Advanced Linux Programming

Contents At a Glance

Advanced Linux Programming

Mark Mitchell, Jeffrey Oldham,
and Alex Samuel

New Riders
www.newriders.com
201 West 103rd Street, Indianapolis, Indiana 46290
An Imprint of Pearson Education
Boston • Indianapolis • London • Munich • New York • San Francisco

Advanced Linux Programming

FIRST EDITION: June, 2001

International Standard Book Number: 0-7357-1043-0

Library of Congress Catalog Card Number: *00-105343*

05 04 03 02 01 7 6 5 4 3 2 1

Interpretation of the printing code: The rightmost double-digit number is the year of the book's printing; the right-most single-digit number is the number of the book's printing. For example, the printing code 01-1 shows that the first printing of the book occurred in 2001.

Composed in Bembo and MCPdigital by New Riders Publishing.

Printed in the United States of America.

Trademarks

Warning and Disclaimer

Publisher
David Dwyer

Associate Publisher
Al Valvano

Executive Editor
Stephanie Wall

Managing Editor
Gina Brown

Acquisitions Editor
Ann Quinn

Development Editor
Laura Loveall

Product Marketing Manager
Stephanie Layton

Publicity Manager
Susan Petro

Project Editor
Caroline Wise

Copy Editor
Krista Hansing

Senior Indexer
Cheryl Lenser

Manufacturing Coordinator
Jim Conway

Book Designer
Louisa Klucznik

Cover Designer
Brainstorm Design, Inc.

Cover Production
Aren Howell

Proofreader
Debra Neel

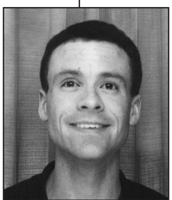
Composition
Amy Parker

Table of Contents

Table of Program Listings

About the Authors

Mark Mitchell received a bachelor of arts degree in computer science from Harvard in 1994 and a master of science degree from Stanford in 1999. His research interests centered on computational complexity and computer security. Mark has participated substantially in the development of the GNU Compiler Collection, and he has a strong interest in developing quality software.



Jeffrey Oldham received a bachelor of arts degree in computer science from Rice University in 1991. After working at the Center for Research on Parallel Computation, he obtained a doctor of philosophy degree from Stanford in 2000. His research interests center on algorithm engineering, concentrating on flow and other combinatorial algorithms. He works on GCC and scientific computing software.

Alex Samuel graduated from Harvard in 1995 with a degree in physics. He worked as a software engineer at BBN before returning to study physics at Caltech and the Stanford Linear Accelerator Center. Alex administers the Software Carpentry project and works on various other projects, such as optimizations in GCC.

Mark and Alex founded **CodeSourcery LLC** together in 1999. Jeffrey joined the company in 2000. CodeSourcery's mission is to provide development tools for GNU/Linux and other operating systems; to make the GNU tool chain a commercial-quality, standards-conforming development tool set; and to provide general consulting and engineering services. CodeSourcery's Web site is `http://www.codesourcery.com`.

About the Technical Reviewers

These reviewers contributed their considerable hands-on expertise to the entire development process for *Advanced Linux Programming*. As the book was being written, these dedicated professionals reviewed all the material for technical content, organization, and flow. Their feedback was critical to ensuring that *Advanced Linux Programming* fits our reader's need for the highest quality technical information.

Glenn Becker has many degrees, all in theatre. He presently works as an online producer for SCIFI.COM, the online component of the SCI FI channel, in New York City. At home he runs Debian GNU/Linux and obsesses about such topics as system administration, security, software internationalization, and XML.

John Dean received a BSc(Hons) from the University of Sheffield in 1974, in pure science. As an undergraduate at Sheffield, John developed his interest in computing. In 1986 he received a MSc from Cranfield Institute of Science and Technology in Control Engineering. While working for Roll Royce and Associates, John became involved in developing control software for computer-aided inspection equipment of nuclear steam-raising plants. Since leaving RR&A in 1978, he has worked in the petrochemical industry developing and maintaining process control software. John worked a volunteer software developer for MySQL from 1996 until May 2000, when he joined MySQL as a full-time employee. John's area of responsibility is MySQL on MS Windows and developing a new MySQL GUI client using Trolltech's Qt GUI application toolkit on both Windows and platforms that run X-11.

Acknowledgments

We greatly appreciate the pioneering work of Richard Stallman, without whom there would never have been the GNU Project, and of Linus Torvalds, without whom there would never have been the Linux kernel. Countless others have worked on parts of the GNU/Linux operating system, and we thank them all.

We thank the faculties of Harvard and Rice for our undergraduate educations, and Caltech and Stanford for our graduate training. Without all who taught us, we would never have dared to teach others!

W. Richard Stevens wrote three excellent books on UNIX programming, and we have consulted them extensively. Roland McGrath, Ulrich Drepper, and many others wrote the GNU C library and its outstanding documentation.

Robert Brazile and Sam Kendall reviewed early outlines of this book and made wonderful suggestions about tone and content. Our technical editors and reviewers (especially Glenn Becker and John Dean) pointed out errors, made suggestions, and provided continuous encouragement. Of course, any errors that remain are no fault of theirs!

Thanks to Ann Quinn, of New Riders, for handling all the details involved in publishing a book; Laura Loveall, also of New Riders, for not letting us fall too far behind on our deadlines; and Stephanie Wall, also of New Riders, for encouraging us to write this book in the first place!

Tell Us What You Think

As the reader of this book, you are the most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As the Executive Editor for the Web Development team at New Riders Publishing, I welcome your comments. You can fax, email, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author, as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

Fax: 317-581-4663

Email: `Stephanie.Wall@newriders.com`

Mail: Stephanie Wall
Executive Editor
New Riders Publishing
201 West 103rd Street
Indianapolis, IN 46290 USA

Introduction

GNU/Linux has taken the world of computers by storm. At one time, personal computer users were forced to choose among proprietary operating environments and applications. Users had no way of fixing or improving these programs, could not look "under the hood," and were often forced to accept restrictive licenses. GNU/Linux and other open source systems have changed that—now PC users, administrators, and developers can choose a free operating environment complete with tools, applications, and full source code.

A great deal of the success of GNU/Linux is owed to its open source nature. Because the source code for programs is publicly available, everyone can take part in development, whether by fixing a small bug or by developing and distributing a complete major application. This opportunity has enticed thousands of capable developers worldwide to contribute new components and improvements to GNU/Linux, to the point that modern GNU/Linux systems rival the features of any proprietary system, and distributions include thousands of programs and applications spanning many CD-ROMs or DVDs.

The success of GNU/Linux has also validated much of the UNIX philosophy. Many of the application programming interfaces (APIs) introduced in AT&T and BSD UNIX variants survive in Linux and form the foundation on which programs are built. The UNIX philosophy of many small command line-oriented programs working together is the organizational principle that makes GNU/Linux so powerful. Even when these programs are wrapped in easy-to-use graphical user interfaces, the underlying commands are still available for power users and automated scripts.

A powerful GNU/Linux application harnesses the power of these APIs and commands in its inner workings. GNU/Linux's APIs provide access to sophisticated features such as interprocess communication, multithreading, and high-performance networking. And many problems can be solved simply by assembling existing commands and programs using simple scripts.

GNU and Linux

Where did the name GNU/Liux come from? You've certainly heard of Linux before, and you may have heard of the GNU Project. You may not have heard the name GNU/Linux, although you're probably familiar with the system it refers to.

Linux is named after Linus Torvalds, the creator and original author of the *kernel* that runs a GNU/Linux system. The kernel is the program that performs the most basic functions of an operating system: It controls and interfaces with the computer's hardware, handles allocation of memory and other resources, allows multiple programs to run at the same time, manages the file system, and so on.

The kernel by itself doesn't provide features that are useful to users. It can't even provide a simple prompt for users to enter basic commands. It provides no way for users to manage or edit files, communicate with other computers, or write other programs. These tasks require the use of a wide array of other programs, including command shells, file utilities, editors, and compilers. Many of these programs, in turn, use libraries of general-purpose functions, such as the library containing standard C library functions, which are not included in the kernel.

On GNU/Linux systems, many of these other programs and libraries are software developed as part of the GNU Project.[1] A great deal of this software predates the Linux kernel. The aim of the GNU Project is "to develop a complete UNIX-like operating system which is free software" (from the GNU Project Web site, `http://www.gnu.org`).

The Linux kernel and software from the GNU Project has proven to be a powerful combination. Although the combination is often called "Linux" for short, the complete system couldn't work without GNU software, any more than it could operate without the kernel. For this reason, throughout this book we'll refer to the complete system as GNU/Linux, except when we are specifically talking about the Linux kernel.

The GNU General Public License

The source code contained in this book is covered by the GNU *General Public License* (*GPL*), which is listed in Appendix F, "GNU General Public License." A great deal of free software, especially GNU/Linux software, is licensed under it. For instance, the Linux kernel itself is licensed under the GPL, as are many other GNU programs and libraries you'll find in GNU/Linux distributions. If you use the source code in this book, be sure to read and understand the terms of the GPL.

The GNU Project Web site includes an extensive discussion of the GPL (`http://www.gnu.org/copyleft/`) and other free software licenses. You can find information about open source software licenses at `http://www.opensource.org/licenses/index.html`.

Who Should Read This Book?

This book is intended for three types of readers:

- You might be a developer already experienced with programming for the GNU/Linux system, and you want to learn about some of its advanced features and capabilities. You might be interested in writing more sophisticated programs with features such as multiprocessing, multithreading, interprocess communication, and interaction with hardware devices. You might want to improve your programs by making them run faster, more reliably, and more securely, or by designing them to interact better with the rest of the GNU/Linux system.

1. GNU is a recursive acronym: It stands for "GNU's Not UNIX."

- You might be a developer experienced with another UNIX-like system who's interested in developing GNU/Linux software, too. You might already be familiar with standard APIs such as those in the POSIX specification. To develop GNU/Linux software, you need to know the peculiarities of the system, its limitations, additional capabilities, and conventions.
- You might be a developer making the transition from a non-UNIX environment, such as Microsoft's Win32 platform. You might already be familiar with the general principles of writing good software, but you need to know the specific techniques that GNU/Linux programs use to interact with the system and with each other. And you want to make sure your programs fit naturally into the GNU/Linux system and behave as users expect them to.

This book is not intended to be a comprehensive guide or reference to all aspects of GNU/Linux programming. Instead, we'll take a tutorial approach, introducing the most important concepts and techniques, and giving examples of how to use them. Section 1.5, "Finding More Information," in Chapter 1, "Getting Started," contains references to additional documentation, where you can obtain complete details about these and other aspects of GNU/Linux programming.

Because this is a book about advanced topics, we'll assume that you are already familiar with the C programming language and that you know how to use the standard C library functions in your programs. The C language is the most widely used language for developing GNU/Linux software; most of the commands and libraries that we discuss in this book, and most of the Linux kernel itself, are written in C.

The information in this book is equally applicable to C++ programs because that language is roughly a superset of C. Even if you program in another language, you'll find this information useful because C language APIs and conventions are the *lingua franca* of GNU/Linux.

If you've programmed on another UNIX-like system platform before, chances are good that you already know your way around Linux's low-level I/O functions (`open`, `read`, `stat`, and so on). These are different from the standard C library's I/O functions (`fopen`, `fprintf`, `fscanf`, and so on). Both are useful in GNU/Linux programming, and we use both sets of I/O functions throughout this book. If you're not familiar with the low-level I/O functions, jump to the end of the book and read Appendix B, "Low-Level I/O," before you start Chapter 2, "Writing Good GNU/Linux Software."

This book does not provide a general introduction to GNU/Linux systems. We assume that you already have a basic knowledge of how to interact with a GNU/Linux system and perform basic operations in graphical and command-line environments. If you're new to GNU/Linux, start with one of the many excellent introductory books, such as Michael Tolber's *Inside Linux* (New Riders Publishing, 2001).

Conventions

This book follows a few typographical conventions:

- A new term is set in *italics* the first time it is introduced.
- Program text, functions, variables, and other "computer language" are set in a fixed-pitch font—for example, `printf ("Hello, world!\bksl n")`.
- Names of commands, files, and directories are also set in a fixed-pitch font—for example, `cd /`.
- When we show interactions with a command shell, we use `%` as the shell prompt (your shell is probably configured to use a different prompt). Everything after the prompt is what you type, while other lines of text are the system's response.

  For example, in this interaction

  ```
  % uname
  Linux
  ```

  the system prompted you with `%`. You entered the `uname` command. The system responded by printing `Linux`.
- The title of each source code listing includes a filename in parentheses. If you type in the listing, save it to a file by this name. You can also download the source code listings from the *Advanced Linux Programming* Web site (`http://www.newriders.com` or `http://www.advancedlinuxprogramming.com`).

We wrote this book and developed the programs listed in it using the Red Hat 6.2 distribution of GNU/Linux. This distribution incorporates release 2.2.14 of the Linux kernel, release 2.1.3 of the GNU C library, and the EGCS 1.1.2 release of the GNU C compiler. The information and programs in this book should generally be applicable to other versions and distributions of GNU/Linux as well, including 2.4 releases of the Linux kernel and 2.2 releases of the GNU C library.

I

Advanced UNIX Programming with Linux

1 Getting Started

THIS CHAPTER SHOWS YOU HOW TO PERFORM THE BASIC steps required to create a C or C++ Linux program. In particular, this chapter shows you how to create and modify C and C++ source code, compile that code, and debug the result. If you're already accustomed to programming under Linux, you can skip ahead to Chapter 2, "Writing Good GNU/Linux Software;" pay careful attention to Section 2.3, "Writing and Using Libraries," for information about static versus dynamic linking that you might not already know.

