diamond. *Just for Fun: The Story of an Accidental Revolutionary*. Harper Collins, New York, NY, 2001.

[34] David A. Wheeler. Secure programming for Linux and Unix HOWTO, 2003.