Throughout this book, we'll assume that you're familiar with the C or C++ programming languages and the most common functions in the standard C library. The source code examples in this book are in C, except when demonstrating a particular feature or complication of C++ programming. We also assume that you know how to perform basic operations in the Linux command shell, such as creating directories and copying files. Because many Linux programmers got started programming in the Windows environment, we'll occasionally point out similarities and contrasts between Windows and Linux.

1.1 Editing with Emacs

An *editor* is the program that you use to edit source code. Lots of different editors are available for Linux, but the most popular and full-featured editor is probably GNU Emacs.

> **About Emacs**
>
> Emacs is much more than an editor. It is an incredibly powerful program, so much so that at CodeSourcery, it is affectionately known as the One True Program, or just the OTP for short. You can read and send email from within Emacs, and you can customize and extend Emacs in ways far too numerous to discuss here. You can even browse the Web from within Emacs!

If you're familiar with another editor, you can certainly use it instead. Nothing in the rest of this book depends on using Emacs. If you don't already have a favorite Linux editor, then you should follow along with the mini-tutorial given here.

If you like Emacs and want to learn about its advanced features, you might consider reading one of the many Emacs books available. One excellent tutorial, *Learning GNU Emacs*, is written by Debra Cameron, Bill Rosenblatt, and Eric S. Raymond (O'Reilly, 1996).

1.1.1 Opening a C or C++ Source File

You can start Emacs by typing `emacs` in your terminal window and pressing the Return key. When Emacs has been started, you can use the menus at the top to create a new source file. Click the Files menu, choose Open Files, and then type the name of the file that you want to open in the "minibuffer" at the bottom of the screen.[1] If you want to create a C source file, use a filename that ends in `.c` or `.h`. If you want to create a C++ source file, use a filename that ends in `.cpp`, `.hpp`, `.cxx`, `.hxx`, `.C`, or `.H`. When the file is open, you can type as you would in any ordinary word-processing program. To save the file, choose the Save Buffer entry on the Files menu. When you're finished using Emacs, you can choose the Exit Emacs option on the Files menu.

If you don't like to point and click, you can use keyboard shortcuts to automatically open files, save files, and exit Emacs. To open a file, type `C-x C-f`. (The `C-x` means to hold down the Control key and then press the `x` key.) To save a file, type `C-x C-s`. To exit Emacs, just type `C-x C-c`. If you want to get a little better acquainted with Emacs, choose the Emacs Tutorial entry on the Help menu. The tutorial provides you with lots of tips on how to use Emacs effectively.

1. If you're not running in an X Window system, you'll have to press F10 to access the menus.

1.1.2 Automatic Formatting

If you're accustomed to programming in an *Integrated Development Environment (IDE)*, you'll also be accustomed to having the editor help you format your code. Emacs can provide the same kind of functionality. If you open a C or C++ source file, Emacs automatically figures out that the file contains source code, not just ordinary text. If you hit the Tab key on a blank line, Emacs moves the cursor to an appropriately indented point. If you hit the Tab key on a line that already contains some text, Emacs indents the text. So, for example, suppose that you have typed in the following:

```
int main ()
{
printf ("Hello, world\n");
}
```

If you press the Tab key on the line with the call to `printf`, Emacs will reformat your code to look like this:

```
int main ()
{
  printf ("Hello, world\n");
}
```

Notice how the line has been appropriately indented.

As you use Emacs more, you'll see how it can help you perform all kinds of complicated formatting tasks. If you're ambitious, you can program Emacs to perform literally any kind of automatic formatting you can imagine. People have used this facility to implement Emacs modes for editing just about every kind of document, to implement games[2], and to implement database front ends.

1.1.3 Syntax Highlighting

In addition to formatting your code, Emacs can make it easier to read C and C++ code by coloring different syntax elements. For example, Emacs can turn keywords one color, built-in types such as `int` another color, and comments another color. Using color makes it a lot easier to spot some common syntax errors.

The easiest way to turn on colorization is to edit the file `~/.emacs` and insert the following string:

```
(global-font-lock-mode t)
```

Save the file, exit Emacs, and restart. Now open a C or C++ source file and enjoy!

You might have noticed that the string you inserted into your `.emacs` looks like code from the LISP programming language. That's because it *is* LISP code! Much of Emacs is actually written in LISP. You can add functionality to Emacs by writing more LISP code.

2. Try running the command `M-x dunnet` if you want to play an old-fashioned text adventure game.

1.2 Compiling with GCC

A *compiler* turns human-readable source code into machine-readable object code that can actually run. The compilers of choice on Linux systems are all part of the GNU Compiler Collection, usually known as GCC.[3] GCC also include compilers for C, C++, Java, Objective-C, Fortran, and Chill. This book focuses mostly on C and C++ programming.

Suppose that you have a project like the one in Listing 1.2 with one C++ source file (`reciprocal.cpp`) and one C source file (`main.c`) like in Listing 1.1. These two files are supposed to be compiled and then linked together to produce a program called `reciprocal`.[4] This program will compute the reciprocal of an integer.

Listing 1.1 **(*main.c*) C source file—*main.c***

```
#include <stdio.h>
#include "reciprocal.hpp"

int main (int argc, char **argv)
{
  int i;

  i = atoi (argv[1]);
  printf ("The reciprocal of %d is %g\n", i, reciprocal (i));
  return 0;
}
```

Listing 1.2 **(*reciprocal.cpp*) C++ source file—*reciprocal.cpp***

```
#include <cassert>
#include "reciprocal.hpp"

double reciprocal (int i) {
  // I should be non-zero.
  assert (i != 0);
  return 1.0/i;
}
```

3. For more information about GCC, visit `http://gcc.gnu.org`.

4. In Windows, executables usually have names that end in `.exe`. Linux programs, on the other hand, usually have no extension. So, the Windows equivalent of this program would probably be called `reciprocal.exe`; the Linux version is just plain `reciprocal`.

There's also one header file called `reciprocal.hpp` (see Listing 1.3).

Listing 1.3 (*reciprocal.hpp*) Header file—*reciprocal.hpp*

```
#ifdef __cplusplus
extern "C" {
#endif

extern  double reciprocal (int i);

#ifdef __cplusplus
}
#endif
```

The first step is to turn the C and C++ source code into object code.

1.2.1 Compiling a Single Source File

The name of the C compiler is `gcc`. To compile a C source file, you use the `-c` option. So, for example, entering this at the command prompt compiles the `main.c` source file:

```
% gcc -c main.c
```

The resulting object file is named `main.o`.

The C++ compiler is called `g++`. Its operation is very similar to `gcc`; compiling `reciprocal.cpp` is accomplished by entering the following:

```
% g++ -c reciprocal.cpp
```

The `-c` option tells `g++` to compile the program to an object file only; without it, `g++` will attempt to link the program to produce an executable. After you've typed this command, you'll have an object file called `reciprocal.o`.

You'll probably need a couple other options to build any reasonably large program. The `-I` option is used to tell GCC where to search for header files. By default, GCC looks in the current directory and in the directories where headers for the standard libraries are installed. If you need to include header files from somewhere else, you'll need the `-I` option. For example, suppose that your project has one directory called `src`, for source files, and another called `include`. You would compile `reciprocal.cpp` like this to indicate that `g++` should use the `../include` directory in addition to find `reciprocal.hpp`:

```
% g++ -c -I ../include reciprocal.cpp
```

Sometimes you'll want to define macros on the command line. For example, in production code, you don't want the overhead of the assertion check present in `reciprocal.cpp`; that's only there to help you debug the program. You turn off the check by defining the macro `NDEBUG`. You could add an explicit `#define` to `reciprocal.cpp`, but that would require changing the source itself. It's easier to simply define `NDEBUG` on the command line, like this:

```
% g++ -c -D NDEBUG reciprocal.cpp
```

If you had wanted to define `NDEBUG` to some particular value, you could have done something like this:

```
% g++ -c -D NDEBUG=3 reciprocal.cpp
```

If you're really building production code, you probably want to have GCC optimize the code so that it runs as quickly as possible. You can do this by using the `-O2` command-line option. (GCC has several different levels of optimization; the second level is appropriate for most programs.) For example, the following compiles `reciprocal.cpp` with optimization turned on:

```
% g++ -c -O2 reciprocal.cpp
```

Note that compiling with optimization can make your program more difficult to debug with a debugger (see Section 1.4, "Debugging with GDB"). Also, in certain instances, compiling with optimization can uncover bugs in your program that did not manifest themselves previously.

You can pass lots of other options to `gcc` and `g++`. The best way to get a complete list is to view the online documentation. You can do this by typing the following at your command prompt:

```
% info gcc
```

1.2.2 Linking Object Files

Now that you've compiled `main.c` and `utilities.cpp`, you'll want to link them. You should always use `g++` to link a program that contains C++ code, even if it also contains C code. If your program contains only C code, you should use `gcc` instead. Because this program contains both C and C++, you should use `g++`, like this:

```
% g++ -o reciprocal main.o reciprocal.o
```

The `-o` option gives the name of the file to generate as output from the link step. Now you can run `reciprocal` like this:

```
% ./reciprocal 7
The reciprocal of 7 is 0.142857
```

As you can see, `g++` has automatically linked in the standard C runtime library containing the implementation of `printf`. If you had needed to link in another library (such as a graphical user interface toolkit), you would have specified the library with the `-l` option. In Linux, library names almost always start with `lib`. For example, the Pluggable Authentication Module (PAM) library is called `libpam.a`. To link in `libpam.a`, you use a command like this:

```
% g++ -o reciprocal main.o reciprocal.o -lpam
```

The compiler automatically adds the `lib` prefix and the `.a` suffix.

As with header files, the linker looks for libraries in some standard places, including the `/lib` and `/usr/lib` directories that contain the standard system libraries. If you want the linker to search other directories as well, you should use the `-L` option, which is the parallel of the `-I` option discussed earlier. You can use this line to instruct the linker to look for libraries in the `/usr/local/lib/pam` directory before looking in the usual places:

```
% g++ -o reciprocal main.o reciprocal.o -L/usr/local/lib/pam -lpam
```

Although you don't have to use the `-I` option to get the preprocessor to search the current directory, you do have to use the `-L` option to get the linker to search the current directory. In particular, you could use the following to instruct the linker to find the `test` library in the current directory:

```
% gcc -o app app.o -L. -ltest
```

1.3 Automating the Process with GNU Make

If you're accustomed to programming for the Windows operating system, you're probably accustomed to working with an Integrated Development Environment (IDE). You add sources files to your project, and then the IDE builds your project automatically. Although IDEs are available for Linux, this book doesn't discuss them. Instead, this book shows you how to use GNU Make to automatically recompile your code, which is what most Linux programmers actually do.

The basic idea behind `make` is simple. You tell `make` what *targets* you want to build and then give *rules* explaining how to build them. You also specify *dependencies* that indicate when a particular target should be rebuilt.

In our sample `reciprocal` project, there are three obvious targets: `reciprocal.o`, `main.o`, and the `reciprocal` itself. You already have rules in mind for building these targets in the form of the command lines given previously. The dependencies require a little bit of thought. Clearly, `reciprocal` depends on `reciprocal.o` and `main.o` because you can't link the complete program until you have built each of the object files. The object files should be rebuilt whenever the corresponding source files change. There's one more twist in that a change to `reciprocal.hpp` also should cause both of the object files to be rebuilt because both source files include that header file.

In addition to the obvious targets, there should always be a `clean` target. This target removes all the generated object files and programs so that you can start fresh. The rule for this target uses the `rm` command to remove the files.

You can convey all that information to `make` by putting the information in a file named `Makefile`. Here's what `Makefile` contains:

```
reciprocal: main.o reciprocal.o
	g++ $(CFLAGS) -o reciprocal main.o reciprocal.o

main.o: main.c reciprocal.hpp
	gcc $(CFLAGS) -c main.c

reciprocal.o: reciprocal.cpp reciprocal.hpp
	g++ $(CFLAGS) -c reciprocal.cpp

clean:
	rm -f *.o reciprocal
```

You can see that targets are listed on the left, followed by a colon and then any dependencies. The rule to build that target is on the next line. (Ignore the `$(CFLAGS)` bit for the moment.) The line with the rule on it must start with a Tab character, or `make` will get confused. If you edit your `Makefile` in Emacs, Emacs will help you with the formatting.

If you remove the object files that you've already built, and just type

```
% make
```

on the command-line, you'll see the following:

```
% make
gcc -c main.c
g++ -c reciprocal.cpp
g++ -o reciprocal main.o reciprocal.o
```

You can see that `make` has automatically built the object files and then linked them. If you now change `main.c` in some trivial way and type `make` again, you'll see the following:

```
% make
gcc -c main.c
g++ -o reciprocal main.o reciprocal.o
```

You can see that `make` knew to rebuild `main.o` and to re-link the program, but it didn't bother to recompile `reciprocal.cpp` because none of the dependencies for `reciprocal.o` had changed.

The `$(CFLAGS)` is a `make` variable. You can define this variable either in the `Makefile` itself or on the command line. GNU `make` will substitute the value of the variable when it executes the rule. So, for example, to recompile with optimization enabled, you would do this:

```
% make clean
rm -f *.o reciprocal
% make CFLAGS=-O2
gcc -O2 -c main.c
g++ -O2 -c reciprocal.cpp
g++ -O2 -o reciprocal main.o reciprocal.o
```

Note that the `-O2` flag was inserted in place of `$(CFLAGS)` in the rules.

In this section, you've seen only the most basic capabilities of `make`. You can find out more by typing this:

```
% info make
```

In that manual, you'll find information about how to make maintaining a `Makefile` easier, how to reduce the number of rules that you need to write, and how to automatically compute dependencies. You can also find more information in *GNU, Autoconf, Automake, and Libtool* by Gary V. Vaughan, Ben Elliston, Tom Tromey, and Ian Lance Taylor (New Riders Publishing, 2000).

1.4 Debugging with GNU Debugger (GDB)

The *debugger* is the program that you use to figure out why your program isn't behaving the way you think it should. You'll be doing this a lot.[5] The GNU Debugger (GDB) is the debugger used by most Linux programmers. You can use GDB to step through your code, set breakpoints, and examine the value of local variables.

1.4.1 Compiling with Debugging Information

To use GDB, you'll have to compile with debugging information enabled. Do this by adding the `-g` switch on the compilation command line. If you're using a `Makefile` as described previously, you can just set `CFLAGS` equal to `-g` when you run `make`, as shown here:

```
% make CFLAGS=-g
gcc -g -c main.c
g++ -g -c reciprocal.cpp
g++ -g -o reciprocal main.o reciprocal.o
```

When you compile with `-g`, the compiler includes extra information in the object files and executables. The debugger uses this information to figure out which addresses correspond to which lines in which source files, how to print out local variables, and so forth.

1.4.2 Running GDB

You can start up `gdb` by typing:

```
% gdb reciprocal
```

When `gdb` starts up, you should see the GDB prompt:

```
(gdb)
```

5. …unless your programs always work the first time.

The first step is to run your program inside the debugger. Just enter the command `run` and any program arguments. Try running the program without any arguments, like this:

```
(gdb) run
Starting program: reciprocal

Program received signal SIGSEGV, Segmentation fault.
__strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
at strtol.c:287
287     strtol.c: No such file or directory.
(gdb)
```

The problem is that there is no error-checking code in `main`. The program expects one argument, but in this case the program was run with no arguments. The `SIGSEGV` message indicates a program crash. GDB knows that the actual crash happened in a function called `__strtol_internal`. That function is in the standard library, and the source isn't installed, which explains the "No such file or directory" message. You can see the stack by using the `where` command:

```
(gdb) where
#0  __strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
    at strtol.c:287
#1  0x40096fb6 in atoi (nptr=0x0) at ../stdlib/stdlib.h:251
#2  0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
```

You can see from this display that `main` called the `atoi` function with a `NULL` pointer, which is the source of the trouble.

You can go up two levels in the stack until you reach `main` by using the `up` command:

```
(gdb) up 2
#2  0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
8         i = atoi (argv[1]);
```

Note that `gdb` is capable of finding the source for `main.c`, and it shows the line where the erroneous function call occurred. You can view the value of variables using the `print` command:

```
(gdb) print argv[1]
$2 = 0x0
```

That confirms that the problem is indeed a `NULL` pointer passed into `atoi`.

You can set a breakpoint by using the `break` command:

```
(gdb) break main
Breakpoint 1 at 0x804862e: file main.c, line 8.
```

This command sets a breakpoint on the first line of `main`.[6] Now try rerunning the program with an argument, like this:

```
(gdb) run 7
Starting program: reciprocal 7

Breakpoint 1, main (argc=2, argv=0xbffff5e4) at main.c:8
8         i = atoi (argv[1]);
```

You can see that the debugger has stopped at the breakpoint.

You can step over the call to `atoi` using the `next` command:

```
(gdb) next
9         printf ("The reciprocal of %d is %g\n", i, reciprocal (i));
```

If you want to see what's going on inside `reciprocal`, use the `step` command like this:

```
(gdb) step
reciprocal (i=7) at reciprocal.cpp:6
6         assert (i != 0);
```

You're now in the body of the `reciprocal` function.

You might find it more convenient to run `gdb` from within Emacs rather than using `gdb` directly from the command line. Use the command `M-x gdb` to start up `gdb` in an Emacs window. If you are stopped at a breakpoint, Emacs automatically pulls up the appropriate source file. It's easier to figure out what's going on when you're looking at the whole file rather than just one line of text.

1.5 Finding More Information

Nearly every Linux distribution comes with a great deal of useful documentation. You could learn most of what we'll talk about in this book by reading documentation in your Linux distribution (although it would probably take you much longer). The documentation isn't always well-organized, though, so the tricky part is finding what you need. Documentation is also sometimes out-of-date, so take everything that you read with a grain of salt. If the system doesn't behave the way a *man page* (manual pages) says it should, for instance, it may be that the man page is outdated.

To help you navigate, here are the most useful sources of information about advanced Linux programming.

6. Some people have commented that saying `break main` is a little bit funny because usually you want to do this only when `main` is already broken.

1.5.1 Man Pages

Linux distributions include man pages for most standard commands, system calls, and standard library functions. The man pages are divided into numbered sections; for programmers, the most important are these:

(1) User commands

(2) System calls

(3) Standard library functions

(8) System/administrative commands

The numbers denote man page sections. Linux's man pages come installed on your system; use the `man` command to access them. To look up a man page, simply invoke `man` *`name`*, where *`name`* is a command or function name. In a few cases, the same name occurs in more than one section; you can specify the section explicitly by placing the section number before the name. For example, if you type the following, you'll get the man page for the `sleep` command (in section 1 of the Linux man pages):

```
% man sleep
```

To see the man page for the `sleep` library function, use this command:

```
% man 3 sleep
```

Each man page includes a one-line summary of the command or function. The `whatis` *`name`* command displays all man pages (in all sections) for a command or function matching *`name`*. If you're not sure which command or function you want, you can perform a keyword search on the summary lines, using `man -k` *`keyword`*.

Man pages include a lot of very useful information and should be the first place you turn for help. The man page for a command describes command-line options and arguments, input and output, error codes, configuration, and the like. The man page for a system call or library function describes parameters and return values, lists error codes and side effects, and specifies which include file to use if you call the function.

1.5.2 Info

The Info documentation system contains more detailed documentation for many core components of the GNU/Linux system, plus several other programs. Info pages are hypertext documents, similar to Web pages. To launch the text-based Info browser, just type `info` in a shell window. You'll be presented with a menu of Info documents installed on your system. (Press Control+H to display the keys for navigating an Info document.)

Among the most useful Info documents are these:

- `gcc`—The gcc compiler
- `libc`—The GNU C library, including many system calls
- `gdb`—The GNU debugger

- emacs—The Emacs text editor
- info—The Info system itself

Almost all the standard Linux programming tools (including ld, the linker; as, the assembler; and gprof, the profiler) come with useful Info pages. You can jump directly to a particular Info document by specifying the page name on the command line:

```
% info libc
```

If you do most of your programming in Emacs, you can access the built-in Info browser by typing M-x info or C-h i.

1.5.3 Header Files

You can learn a lot about the system functions that are available and how to use them by looking at the system header files. These reside in /usr/include and /usr/include/sys. If you are getting compile errors from using a system call, for instance, take a look in the corresponding header file to verify that the function's signature is the same as what's listed in the man page.

On Linux systems, a lot of the nitty-gritty details of how the system calls work are reflected in header files in the directories /usr/include/bits, /usr/include/asm, and /usr/include/linux. For instance, the numerical values of signals (described in Section 3.3, "Signals," in Chapter 3, "Processes") are defined in /usr/include/bits/signum.h. These header files make good reading for inquiring minds. Don't include them directly in your programs, though; always use the header files in /usr/include or as mentioned in the man page for the function you're using.

1.5.4 Source Code

This is Open Source, right? The final arbiter of how the system works is the system source code itself, and luckily for Linux programmers, that source code is freely available. Chances are, your Linux distribution includes full source code for the entire system and all programs included with it; if not, you're entitled under the terms of the GNU General Public License to request it from the distributor. (The source code might not be installed on your disk, though. See your distribution's documentation for instructions on installing it.)

The source code for the Linux kernel itself is usually stored under /usr/src/linux. If this book leaves you thirsting for details of how processes, shared memory, and system devices work, you can always learn straight from the source code. Most of the system functions described in this book are implemented in the GNU C library; check your distribution's documentation for the location of the C library source code.

2 Writing Good GNU/Linux Software

THIS CHAPTER COVERS SOME BASIC TECHNIQUES THAT MOST GNU/Linux programmers use. By following the guidelines presented, you'll be able to write programs that work well within the GNU/Linux environment and meet GNU/Linux users' expectations of how programs should operate.

2.1 Interaction With the Execution Environment

When you first studied C or C++, you learned that the special `main` function is the primary entry point for a program. When the operating system executes your program, it automatically provides certain facilities that help the program communicate with the operating system and the user. You probably learned about the two parameters to `main`, usually called `argc` and `argv`, which receive inputs to your program. You learned about the `stdout` and `stdin` (or the `cout` and `cin` streams in C++) that provide console input and output. These features are provided by the C and C++ languages, and they interact with the GNU/Linux system in certain ways. GNU/Linux provides other ways for interacting with the operating environment, too.

2.1.1 The Argument List

You run a program from a shell prompt by typing the name of the program. Optionally, you can supply additional information to the program by typing one or more words after the program name, separated by spaces. These are called *command-line arguments*. (You can also include an argument that contains a space, by enclosing the argument in quotes.) More generally, this is referred to as the program's *argument list* because it need not originate from a shell command line. In Chapter 3, "Processes," you'll see another way of invoking a program, in which a program can specify the argument list of another program directly.

When a program is invoked from the shell, the argument list contains the entire command line, including the name of the program and any command-line arguments that may have been provided. Suppose, for example, that you invoke the `ls` command in your shell to display the contents of the root directory and corresponding file sizes with this command line:

```
% ls -s /
```

The argument list that the `ls` program receives has three elements. The first one is the name of the program itself, as specified on the command line, namely `ls`. The second and third elements of the argument list are the two command-line arguments, `-s` and `/`.

The `main` function of your program can access the argument list via the `argc` and `argv` parameters to `main` (if you don't use them, you may simply omit them). The first parameter, `argc`, is an integer that is set to the number of items in the argument list. The second parameter, `argv`, is an array of character pointers. The size of the array is `argc`, and the array elements point to the elements of the argument list, as NUL-terminated character strings.

Using command-line arguments is as easy as examining the contents of `argc` and `argv`. If you're not interested in the name of the program itself, don't forget to skip the first element.

Listing 2.1 demonstrates how to use `argc` and `argv`.

Listing 2.1 **(*arglist.c*) Using *argc* and *argv***

```
#include <stdio.h>

int main (int argc, char* argv[])
{
  printf ("The name of this program is '%s'.\n", argv[0]);
  printf ("This program was invoked with %d arguments.\n", argc - 1);

  /* Were any command-line arguments specified?  */
  if (argc > 1) {
    /* Yes, print them.  */
    int i;
    printf ("The arguments are:\n");
    for (i = 1; i < argc; ++i)
```

```
      printf ("  %s\n", argv[i]);
    }

    return 0;
}
```

2.1.2 GNU/Linux Command-Line Conventions

Almost all GNU/Linux programs obey some conventions about how command-line arguments are interpreted. The arguments that programs expect fall into two categories: *options* (or *flags*) and other arguments. Options modify how the program behaves, while other arguments provide inputs (for instance, the names of input files).

Options come in two forms:

- *Short options* consist of a single hyphen and a single character (usually a lowercase or uppercase letter). Short options are quicker to type.
- *Long options* consist of two hyphens, followed by a name made of lowercase and uppercase letters and hyphens. Long options are easier to remember and easier to read (in shell scripts, for instance).

Usually, a program provides both a short form and a long form for most options it supports, the former for brevity and the latter for clarity. For example, most programs understand the options `-h` and `--help`, and treat them identically. Normally, when a program is invoked from the shell, any desired options follow the program name immediately. Some options expect an argument immediately following. Many programs, for example, interpret the option `--output foo` to specify that output of the program should be placed in a file named `foo`. After the options, there may follow other command-line arguments, typically input files or input data.

For example, the command `ls -s /` displays the contents of the root directory. The `-s` option modifies the default behavior of `ls` by instructing it to display the size (in kilobytes) of each entry. The `/` argument tells `ls` which directory to list. The `--size` option is synonymous with `-s`, so the same command could have been invoked as `ls --size /`.

The *GNU Coding Standards* list the names of some commonly used command-line options. If you plan to provide any options similar to these, it's a good idea to use the names specified in the coding standards. Your program will behave more like other programs and will be easier for users to learn. You can view the GNU Coding Standards' guidelines for command-line options by invoking the following from a shell prompt on most GNU/Linux systems:

```
% info "(standards)User Interfaces"
```

2.1.3 Using *getopt_long*

Parsing command-line options is a tedious chore. Luckily, the GNU C library provides a function that you can use in C and C++ programs to make this job somewhat easier (although still a bit annoying). This function, `getopt_long`, understands both short and long options. If you use this function, include the header file `<getopt.h>`.

Suppose, for example, that you are writing a program that is to accept the three options shown in Table 2.1.

Table 2.1 **Example Program Options**

Short Form	Long Form	Purpose
`-h`	`--help`	Display usage summary and exit
`-o` *filename*	`--output` *filename*	Specify output filename
`-v`	`--verbose`	Print verbose messages

In addition, the program is to accept zero or more additional command-line arguments, which are the names of input files.

To use `getopt_long`, you must provide two data structures. The first is a character string containing the valid short options, each a single letter. An option that requires an argument is followed by a colon. For your program, the string `ho:v` indicates that the valid options are `-h`, `-o`, and `-v`, with the second of these options followed by an argument.

To specify the available long options, you construct an array of `struct option` elements. Each element corresponds to one long option and has four fields. In normal circumstances, the first field is the name of the long option (as a character string, without the two hyphens); the second is 1 if the option takes an argument, or 0 otherwise; the third is `NULL`; and the fourth is a character constant specifying the short option synonym for that long option. The last element of the array should be all zeros. You could construct the array like this:

```
const struct option long_options[] = {
  { "help",     0, NULL, 'h' },
  { "output",   1, NULL, 'o' },
  { "verbose",  0, NULL, 'v' },
  { NULL,       0, NULL, 0   }
};
```

You invoke the `getopt_long` function, passing it the `argc` and `argv` arguments to `main`, the character string describing short options, and the array of `struct option` elements describing the long options.

- Each time you call `getopt_long`, it parses a single option, returning the short-option letter for that option, or –1 if no more options are found.
- Typically, you'll call `getopt_long` in a loop, to process all the options the user has specified, and you'll handle the specific options in a switch statement.

- If `getopt_long` encounters an invalid option (an option that you didn't specify as a valid short or long option), it prints an error message and returns the character `?` (a question mark). Most programs will exit in response to this, possibly after displaying usage information.
- When handling an option that takes an argument, the global variable `optarg` points to the text of that argument.
- After `getopt_long` has finished parsing all the options, the global variable `optind` contains the index (into `argv`) of the first nonoption argument.

Listing 2.2 shows an example of how you might use `getopt_long` to process your arguments.

Listing 2.2 **(*getopt_long.c*) Using *getopt_long***

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

/* The name of this program.  */
const char* program_name;

/* Prints usage information for this program to STREAM (typically
   stdout or stderr), and exit the program with EXIT_CODE.  Does not
   return.  */

void print_usage (FILE* stream, int exit_code)
{
  fprintf (stream, "Usage:  %s options [ inputfile ... ]\n", program_name);
  fprintf (stream,
           "  -h  --help             Display this usage information.\n"
           "  -o  --output filename  Write output to file.\n"
           "  -v  --verbose          Print verbose messages.\n");
  exit (exit_code);
}

/* Main program entry point.  ARGC contains number of argument list
   elements; ARGV is an array of pointers to them.  */

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

  /* A string listing valid short options letters.  */
  const char* const short_options = "ho:v";
  /* An array describing valid long options.  */
  const struct option long_options[] = {
    { "help",     0, NULL, 'h' },
    { "output",   1, NULL, 'o' },
    { "verbose",  0, NULL, 'v' },
```

continues

Listing 2.2 **Continued**

```
    { NULL,       0, NULL, 0   }   /* Required at end of array.  */
  };

  /* The name of the file to receive program output, or NULL for
     standard output.  */
  const char* output_filename = NULL;
  /* Whether to display verbose messages.  */
  int verbose = 0;

  /* Remember the name of the program, to incorporate in messages.
     The name is stored in argv[0].  */
  program_name = argv[0];

  do {
    next_option = getopt_long (argc, argv, short_options,
                               long_options, NULL);
    switch (next_option)
    {
    case 'h':   /* -h or --help */
      /* User has requested usage information.  Print it to standard
         output, and exit with exit code zero (normal termination).  */
      print_usage (stdout, 0);

    case 'o':   /* -o or --output */
      /* This option takes an argument, the name of the output file.  */
      output_filename = optarg;
      break;

    case 'v':   /* -v or  --verbose */
      verbose = 1;
      break;

    case '?':   /* The user specified an invalid option.  */
      /* Print usage information to standard error, and exit with exit
         code one (indicating abnormal termination).  */
      print_usage (stderr, 1);

    case -1:    /* Done with options.  */
      break;

    default:    /* Something else: unexpected.  */
      abort ();
    }
  }
  while (next_option != -1);

  /* Done with options.  OPTIND points to first nonoption argument.
     For demonstration purposes, print them if the verbose option was
     specified.  */
```

```
  if (verbose) {
    int i;
    for (i = optind; i < argc; ++i)
      printf ("Argument: %s\n", argv[i]);
  }

  /* The main program goes here.  */

  return 0;
}
```

Using getopt_long may seem like a lot of work, but writing code to parse the command-line options yourself would take even longer. The getopt_long function is very sophisticated and allows great flexibility in specifying what kind of options to accept. However, it's a good idea to stay away from the more advanced features and stick with the basic option structure described.

2.1.4 Standard I/O

The standard C library provides standard input and output streams (stdin and stdout, respectively). These are used by scanf, printf, and other library functions. In the UNIX tradition, use of standard input and output is customary for GNU/Linux programs. This allows the chaining of multiple programs using shell pipes and input and output redirection. (See the man page for your shell to learn its syntax.)

The C library also provides stderr, the standard error stream. Programs should print warning and error messages to standard error instead of standard output. This allows users to separate normal output and error messages, for instance, by redirecting standard output to a file while allowing standard error to print on the console. The fprintf function can be used to print to stderr, for example:

```
fprintf (stderr, ("Error: ..."));
```

These three streams are also accessible with the underlying UNIX I/O commands (read, write, and so on) via file descriptors. These are file descriptors 0 for stdin, 1 for stdout, and 2 for stderr.

When invoking a program, it is sometimes useful to redirect both standard output and standard error to a file or pipe. The syntax for doing this varies among shells; for Bourne-style shells (including bash, the default shell on most GNU/Linux distributions), the syntax is this:

```
% program > output_file.txt 2>&1
% program 2>&1 | filter
```

The 2>&1 syntax indicates that file descriptor 2 (stderr) should be merged into file descriptor 1 (stdout). Note that 2>&1 must follow a file redirection (the first example) but must precede a pipe redirection (the second example).

Note that `stdout` is buffered. Data written to `stdout` is not sent to the console (or other device, if it's redirected) until the buffer fills, the program exits normally, or `stdout` is closed. You can explicitly flush the buffer by calling the following:

```
fflush (stdout);
```

In contrast, `stderr` is not buffered; data written to `stderr` goes directly to the console.[1]

This can produce some surprising results. For example, this loop does not print one period every second; instead, the periods are buffered, and a bunch of them are printed together when the buffer fills.

```
while (1) {
  printf (".");
  sleep (1);
}
```

In this loop, however, the periods do appear once a second:

```
while (1) {
  fprintf (stderr, ".");
  sleep (1);
}
```

2.1.5 Program Exit Codes

When a program ends, it indicates its status with an exit code. The exit code is a small integer; by convention, an exit code of zero denotes successful execution, while nonzero exit codes indicate that an error occurred. Some programs use different nonzero exit code values to distinguish specific errors.

With most shells, it's possible to obtain the exit code of the most recently executed program using the special `$?` variable. Here's an example in which the `ls` command is invoked twice and its exit code is printed after each invocation. In the first case, `ls` executes correctly and returns the exit code zero. In the second case, `ls` encounters an error (because the filename specified on the command line does not exist) and thus returns a nonzero exit code.

```
% ls /
bin   coda  etc   lib         misc  nfs  proc  sbin  usr
boot  dev   home  lost+found  mnt   opt  root  tmp   var
% echo $?
0
% ls bogusfile
ls: bogusfile: No such file or directory
% echo $?
1
```

1. In C++, the same distinction holds for `cout` and `cerr`, respectively. Note that the `endl` token flushes a stream in addition to printing a newline character; if you don't want to flush the stream (for performance reasons, for example), use a newline constant, `'\n'`, instead.

A C or C++ program specifies its exit code by returning that value from the `main` function. There are other methods of providing exit codes, and special exit codes are assigned to programs that terminate abnormally (by a signal). These are discussed further in Chapter 3.

2.1.6 The Environment

GNU/Linux provides each running program with an *environment*. The environment is a collection of variable/value pairs. Both environment variable names and their values are character strings. By convention, environment variable names are spelled in all capital letters.

You're probably familiar with several common environment variables already. For instance:

- `USER` contains your username.
- `HOME` contains the path to your home directory.
- `PATH` contains a colon-separated list of directories through which Linux searches for commands you invoke.
- `DISPLAY` contains the name and display number of the X Window server on which windows from graphical X Window programs will appear.

Your shell, like any other program, has an environment. Shells provide methods for examining and modifying the environment directly. To print the current environment in your shell, invoke the `printenv` program. Various shells have different built-in syntax for using environment variables; the following is the syntax for Bourne-style shells.

- The shell automatically creates a shell variable for each environment variable that it finds, so you can access environment variable values using the `$`*`varname`* syntax. For instance:

  ```
  % echo $USER
  samuel
  % echo $HOME
  /home/samuel
  ```

- You can use the `export` command to export a shell variable into the environment. For example, to set the `EDITOR` environment variable, you would use this:

  ```
  % EDITOR=emacs
  % export EDITOR
  ```

  Or, for short:

  ```
  % export EDITOR=emacs
  ```

In a program, you access an environment variable with the `getenv` function in `<stdlib.h>`. That function takes a variable name and returns the corresponding value as a character string, or `NULL` if that variable is not defined in the environment. To set or clear environment variables, use the `setenv` and `unsetenv` functions, respectively.

Enumerating all the variables in the environment is a little trickier. To do this, you must access a special global variable named `environ`, which is defined in the GNU C library. This variable, of type `char**`, is a `NULL`-terminated array of pointers to character strings. Each string contains one environment variable, in the form `VARIABLE=value`.

The program in Listing 2.3, for instance, simply prints the entire environment by looping through the `environ` array.

Listing 2.3 **(*print-env.c*) Printing the Execution Environment**

```
#include <stdio.h>

/* The ENVIRON variable contains the environment.  */
extern char** environ;

int main ()
{
  char** var;
  for (var = environ; *var != NULL; ++var)
    printf ("%s\n", *var);
  return 0;
}
```

Don't modify `environ` yourself; use the `setenv` and `unsetenv` functions instead.

Usually, when a new program is started, it inherits a copy of the environment of the program that invoked it (the shell program, if it was invoked interactively). So, for instance, programs that you run from the shell may examine the values of environment variables that you set in the shell.

Environment variables are commonly used to communicate configuration information to programs. Suppose, for example, that you are writing a program that connects to an Internet server to obtain some information. You could write the program so that the server name is specified on the command line. However, suppose that the server name is not something that users will change very often. You can use a special environment variable—say `SERVER_NAME`—to specify the server name; if that variable doesn't exist, a default value is used. Part of your program might look as shown in Listing 2.4.

Listing 2.4 **(*client.c*) Part of a Network Client Program**

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

int main ()
{
```

```
  char* server_name = getenv ("SERVER_NAME");
  if (server_name == NULL)
    /* The SERVER_NAME environment variable was not set.  Use the
       default.  */
    server_name = "server.my-company.com";

  printf ("accessing server %s\n", server_name);
  /* Access the server here...  */

  return 0;
}
```

Suppose that this program is named `client`. Assuming that you haven't set the `SERVER_NAME` variable, the default value for the server name is used:

```
% client
accessing server server.my-company.com
```

But it's easy to specify a different server:

```
% export SERVER_NAME=backup-server.elsewhere.net
% client
accessing server backup-server.elsewhere.net
```

2.1.7 Using Temporary Files

Sometimes a program needs to make a temporary file, to store large data for a while or to pass data to another program. On GNU/Linux systems, temporary files are stored in the `/tmp` directory. When using temporary files, you should be aware of the following pitfalls:

- More than one instance of your program may be run simultaneously (by the same user or by different users). The instances should use different temporary filenames so that they don't collide.
- The file permissions of the temporary file should be set in such a way that unauthorized users cannot alter the program's execution by modifying or replacing the temporary file.
- Temporary filenames should be generated in a way that cannot be predicted externally; otherwise, an attacker can exploit the delay between testing whether a given name is already in use and opening a new temporary file.

GNU/Linux provides functions, `mkstemp` and `tmpfile`, that take care of these issues for you (in addition to several functions that don't). Which you use depends on whether you plan to hand the temporary file to another program, and whether you want to use UNIX I/O (`open`, `write`, and so on) or the C library's stream I/O functions (`fopen`, `fprintf`, and so on).

Using *mkstemp*

The `mkstemp` function creates a unique temporary filename from a filename template, creates the file with permissions so that only the current user can access it, and opens the file for read/write. The filename template is a character string ending with "XXXXXX" (six capital X's); `mkstemp` replaces the X's with characters so that the filename is unique. The return value is a file descriptor; use the `write` family of functions to write to the temporary file.

Temporary files created with `mkstemp` are not deleted automatically. It's up to you to remove the temporary file when it's no longer needed. (Programmers should be very careful to clean up temporary files; otherwise, the `/tmp` file system will fill up eventually, rendering the system inoperable.) If the temporary file is for internal use only and won't be handed to another program, it's a good idea to call `unlink` on the temporary file immediately. The `unlink` function removes the directory entry corresponding to a file, but because files in a file system are reference-counted, the file itself is not removed until there are no open file descriptors for that file, either. This way, your program may continue to use the temporary file, and the file goes away automatically as soon as you close the file descriptor. Because Linux closes file descriptors when a program ends, the temporary file will be removed even if your program terminates abnormally.

The pair of functions in Listing 2.5 demonstrates `mkstemp`. Used together, these functions make it easy to write a memory buffer to a temporary file (so that memory can be freed or reused) and then read it back later.

Listing 2.5 **(*temp_file.c*) Using *mkstemp***

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

/* A handle for a temporary file created with write_temp_file.  In
   this implementation, it's just a file descriptor.  */
typedef int temp_file_handle;

/* Writes LENGTH bytes from BUFFER into a temporary file.  The
   temporary file is immediately unlinked.  Returns a handle to the
   temporary file.  */

temp_file_handle write_temp_file (char* buffer, size_t length)
{
  /* Create the filename and file.  The XXXXXX will be replaced with
     characters that make the filename unique.  */
  char temp_filename[] = "/tmp/temp_file.XXXXXX";
  int fd = mkstemp (temp_filename);
  /* Unlink the file immediately, so that it will be removed when the
     file descriptor is closed.  */
  unlink (temp_filename);
  /* Write the number of bytes to the file first.  */
  write (fd, &length, sizeof (length));
```

```
  /* Now write the data itself.  */
  write (fd, buffer, length);
  /* Use the file descriptor as the handle for the temporary file.  */
  return fd;
}

/* Reads the contents of a temporary file TEMP_FILE created with
   write_temp_file.  The return value is a newly allocated buffer of
   those contents, which the caller must deallocate with free.
   *LENGTH is set to the size of the contents, in bytes.  The
   temporary file is removed.  */

char* read_temp_file (temp_file_handle temp_file, size_t* length)
{
  char* buffer;
  /* The TEMP_FILE handle is a file descriptor to the temporary file.  */
  int fd = temp_file;
  /* Rewind to the beginning of the file.  */
  lseek (fd, 0, SEEK_SET);
  /* Read the size of the data in the temporary file.  */
  read (fd, length, sizeof (*length));
  /* Allocate a buffer and read the data.  */
  buffer = (char*) malloc (*length);
  read (fd, buffer, *length);
  /* Close the file descriptor, which will cause the temporary file to
     go away.  */
  close (fd);
  return buffer;
}
```

Using *tmpfile*

If you are using the C library I/O functions and don't need to pass the temporary file to another program, you can use the `tmpfile` function. This creates and opens a temporary file, and returns a file pointer to it. The temporary file is already unlinked, as in the previous example, so it is deleted automatically when the file pointer is closed (with `fclose`) or when the program terminates.

GNU/Linux provides several other functions for generating temporary files and temporary filenames, including `mktemp`, `tmpnam`, and `tempnam`. Don't use these functions, though, because they suffer from the reliability and security problems already mentioned.

2.2 Coding Defensively

Writing programs that run correctly under "normal" use is hard; writing programs that behave gracefully in failure situations is harder. This section demonstrates some coding techniques for finding bugs early and for detecting and recovering from problems in a running program.

The code samples presented later in this book deliberately skip extensive error checking and recovery code because this would obscure the basic functionality being presented. However, the final example in Chapter 11, "A Sample GNU/Linux Application," comes back to demonstrating how to use these techniques to write robust programs.

2.2.1 Using *assert*

A good objective to keep in mind when coding application programs is that bugs or unexpected errors should cause the program to fail dramatically, as early as possible. This will help you find bugs earlier in the development and testing cycles. Failures that don't exhibit themselves dramatically are often missed and don't show up until the application is in users' hands.

One of the simplest methods to check for unexpected conditions is the standard C `assert` macro. The argument to this macro is a Boolean expression. The program is terminated if the expression evaluates to false, after printing an error message containing the source file and line number and the text of the expression. The `assert` macro is very useful for a wide variety of consistency checks internal to a program. For instance, use `assert` to test the validity of function arguments, to test preconditions and postconditions of function calls (and method calls, in C++), and to test for unexpected return values.

Each use of `assert` serves not only as a runtime check of a condition, but also as documentation about the program's operation within the source code. If your program contains an `assert (`*`condition`*`)` that says to someone reading your source code that *`condition`* should always be true at that point in the program, and if *`condition`* is not true, it's probably a bug in the program.

For performance-critical code, runtime checks such as uses of `assert` can impose a significant performance penalty. In these cases, you can compile your code with the `NDEBUG` macro defined, by using the `-DNDEBUG` flag on your compiler command line. With `NDEBUG` set, appearances of the `assert` macro will be preprocessed away. It's a good idea to do this only when necessary for performance reasons, though, and only with performance-critical source files.

Because it is possible to preprocess `assert` macros away, be careful that any expression you use with `assert` has no side effects. Specifically, you shouldn't call functions inside `assert` expressions, assign variables, or use modifying operators such as `++`.

Suppose, for example, that you call a function, `do_something`, repeatedly in a loop. The `do_something` function returns zero on success and nonzero on failure, but you don't expect it ever to fail in your program. You might be tempted to write:

```
for (i = 0; i < 100; ++i)
  assert (do_something () == 0);
```

However, you might find that this runtime check imposes too large a performance penalty and decide later to recompile with `NDEBUG` defined. This will remove the `assert` call entirely, so the expression will never be evaluated and `do_something` will never be called. You should write this instead:

```
for (i = 0; i < 100; ++i) {
  int status = do_something ();
  assert (status == 0);
}
```

Another thing to bear in mind is that you should not use `assert` to test for invalid user input. Users don't like it when applications simply crash with a cryptic error message, even in response to invalid input. You should still always check for invalid input and produce sensible error messages in response input. Use `assert` for internal runtime checks only.

Some good places to use `assert` are these:

- Check against null pointers, for instance, as invalid function arguments. The error message generated by {`assert (pointer !=  NULL)`},

  ```
  Assertion 'pointer != ((void *)0)' failed.
  ```

  is more informative than the error message that would result if your program dereferenced a null pointer:

  ```
  Segmentation fault (core dumped)
  ```

- Check conditions on function parameter values. For instance, if a function should be called only with a positive value for parameter `foo`, use this at the beginning of the function body:

  ```
  assert (foo > 0);
  ```

  This will help you detect misuses of the function, and it also makes it very clear to someone reading the function's source code that there is a restriction on the parameter's value.

Don't hold back; use `assert` liberally throughout your programs.

2.2.2 System Call Failures

Most of us were originally taught how to write programs that execute to completion along a well-defined path. We divide the program into tasks and subtasks, and each function completes a task by invoking other functions to perform corresponding subtasks. Given appropriate inputs, we expect a function to produce the correct output and side effects.

The realities of computer hardware and software intrude into this idealized dream. Computers have limited resources; hardware fails; many programs execute at the same time; users and programmers make mistakes. It's often at the boundary between the application and the operating system that these realities exhibit themselves. Therefore, when using system calls to access system resources, to perform I/O, or for other purposes, it's important to understand not only what happens when the call succeeds, but also how and when the call can fail.

System calls can fail in many ways. For example:

- The system can run out of resources (or the program can exceed the resource limits enforced by the system of a single program). For example, the program might try to allocate too much memory, to write too much to a disk, or to open too many files at the same time.
- Linux may block a certain system call when a program attempts to perform an operation for which it does not have permission. For example, a program might attempt to write to a file marked read-only, to access the memory of another process, or to kill another user's program.
- The arguments to a system call might be invalid, either because the user provided invalid input or because of a program bug. For instance, the program might pass an invalid memory address or an invalid file descriptor to a system call. Or, a program might attempt to open a directory as an ordinary file, or might pass the name of an ordinary file to a system call that expects a directory.
- A system call can fail for reasons external to a program. This happens most often when a system call accesses a hardware device. The device might be faulty or might not support a particular operation, or perhaps a disk is not inserted in the drive.
- A system call can sometimes be interrupted by an external event, such as the delivery of a signal. This might not indicate outright failure, but it is the responsibility of the calling program to restart the system call, if desired.

In a well-written program that makes extensive use of system calls, it is often the case that more code is devoted to detecting and handling errors and other exceptional circumstances than to the main work of the program.

2.2.3 Error Codes from System Calls

A majority of system calls return zero if the operation succeeds, or a nonzero value if the operation fails. (Many, though, have different return value conventions; for instance, `malloc` returns a null pointer to indicate failure. Always read the man page carefully when using a system call.) Although this information may be enough to determine whether the program should continue execution as usual, it probably does not provide enough information for a sensible recovery from errors.

Most system calls use a special variable named `errno` to store additional information in case of failure.[2] When a call fails, the system sets `errno` to a value indicating what went wrong. Because all system calls use the same `errno` variable to store error information, you should copy the value into another variable immediately after the failed call. The value of `errno` will be overwritten the next time you make a system call.

Error values are integers; possible values are given by preprocessor macros, by convention named in all capitals and starting with "E"—for example, `EACCES` and `EINVAL`. Always use these macros to refer to `errno` values rather than integer values. Include the `<errno.h>` header if you use `errno` values.

GNU/Linux provides a convenient function, `strerror`, that returns a character string description of an `errno` error code, suitable for use in error messages. Include `<string.h>` if you use `strerror`.

GNU/Linux also provides `perror`, which prints the error description directly to the `stderr` stream. Pass to `perror` a character string prefix to print before the error description, which should usually include the name of the function that failed. Include `<stdio.h>` if you use `perror`.

This code fragment attempts to open a file; if the open fails, it prints an error message and exits the program. Note that the `open` call returns an open file descriptor if the open operation succeeds, or –1 if the operation fails.

```
fd = open ("inputfile.txt", O_RDONLY);
if (fd == -1) {
  /* The open failed.  Print an error message and exit.  */
  fprintf (stderr, "error opening file: %s\n", strerror (errno));
  exit (1);
}
```

Depending on your program and the nature of the system call, the appropriate action in case of failure might be to print an error message, to cancel an operation, to abort the program, to try again, or even to ignore the error. It's important, though, to include logic that handles all possible failure modes in some way or another.

2. Actually, for reasons of thread safety, `errno` is implemented as a macro, but it is used like a global variable.

One possible error code that you should be on the watch for, especially with I/O functions, is `EINTR`. Some functions, such as `read`, `select`, and `sleep`, can take significant time to execute. These are considered *blocking* functions because program execution is blocked until the call is completed. However, if the program receives a signal while blocked in one of these calls, the call will return without completing the operation. In this case, `errno` is set to `EINTR`. Usually, you'll want to retry the system call in this case.

Here's a code fragment that uses the `chown` call to change the owner of a file given by `path` to the user by `user_id`. If the call fails, the program takes action depending on the value of `errno`. Notice that when we detect what's probably a bug in the program, we exit using `abort` or `assert`, which cause a core file to be generated. This can be useful for post-mortem debugging. For other unrecoverable errors, such as out-of-memory conditions, we exit using `exit` and a nonzero exit value instead because a core file wouldn't be very useful.

```
rval = chown (path, user_id, -1);
if (rval != 0) {
  /* Save errno because it's clobbered by the next system call.  */
  int error_code = errno;
  /* The operation didn't succeed; chown should return -1 on error.  */
  assert (rval == -1);
  /* Check the value of errno, and take appropriate action.  */
  switch (error_code) {
  case EPERM:         /* Permission denied.  */
  case EROFS:         /* PATH is on a read-only file system.  */
  case ENAMETOOLONG:  /* PATH is too long.  */
  case ENOENT:        /* PATH does not exit.  */
  case ENOTDIR:       /* A component of PATH is not a directory.  */
  case EACCES:        /* A component of PATH is not accessible.  */
    /* Something's wrong with the file.  Print an error message.  */
    fprintf (stderr, "error changing ownership of %s: %s\n",
             path, strerror (error_code));
    /* Don't end the program; perhaps give the user a chance to
       choose another file...  */
    break;

  case EFAULT:
    /* PATH contains an invalid memory address.  This is probably a bug.  */
    abort ();

  case ENOMEM:
    /* Ran out of kernel memory.  */
    fprintf (stderr, "%s\n", strerror (error_code));
    exit (1);

  default:
    /* Some other, unexpected, error code.  We've tried to handle all
       possible error codes; if we've missed one, that's a bug!  */
    abort ();
  };
}
```

You could simply have used this code, which behaves the same way if the call succeeds:

```
rval = chown (path, user_id, -1);
assert (rval == 0);
```

But if the call fails, this alternative makes no effort to report, handle, or recover from errors.

Whether you use the first form, the second form, or something in between depends on the error detection and recovery requirements for your program.

2.2.4 Errors and Resource Allocation

Often, when a system call fails, it's appropriate to cancel the current operation but not to terminate the program because it may be possible to recover from the error. One way to do this is to return from the current function, passing a return code to the caller indicating the error.

If you decide to return from the middle of a function, it's important to make sure that any resources successfully allocated previously in the function are first deallocated. These resources can include memory, file descriptors, file pointers, temporary files, synchronization objects, and so on. Otherwise, if your program continues running, the resources allocated before the failure occurred will be leaked.

Consider, for example, a function that reads from a file into a buffer. The function might follow these steps:

1. Allocate the buffer.
2. Open the file.
3. Read from the file into the buffer.
4. Close the file.
5. Return the buffer.

If the file doesn't exist, Step 2 will fail. An appropriate course of action might be to return `NULL` from the function. However, if the buffer has already been allocated in Step 1, there is a risk of leaking that memory. You must remember to deallocate the buffer somewhere along any flow of control from which you don't return. If Step 3 fails, not only must you deallocate the buffer before returning, but you also must close the file.

Listing 2.6 shows an example of how you might write this function.

Listing 2.6 **(*readfile.c*) Freeing Resources During Abnormal Conditions**

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

continues

Listing 2.6 **Continued**

```
char* read_from_file (const char* filename, size_t length)
{
  char* buffer;
  int fd;
  ssize_t bytes_read;

  /* Allocate the buffer.  */
  buffer = (char*) malloc (length);
  if (buffer == NULL)
    return NULL;
  /* Open the file.  */
  fd = open (filename, O_RDONLY);
  if (fd == -1) {
    /* open failed.  Deallocate buffer before returning.  */
    free (buffer);
    return NULL;
  }
  /* Read the data.  */
  bytes_read = read (fd, buffer, length);
  if (bytes_read != length) {
    /* read failed.  Deallocate buffer and close fd before returning.  */
    free (buffer);
    close (fd);
    return NULL;
  }
  /* Everything's fine.  Close the file and return the buffer.  */
  close (fd);
  return buffer;
}
```

Linux cleans up allocated memory, open files, and most other resources when a program terminates, so it's not necessary to deallocate buffers and close files before calling `exit`. You might need to manually free other shared resources, however, such as temporary files and shared memory, which can potentially outlive a program.

2.3 Writing and Using Libraries

Virtually all programs are linked against one or more libraries. Any program that uses a C function (such as `printf` or `malloc`) will be linked against the C runtime library. If your program has a graphical user interface (GUI), it will be linked against windowing libraries. If your program uses a database, the database provider will give you libraries that you can use to access the database conveniently.

In each of these cases, you must decide whether to link the library *statically* or *dynamically*. If you choose to link statically, your programs will be bigger and harder to upgrade, but probably easier to deploy. If you link dynamically, your programs will be smaller, easier to upgrade, but harder to deploy. This section explains how to link both statically and dynamically, examines the trade-offs in more detail, and gives some "rules of thumb" for deciding which kind of linking is better for you.

2.3.1 Archives

An *archive* (or static library) is simply a collection of object files stored as a single file. (An archive is roughly the equivalent of a Windows `.LIB` file.) When you provide an archive to the linker, the linker searches the archive for the object files it needs, extracts them, and links them into your program much as if you had provided those object files directly.

You can create an archive using the `ar` command. Archive files traditionally use a `.a` extension rather than the `.o` extension used by ordinary object files. Here's how you would combine `test1.o` and `test2.o` into a single `libtest.a` archive:

```
% ar cr libtest.a test1.o test2.o
```

The `cr` flags tell `ar` to create the archive.[3] Now you can link with this archive using the `-ltest` option with `gcc` or `g++`, as described in Section 1.2.2, "Linking Object Files," in Chapter 1, "Getting Started."

When the linker encounters an archive on the command line, it searches the archive for all definitions of symbols (functions or variables) that are referenced from the object files that it has already processed but not yet defined. The object files that define those symbols are extracted from the archive and included in the final executable. Because the linker searches the archive when it is encountered on the command line, it usually makes sense to put archives at the end of the command line. For example, suppose that `test.c` contains the code in Listing 2.7 and `app.c` contains the code in Listing 2.8.

Listing 2.7 **(*test.c*) Library Contents**

```
int f ()
{
  return 3;
}
```

Listing 2.8 **(*app.c*) A Program That Uses Library Functions**

```
int main ()
{
  return f ();
}
```

3. You can use other flags to remove a file from an archive or to perform other operations on the archive. These operations are rarely used but are documented on the `ar` man page.

Now suppose that `test.o` is combined with some other object files to produce the `libtest.a` archive. The following command line will not work:

```
% gcc -o app -L. -ltest app.o
app.o: In function 'main':
app.o(.text+0x4): undefined reference to 'f'
collect2: ld returned 1 exit status
```

The error message indicates that even though `libtest.a` contains a definition of `f`, the linker did not find it. That's because `libtest.a` was searched when it was first encountered, and at that point the linker hadn't seen any references to `f`.

On the other hand, if we use this line, no error messages are issued:

```
% gcc -o app app.o -L. –ltest
```

The reason is that the reference to `f` in `app.o` causes the linker to include the `test.o` object file from the `libtest.a` archive.

2.3.2 Shared Libraries

A *shared library* (also known as a shared object, or as a dynamically linked library) is similar to a archive in that it is a grouping of object files. However, there are many important differences. The most fundamental difference is that when a shared library is linked into a program, the final executable does not actually contain the code that is present in the shared library. Instead, the executable merely contains a reference to the shared library. If several programs on the system are linked against the same shared library, they will all reference the library, but none will actually be included. Thus, the library is "shared" among all the programs that link with it.

A second important difference is that a shared library is not merely a collection of object files, out of which the linker chooses those that are needed to satisfy undefined references. Instead, the object files that compose the shared library are combined into a single object file so that a program that links against a shared library always includes all of the code in the library, rather than just those portions that are needed.

To create a shared library, you must compile the objects that will make up the library using the `-fPIC` option to the compiler, like this:

```
% gcc -c -fPIC test1.c
```

The `-fPIC` option tells the compiler that you are going to be using `test.o` as part of a shared object.

Position-Independent Code (PIC)

PIC stands for position-independent code. The functions in a shared library may be loaded at different addresses in different programs, so the code in the shared object must not depend on the address (or position) at which it is loaded. This consideration has no impact on you, as the programmer, except that you must remember to use the `-fPIC` flag when compiling code that will be used in a shared library.

Then you combine the object files into a shared library, like this:

```
% gcc -shared -fPIC -o libtest.so test1.o test2.o
```

The `-shared` option tells the linker to produce a shared library rather than an ordinary executable. Shared libraries use the extension `.so`, which stands for shared object. Like static archives, the name always begins with `lib` to indicate that the file is a library.

Linking with a shared library is just like linking with a static archive. For example, the following line will link with `libtest.so` if it is in the current directory, or one of the standard library search directories on the system:

```
% gcc -o app app.o -L. –ltest
```

Suppose that both `libtest.a` and `libtest.so` are available. Then the linker must choose one of the libraries and not the other. The linker searches each directory (first those specified with `-L` options, and then those in the standard directories). When the linker finds a directory that contains either `libtest.a` or `libtest.so`, the linker stops search directories. If only one of the two variants is present in the directory, the linker chooses that variant. Otherwise, the linker chooses the shared library version, unless you explicitly instruct it otherwise. You can use the `-static` option to demand static archives. For example, the following line will use the `libtest.a` archive, even if the `libtest.so` shared library is also available:

```
% gcc -static -o app app.o -L. –ltest
```

The `ldd` command displays the shared libraries that are linked into an executable. These libraries need to be available when the executable is run. Note that `ldd` will list an additional library called `ld-linux.so`, which is a part of GNU/Linux's dynamic linking mechanism.

Using *LD_LIBRARY_PATH*

When you link a program with a shared library, the linker does not put the full path to the shared library in the resulting executable. Instead, it places only the name of the shared library. When the program is actually run, the system searches for the shared library and loads it. The system searches only `/lib` and `/usr/lib`, by default. If a shared library that is linked into your program is installed outside those directories, it will not be found, and the system will refuse to run the program.

One solution to this problem is to use the `-Wl,-rpath` option when linking the program. Suppose that you use this:

```
% gcc -o app app.o -L. -ltest -Wl,-rpath,/usr/local/lib
```

Then, when `app` is run, the system will search `/usr/local/lib` for any required shared libraries.

Another solution to this problem is to set the LD_LIBRARY_PATH environment variable when running the program. Like the PATH environment variable, LD_LIBRARY_PATH is a colon-separated list of directories. For example, if LD_LIBRARY_PATH is /usr/local/lib:/opt/lib, then /usr/local/lib and /opt/lib will be searched before the standard /lib and /usr/lib directories. You should also note that if you have LD_LIBRARY_PATH, the linker will search the directories given there in addition to the directories given with the -L option when it is building an executable.[4]

2.3.3 Standard Libraries

Even if you didn't specify any libraries when you linked your program, it almost certainly uses a shared library. That's because GCC automatically links in the standard C library, libc, for you. The standard C library math functions are not included in libc; instead, they're in a separate library, libm, which you need to specify explicitly. For example, to compile and link a program compute.c which uses trigonometric functions such as sin and cos, you must invoke this code:

```
% gcc -o compute compute.c -lm
```

If you write a C++ program and link it using the c++ or g++ commands, you'll also get the standard C++ library, libstdc++, automatically.

2.3.4 Library Dependencies

One library will often depend on another library. For example, many GNU/Linux systems include libtiff, a library that contains functions for reading and writing image files in the TIFF format. This library, in turn, uses the libraries libjpeg (JPEG image routines) and libz (compression routines).

Listing 2.9 shows a very small program that uses libtiff to open a TIFF image file.

Listing 2.9 (*tifftest.c*) Using *libtiff*

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

int main (int argc, char** argv)
{
  TIFF* tiff;
  tiff = TIFFOpen (argv[1], "r");
  TIFFClose (tiff);
  return 0;
}
```

4. You might see a reference to LD_RUN_PATH in some online documentation. Don't believe what you read; this variable does not actually do anything under GNU/Linux.

Save this source file as `tifftest.c`. To compile this program and link with `libtiff`, specify `-ltiff` on your link line:

```
% gcc -o tifftest tifftest.c -ltiff
```

By default, this will pick up the shared-library version of `libtiff`, found at `/usr/lib/libtiff.so`. Because `libtiff` uses `libjpeg` and `libz`, the shared-library versions of these two are also drawn in (a shared library can point to other shared libraries that it depends on). To verify this, use the `ldd` command:

```
% ldd tifftest
        libtiff.so.3 => /usr/lib/libtiff.so.3 (0x4001d000)
        libc.so.6 => /lib/libc.so.6 (0x40060000)
        libjpeg.so.62 => /usr/lib/libjpeg.so.62 (0x40155000)
        libz.so.1 => /usr/lib/libz.so.1 (0x40174000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Static libraries, on the other hand, cannot point to other libraries. If decide to link with the static version of `libtiff` by specifying `-static` on your command line, you will encounter unresolved symbols:

```
% gcc -static -o tifftest tifftest.c -ltiff
/usr/bin/../lib/libtiff.a(tif_jpeg.o): In function 'TIFFjpeg_error_exit':
tif_jpeg.o(.text+0x2a): undefined reference to 'jpeg_abort'
/usr/bin/../lib/libtiff.a(tif_jpeg.o): In function 'TIFFjpeg_create_compress':
tif_jpeg.o(.text+0x8d): undefined reference to 'jpeg_std_error'
tif_jpeg.o(.text+0xcf): undefined reference to 'jpeg_CreateCompress'
...
```

To link this program statically, you must specify the other two libraries yourself:

```
% gcc -static -o tifftest tifftest.c -ltiff -ljpeg -lz
```

Occasionally, two libraries will be mutually dependent. In other words, the first archive will reference symbols defined in the second archive, and vice versa. This situation generally arises out of poor design, but it does occasionally arise. In this case, you can provide a single library multiple times on the command line. The linker will research the library each time it occurs. For example, this line will cause `libfoo.a` to be searched multiple times:

```
% gcc -o app app.o -lfoo -lbar -lfoo
```

So, even if `libfoo.a` references symbols in `libbar.a`, and vice versa, the program will link successfully.

2.3.5 Pros and Cons

Now that you know all about static archives and shared libraries, you're probably wondering which to use. There are a few major considerations to keep in mind.

One major advantage of a shared library is that it saves space on the system where the program is installed. If you are installing 10 programs, and they all make use of the same shared library, then you save a lot of space by using a shared library. If you used a static archive instead, the archive is included in all 10 programs. So, using shared libraries saves disk space. It also reduces download times if your program is being downloaded from the Web.

A related advantage to shared libraries is that users can upgrade the libraries without upgrading all the programs that depend on them. For example, suppose that you produce a shared library that manages HTTP connections. Many programs might depend on this library. If you find a bug in this library, you can upgrade the library. Instantly, all the programs that depend on the library will be fixed; you don't have to relink all the programs the way you do with a static archive.

Those advantages might make you think that you should always use shared libraries. However, substantial reasons exist to use static archives instead. The fact that an upgrade to a shared library affects all programs that depend on it can be a disadvantage. For example, if you're developing mission-critical software, you might rather link to a static archive so that an upgrade to shared libraries on the system won't affect your program. (Otherwise, users might upgrade the shared library, thereby breaking your program, and then call your customer support line, blaming you!)

If you're not going to be able to install your libraries in `/lib` or `/usr/lib`, you should definitely think twice about using a shared library. (You won't be able to install your libraries in those directories if you expect users to install your software without administrator privileges.) In particular, the `-Wl,-rpath` trick won't work if you don't know where the libraries are going to end up. And asking your users to set `LD_LIBRARY_PATH` means an extra step for them. Because each user has to do this individually, this is a substantial additional burden.

You'll have to weigh these advantages and disadvantages for every program you distribute.

2.3.6 Dynamic Loading and Unloading

Sometimes you might want to load some code at run time without explicitly linking in that code. For example, consider an application that supports "plug-in" modules, such as a Web browser. The browser allows third-party developers to create plug-ins to provide additional functionality. The third-party developers create shared libraries and place them in a known location. The Web browser then automatically loads the code in these libraries.

This functionality is available under Linux by using the `dlopen` function. You could open a shared library named `libtest.so` by calling `dlopen` like this:

```
dlopen ("libtest.so", RTLD_LAZY)
```

(The second parameter is a flag that indicates how to bind symbols in the shared library. You can consult the online man pages for `dlopen` if you want more information, but `RTLD_LAZY` is usually the setting that you want.) To use dynamic loading functions, include the `<dlfcn.h>` header file and link with the `-ldl` option to pick up the `libdl` library.

The return value from this function is a `void *` that is used as a handle for the shared library. You can pass this value to the `dlsym` function to obtain the address of a function that has been loaded with the shared library. For example, if `libtest.so` defines a function named `my_function`, you could call it like this:

```
void* handle = dlopen ("libtest.so", RTLD_LAZY);
void (*test)() = dlsym (handle, "my_function");
(*test)();
dlclose (handle);
```

The `dlsym` system call can also be used to obtain a pointer to a static variable in the shared library.

Both `dlopen` and `dlsym` return `NULL` if they do not succeed. In that event, you can call `dlerror` (with no parameters) to obtain a human-readable error message describing the problem.

The `dlclose` function unloads the shared library. Technically, `dlopen` actually loads the library only if it is not already loaded. If the library has already been loaded, `dlopen` simply increments the library reference count. Similarly, `dlclose` decrements the reference count and then unloads the library only if the reference count has reached zero.

If you're writing the code in your shared library in C++, you will probably want to declare those functions and variables that you plan to access elsewhere with the `extern "C"` linkage specifier. For instance, if the C++ function `my_function` is in a shared library and you want to access it with `dlsym`, you should declare it like this:

```
extern "C" void foo ();
```

This prevents the C++ compiler from mangling the function name, which would change the function's name from `foo` to a different, funny-looking name that encodes extra information about the function. A C compiler will not mangle names; it will use whichever name you give to your function or variable.

3
Processes

A RUNNING INSTANCE OF A PROGRAM IS CALLED A *PROCESS*. If you have two terminal windows showing on your screen, then you are probably running the same terminal program twice—you have two terminal processes. Each terminal window is probably running a shell; each running shell is another process. When you invoke a command from a shell, the corresponding program is executed in a new process; the shell process resumes when that process completes.

Advanced programmers often use multiple cooperating processes in a single application to enable the application to do more than one thing at once, to increase application robustness, and to make use of already-existing programs.

Most of the process manipulation functions described in this chapter are similar to those on other UNIX systems. Most are declared in the header file `<unistd.h>`; check the man page for each function to be sure.

3.1 Looking at Processes

Even as you sit down at your computer, there are processes running. Every executing program uses one or more processes. Let's start by taking a look at the processes already on your computer.

3.1.1 Process IDs

Each process in a Linux system is identified by its unique *process ID*, sometimes referred to as *pid*. Process IDs are 16-bit numbers that are assigned sequentially by Linux as new processes are created.

Every process also has a parent process (except the special `init` process, described in Section 3.4.3, "Zombie Processes"). Thus, you can think of the processes on a Linux system as arranged in a tree, with the `init` process at its root. The *parent process ID*, or *ppid*, is simply the process ID of the process's parent.

When referring to process IDs in a C or C++ program, always use the `pid_t` typedef, which is defined in `<sys/types.h>`. A program can obtain the process ID of the process it's running in with the `getpid()` system call, and it can obtain the process ID of its parent process with the `getppid()` system call. For instance, the program in Listing 3.1 prints its process ID and its parent's process ID.

Listing 3.1 **(*print-pid.c*) Printing the Process ID**

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

int main ()
{
  printf ("The process ID is %d\n", (int) getpid ());
  printf ("The parent process ID is %d\n", (int) getppid ());
  return 0;
}
```

Observe that if you invoke this program several times, a different process ID is reported because each invocation is in a new process. However, if you invoke it every time from the same shell, the parent process ID (that is, the process ID of the shell process) is the same.

3.1.2 Viewing Active Processes

The `ps` command displays the processes that are running on your system. The GNU/Linux version of `ps` has lots of options because it tries to be compatible with versions of `ps` on several other UNIX variants. These options control which processes are listed and what information about each is shown.

By default, invoking `ps` displays the processes controlled by the terminal or terminal window in which `ps` is invoked. For example:

```
% ps
  PID TTY          TIME CMD
21693 pts/8    00:00:00 bash
21694 pts/8    00:00:00 ps
```

This invocation of `ps` shows two processes. The first, `bash`, is the shell running on this terminal. The second is the running instance of the `ps` program itself. The first column, labeled `PID`, displays the process ID of each.

For a more detailed look at what's running on your GNU/Linux system, invoke this:

```
% ps -e -o pid,ppid,command
```

The `-e` option instructs `ps` to display all processes running on the system. The `-o pid,ppid,command` option tells `ps` what information to show about each process—in this case, the process ID, the parent process ID, and the command running in this process.

ps Output Formats

With the `-o` option to the `ps` command, you specify the information about processes that you want in the output as a comma-separated list. For example, `ps -o pid,user,start_time,command` displays the process ID, the name of the user owning the process, the wall clock time at which the process started, and the command running in the process. See the man page for `ps` for the full list of field codes. You can use the `-f` (full listing), `-l` (long listing), or `-j` (jobs listing) options instead to get three different preset listing formats.

Here are the first few lines and last few lines of output from this command on my system. You may see different output, depending on what's running on your system.

```
% ps -e -o pid,ppid,command
  PID  PPID COMMAND
    1     0 init [5]
    2     1 [kflushd]
    3     1 [kupdate]
...
21725 21693 xterm
21727 21725 bash
21728 21727 ps -e -o pid,ppid,command
```

Note that the parent process ID of the `ps` command, 21727, is the process ID of `bash`, the shell from which I invoked `ps`. The parent process ID of `bash` is in turn 21725, the process ID of the `xterm` program in which the shell is running.

3.1.3 Killing a Process

You can kill a running process with the `kill` command. Simply specify on the command line the process ID of the process to be killed.

The `kill` command works by sending the process a `SIGTERM`, or termination, signal.[1] This causes the process to terminate, unless the executing program explicitly handles or masks the `SIGTERM` signal. Signals are described in Section 3.3, "Signals."

1. You can also use the `kill` command to send other signals to a process. This is described in Section 3.4, "Process Termination."

3.2 Creating Processes

Two common techniques are used for creating a new process. The first is relatively simple but should be used sparingly because it is inefficient and has considerably security risks. The second technique is more complex but provides greater flexibility, speed, and security.

3.2.1 Using *system*

The `system` function in the standard C library provides an easy way to execute a command from within a program, much as if the command had been typed into a shell. In fact, `system` creates a subprocess running the standard Bourne shell (`/bin/sh`) and hands the command to that shell for execution. For example, this program in Listing 3.2 invokes the `ls` command to display the contents of the root directory, as if you typed `ls -l /` into a shell.

Listing 3.2 **(*system.c*) Using the *system* Call**

```
#include <stdlib.h>

int main ()
{
  int return_value;
  return_value = system ("ls -l /");
  return return_value;
}
```

The `system` function returns the exit status of the shell command. If the shell itself cannot be run, `system` returns 127; if another error occurs, `system` returns –1.

Because the `system` function uses a shell to invoke your command, it's subject to the features, limitations, and security flaws of the system's shell. You can't rely on the availability of any particular version of the Bourne shell. On many UNIX systems, `/bin/sh` is a symbolic link to another shell. For instance, on most GNU/Linux systems, `/bin/sh` points to `bash` (the Bourne-Again SHell), and different GNU/Linux distributions use different versions of `bash`. Invoking a program with root privilege with the `system` function, for instance, can have different results on different GNU/Linux systems. Therefore, it's preferable to use the `fork` and `exec` method for creating processes.

3.2.2 Using *fork* and *exec*

The DOS and Windows API contains the `spawn` family of functions. These functions take as an argument the name of a program to run and create a new process instance of that program. Linux doesn't contain a single function that does all this in one step. Instead, Linux provides one function, `fork`, that makes a child process that is an exact copy of its parent process. Linux provides another set of functions, the `exec` family, that causes a particular process to cease being an instance of one program and to instead become an instance of another program. To spawn a new process, you first use `fork` to make a copy of the current process. Then you use `exec` to transform one of these processes into an instance of the program you want to spawn.

Calling *fork*

When a program calls `fork`, a duplicate process, called the *child process*, is created. The parent process continues executing the program from the point that `fork` was called. The child process, too, executes the same program from the same place.

So how do the two processes differ? First, the child process is a new process and therefore has a new process ID, distinct from its parent's process ID. One way for a program to distinguish whether it's in the parent process or the child process is to call `getpid`. However, the `fork` function provides different return values to the parent and child processes—one process "goes in" to the `fork` call, and two processes "come out," with different return values. The return value in the parent process is the process ID of the child. The return value in the child process is zero. Because no process ever has a process ID of zero, this makes it easy for the program whether it is now running as the parent or the child process.

Listing 3.3 is an example of using `fork` to duplicate a program's process. Note that the first block of the `if` statement is executed only in the parent process, while the `else` clause is executed in the child process.

Listing 3.3 **(*fork.c*) Using *fork* to Duplicate a Program's Process**

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

int main ()
{
  pid_t child_pid;

  printf ("the main program process ID is %d\n", (int) getpid ());

  child_pid = fork ();
  if (child_pid != 0) {
    printf ("this is the parent process, with id %d\n", (int) getpid ());
    printf ("the child's process ID is %d\n", (int) child_pid);
  }
  else
    printf ("this is the child process, with id %d\n", (int) getpid ());

  return 0;
}
```

Using the *exec* Family

The `exec` functions replace the program running in a process with another program. When a program calls an `exec` function, that process immediately ceases executing that program and begins executing a new program from the beginning, assuming that the `exec` call doesn't encounter an error.

Within the `exec` family, there are functions that vary slightly in their capabilities and how they are called.

- Functions that contain the letter *p* in their names (`execvp` and `execlp`) accept a program name and search for a program by that name in the current execution path; functions that don't contain the *p* must be given the full path of the program to be executed.
- Functions that contain the letter `v` in their names (`execv`, `execvp`, and `execve`) accept the argument list for the new program as a NULL-terminated array of pointers to strings. Functions that contain the letter *l* (`execl`, `execlp`, and `execle`) accept the argument list using the C language's varargs mechanism.
- Functions that contain the letter *e* in their names (`execve` and `execle`) accept an additional argument, an array of environment variables. The argument should be a NULL-terminated array of pointers to character strings. Each character string should be of the form "`VARIABLE=value`".

Because `exec` replaces the calling program with another one, it never returns unless an error occurs.

The argument list passed to the program is analogous to the command-line arguments that you specify to a program when you run it from the shell. They are available through the `argc` and `argv` parameters to `main`. Remember, when a program is invoked from the shell, the shell sets the first element of the argument list `argv[0]`) to the name of the program, the second element of the argument list (`argv[1]`) to the first command-line argument, and so on. When you use an `exec` function in your programs, you, too, should pass the name of the function as the first element of the argument list.

Using *fork* and *exec* Together

A common pattern to run a subprogram within a program is first to fork the process and then exec the subprogram. This allows the calling program to continue execution in the parent process while the calling program is replaced by the subprogram in the child process.

The program in Listing 3.4, like Listing 3.2, lists the contents of the root directory using the `ls` command. Unlike the previous example, though, it invokes the `ls` command directly, passing it the command-line arguments `-l` and `/` rather than invoking it through a shell.

Listing 3.4 (*fork-exec.c*) **Using *fork* and *exec* Together**

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

/* Spawn a child process running a new program. PROGRAM is the name
   of the program to run; the path will be searched for this program.
   ARG_LIST is a NULL-terminated list of character strings to be
   passed as the program's argument list. Returns the process ID of
   the spawned process.  */

int spawn (char* program, char** arg_list)
{
  pid_t child_pid;

  /* Duplicate this process.  */
  child_pid = fork ();
  if (child_pid != 0)
    /* This is the parent process.  */
    return child_pid;
  else {
    /* Now execute PROGRAM, searching for it in the path.  */
    execvp (program, arg_list);
    /* The execvp function returns only if an error occurs.  */
    fprintf (stderr, "an error occurred in execvp\n");
    abort ();
  }
}

int main ()
{
  /* The argument list to pass to the "ls" command.  */
  char* arg_list[] = {
    "ls",     /* argv[0], the name of the program.  */
    "-l",
    "/",
    NULL      /* The argument list must end with a NULL.  */
  };

  /* Spawn a child process running the "ls" command.  Ignore the
     returned child process ID.  */
  spawn ("ls", arg_list);

  printf ("done with main program\n");

  return 0;
}
```

3.2.3 Process Scheduling

Linux schedules the parent and child processes independently; there's no guarantee of which one will run first, or how long it will run before Linux interrupts it and lets the other process (or some other process on the system) run. In particular, none, part, or all of the `ls` command may run in the child process before the parent completes.[2] Linux promises that each process will run eventually—no process will be completely starved of execution resources.

You may specify that a process is less important—and should be given a lower priority—by assigning it a higher *niceness* value. By default, every process has a niceness of zero. A higher niceness value means that the process is given a lesser execution priority; conversely, a process with a lower (that is, negative) niceness gets more execution time.

To run a program with a nonzero niceness, use the `nice` command, specifying the niceness value with the `-n` option. For example, this is how you might invoke the command "`sort input.txt > output.txt`", a long sorting operation, with a reduced priority so that it doesn't slow down the system too much:

```
% nice -n 10 sort input.txt > output.txt
```

You can use the `renice` command to change the niceness of a running process from the command line.

To change the niceness of a running process programmatically, use the `nice` function. Its argument is an increment value, which is added to the niceness value of the process that calls it. Remember that a positive value raises the niceness value and thus reduces the process's execution priority.

Note that only a process with root privilege can run a process with a negative niceness value or reduce the niceness value of a running process. This means that you may specify negative values to the `nice` and `renice` commands only when logged in as root, and only a process running as root can pass a negative value to the `nice` function. This prevents ordinary users from grabbing execution priority away from others using the system.

3.3 Signals

Signals are mechanisms for communicating with and manipulating processes in Linux. The topic of signals is a large one; here we discuss some of the most important signals and techniques that are used for controlling processes.

A signal is a special message sent to a process. Signals are asynchronous; when a process receives a signal, it processes the signal immediately, without finishing the current function or even the current line of code. There are several dozen different signals, each with a different meaning. Each signal type is specified by its signal number, but in programs, you usually refer to a signal by its name. In Linux, these are defined in `/usr/include/bits/signum.h`. (You shouldn't include this header file directly in your programs; instead, use `<signal.h>`.)

2. A method for serializing the two processes is presented in Section 3.4.1, "Waiting for Process Termination."

When a process receives a signal, it may do one of several things, depending on the signal's *disposition*. For each signal, there is a *default disposition*, which determines what happens to the process if the program does not specify some other behavior. For most signal types, a program may specify some other behavior—either to ignore the signal or to call a special *signal-handler* function to respond to the signal. If a signal handler is used, the currently executing program is paused, the signal handler is executed, and, when the signal handler returns, the program resumes.

The Linux system sends signals to processes in response to specific conditions. For instance, `SIGBUS` (bus error), `SIGSEGV` (segmentation violation), and `SIGFPE` (floating point exception) may be sent to a process that attempts to perform an illegal operation. The default disposition for these signals it to terminate the process and produce a core file.

A process may also send a signal to another process. One common use of this mechanism is to end another process by sending it a `SIGTERM` or `SIGKILL` signal.[3] Another common use is to send a command to a running program. Two "user-defined" signals are reserved for this purpose: `SIGUSR1` and `SIGUSR2`. The `SIGHUP` signal is sometimes used for this purpose as well, commonly to wake up an idling program or cause a program to reread its configuration files.

The `sigaction` function can be used to set a signal disposition. The first parameter is the signal number. The next two parameters are pointers to `sigaction` structures; the first of these contains the desired disposition for that signal number, while the second receives the previous disposition. The most important field in the first or second `sigaction` structure is `sa_handler`. It can take one of three values:

- `SIG_DFL`, which specifies the default disposition for the signal.
- `SIG_IGN`, which specifies that the signal should be ignored.
- A pointer to a signal-handler function. The function should take one parameter, the signal number, and return `void`.

Because signals are asynchronous, the main program may be in a very fragile state when a signal is processed and thus while a signal handler function executes. Therefore, you should avoid performing any I/O operations or calling most library and system functions from signal handlers.

A signal handler should perform the minimum work necessary to respond to the signal, and then return control to the main program (or terminate the program). In most cases, this consists simply of recording the fact that a signal occurred. The main program then checks periodically whether a signal has occurred and reacts accordingly.

It is possible for a signal handler to be interrupted by the delivery of another signal. While this may sound like a rare occurrence, if it does occur, it will be very difficult to diagnose and debug the problem. (This is an example of a race condition, discussed in Chapter 4, "Threads," Section 4.4, "Synchronization and Critical Sections.") Therefore, you should be very careful about what your program does in a signal handler.

3. What's the difference? The `SIGTERM` signal asks a process to terminate; the process may ignore the request by masking or ignoring the signal. The `SIGKILL` signal always kills the process immediately because the process may not mask or ignore `SIGKILL`.

Even assigning a value to a global variable can be dangerous because the assignment may actually be carried out in two or more machine instructions, and a second signal may occur between them, leaving the variable in a corrupted state. If you use a global variable to flag a signal from a signal-handler function, it should be of the special type `sig_atomic_t`. Linux guarantees that assignments to variables of this type are performed in a single instruction and therefore cannot be interrupted midway. In Linux, `sig_atomic_t` is an ordinary `int`; in fact, assignments to integer types the size of `int` or smaller, or to pointers, are atomic. If you want to write a program that's portable to any standard UNIX system, though, use `sig_atomic_t` for these global variables.

This program skeleton in Listing 3.5, for instance, uses a signal-handler function to count the number of times that the program receives `SIGUSR1`, one of the signals reserved for application use.

Listing 3.5 **(*sigusr1.c*) Using a Signal Handler**

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

sig_atomic_t sigusr1_count = 0;

void handler (int signal_number)
{
  ++sigusr1_count;
}

int main ()
{
  struct sigaction sa;
  memset (&sa, 0, sizeof (sa));
  sa.sa_handler = &handler;
  sigaction (SIGUSR1, &sa, NULL);

  /* Do some lengthy stuff here.  */
  /* ...  */

  printf ("SIGUSR1 was raised %d times\n", sigusr1_count);
  return 0;
}
```

3.4 Process Termination

Normally, a process terminates in one of two ways. Either the executing program calls the `exit` function, or the program's `main` function returns. Each process has an exit code: a number that the process returns to its parent. The exit code is the argument passed to the `exit` function, or the value returned from `main`.

A process may also terminate abnormally, in response to a signal. For instance, the `SIGBUS`, `SIGSEGV`, and `SIGFPE` signals mentioned previously cause the process to terminate. Other signals are used to terminate a process explicitly. The `SIGINT` signal is sent to a process when the user attempts to end it by typing Ctrl+C in its terminal. The `SIGTERM` signal is sent by the `kill` command. The default disposition for both of these is to terminate the process. By calling the `abort` function, a process sends itself the `SIGABRT` signal, which terminates the process and produces a core file. The most powerful termination signal is `SIGKILL`, which ends a process immediately and cannot be blocked or handled by a program.

Any of these signals can be sent using the `kill` command by specifying an extra command-line flag; for instance, to end a troublesome process by sending it a `SIGKILL`, invoke the following, where `pid` is its process ID:

```
% kill -KILL pid
```

To send a signal from a program, use the `kill` function. The first parameter is the target process ID. The second parameter is the signal number; use `SIGTERM` to simulate the default behavior of the `kill` command. For instance, where `child pid` contains the process ID of the child process, you can use the `kill` function to terminate a child process from the parent by calling it like this:

```
kill (child_pid, SIGTERM);
```

Include the <sys/types.h> and <signal.h> headers if you use the `kill` function.

By convention, the exit code is used to indicate whether the program executed correctly. An exit code of zero indicates correct execution, while a nonzero exit code indicates that an error occurred. In the latter case, the particular value returned may give some indication of the nature of the error. It's a good idea to stick with this convention in your programs because other components of the GNU/Linux system assume this behavior. For instance, shells assume this convention when you connect multiple programs with the `&&` (logical and) and `||` (logical or) operators. Therefore, you should explicitly return zero from your `main` function, unless an error occurs.

With most shells, it's possible to obtain the exit code of the most recently executed program using the special `$?` variable. Here's an example in which the `ls` command is invoked twice and its exit code is displayed after each invocation. In the first case, `ls` executes correctly and returns the exit code zero. In the second case, `ls` encounters an error (because the filename specified on the command line does not exist) and thus returns a nonzero exit code.

```
% ls /
bin   coda  etc   lib         misc  nfs  proc  sbin  usr
boot  dev   home  lost+found  mnt   opt  root  tmp   var
% echo $?
0
% ls bogusfile
ls: bogusfile: No such file or directory
% echo $?
1
```

Note that even though the parameter type of the `exit` function is `int` and the `main` function returns an `int`, Linux does not preserve the full 32 bits of the return code. In fact, you should use exit codes only between zero and 127. Exit codes above 128 have a special meaning—when a process is terminated by a signal, its exit code is 128 plus the signal number.

3.4.1 Waiting for Process Termination

If you typed in and ran the `fork` and `exec` example in Listing 3.4, you may have noticed that the output from the `ls` program often appears after the "main program" has already completed. That's because the child process, in which `ls` is run, is scheduled independently of the parent process. Because Linux is a multitasking operating system, both processes appear to execute simultaneously, and you can't predict whether the `ls` program will have a chance to run before or after the parent process runs.

In some situations, though, it is desirable for the parent process to wait until one or more child processes have completed. This can be done with the `wait` family of system calls. These functions allow you to wait for a process to finish executing, and enable the parent process to retrieve information about its child's termination. There are four different system calls in the `wait` family; you can choose to get a little or a lot of information about the process that exited, and you can choose whether you care about which child process terminated.

3.4.2 The *wait* System Calls

The simplest such function is called simply `wait`. It blocks the calling process until one of its child processes exits (or an error occurs). It returns a status code via an integer pointer argument, from which you can extract information about how the child process exited. For instance, the `WEXITSTATUS` macro extracts the child process's exit code.

You can use the `WIFEXITED` macro to determine from a child process's exit status whether that process exited normally (via the `exit` function or returning from `main`) or died from an unhandled signal. In the latter case, use the `WTERMSIG` macro to extract from its exit status the signal number by which it died.

Here is the `main` function from the `fork` and `exec` example again. This time, the parent process calls `wait` to wait until the child process, in which the `ls` command executes, is finished.

```
int main ()
{
  int child_status;

  /* The argument list to pass to the "ls" command.  */
  char* arg_list[] = {
    "ls",     /* argv[0], the name of the program.  */
    "-l",
    "/",
    NULL      /* The argument list must end with a NULL.  */
  };

  /* Spawn a child process running the "ls" command.  Ignore the
     returned child process ID.  */
  spawn ("ls", arg_list);

  /* Wait for the child process to complete.  */
  wait (&child_status);
  if (WIFEXITED (child_status))
    printf ("the child process exited normally, with exit code %d\n",
            WEXITSTATUS (child_status));
  else
    printf ("the child process exited abnormally\n");

  return 0;
}
```

Several similar system calls are available in Linux, which are more flexible or provide more information about the exiting child process. The `waitpid` function can be used to wait for a specific child process to exit instead of any child process. The `wait3` function returns CPU usage statistics about the exiting child process, and the `wait4` function allows you to specify additional options about which processes to wait for.

3.4.3 Zombie Processes

If a child process terminates while its parent is calling a `wait` function, the child process vanishes and its termination status is passed to its parent via the `wait` call. But what happens when a child process terminates and the parent is not calling `wait`? Does it simply vanish? No, because then information about its termination—such as whether it exited normally and, if so, what its exit status is—would be lost. Instead, when a child process terminates, is becomes a zombie process.

A *zombie process* is a process that has terminated but has not been cleaned up yet. It is the responsibility of the parent process to clean up its zombie children. The `wait` functions do this, too, so it's not necessary to track whether your child process is still executing before waiting for it. Suppose, for instance, that a program forks a child process, performs some other computations, and then calls `wait`. If the child process has not terminated at that point, the parent process will block in the `wait` call until the child process finishes. If the child process finishes before the parent process calls `wait`, the child process becomes a zombie. When the parent process calls `wait`, the zombie child's termination status is extracted, the child process is deleted, and the `wait` call returns immediately.

What happens if the parent does not clean up its children? They stay around in the system, as zombie processes. The program in Listing 3.6 forks a child process, which terminates immediately and then goes to sleep for a minute, without ever cleaning up the child process.

Listing 3.6 **(*zombie.c*) Making a Zombie Process**

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

int main ()
{
  pid_t child_pid;

  /* Create a child process.  */
  child_pid = fork ();
  if (child_pid > 0) {
    /* This is the parent process.  Sleep for a minute.  */
    sleep (60);
  }
  else {
    /* This is the child process.  Exit immediately.  */
    exit (0);
  }
  return 0;
}
```

Try compiling this file to an executable named `make-zombie`. Run it, and while it's still running, list the processes on the system by invoking the following command in another window:

```
% ps -e -o pid,ppid,stat,cmd
```

This lists the process ID, parent process ID, process status, and process command line. Observe that, in addition to the parent `make-zombie` process, there is another `make-zombie` process listed. It's the child process; note that its parent process ID is the process ID of the main `make-zombie` process. The child process is marked as `<defunct>`, and its status code is Z, for zombie.

What happens when the main `make-zombie` program ends when the parent process exits, without ever calling `wait`? Does the zombie process stay around? No—try running `ps` again, and note that both of the `make-zombie` processes are gone. When a program exits, its children are inherited by a special process, the `init` program, which always runs with process ID of 1 (it's the first process started when Linux boots). The `init` process automatically cleans up any zombie child processes that it inherits.

3.4.4 Cleaning Up Children Asynchronously

If you're using a child process simply to `exec` another program, it's fine to call `wait` immediately in the parent process, which will block until the child process completes. But often, you'll want the parent process to continue running, as one or more children execute synchronously. How can you be sure that you clean up child processes that have completed so that you don't leave zombie processes, which consume system resources, lying around?

One approach would be for the parent process to call `wait3` or `wait4` periodically, to clean up zombie children. Calling `wait` for this purpose doesn't work well because, if no children have terminated, the call will block until one does. However, `wait3` and `wait4` take an additional flag parameter, to which you can pass the flag value `WNOHANG`. With this flag, the function runs in *nonblocking mode*—it will clean up a terminated child process if there is one, or simply return if there isn't. The return value of the call is the process ID of the terminated child in the former case, or zero in the latter case.

A more elegant solution is to notify the parent process when a child terminates. There are several ways to do this using the methods discussed in Chapter 5, "Interprocess Communication," but fortunately Linux does this for you, using signals. When a child process terminates, Linux sends the parent process the `SIGCHLD` signal. The default disposition of this signal is to do nothing, which is why you might not have noticed it before.

Thus, an easy way to clean up child processes is by handling `SIGCHLD`. Of course, when cleaning up the child process, it's important to store its termination status if this information is needed, because once the process is cleaned up using `wait`, that information is no longer available. Listing 3.7 is what it looks like for a program to use a `SIGCHLD` handler to clean up its child processes.

Listing 3.7 (*sigchld.c*) **Cleaning Up Children by Handling** ***SIGCHLD***

```
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

sig_atomic_t child_exit_status;

void clean_up_child_process (int signal_number)
{
  /* Clean up the child process.  */
  int status;
  wait (&status);
  /* Store its exit status in a global variable.  */
  child_exit_status = status;
}

int main ()
{
  /* Handle SIGCHLD by calling clean_up_child_process.  */
  struct sigaction sigchld_action;
  memset (&sigchld_action, 0, sizeof (sigchld_action));
  sigchld_action.sa_handler = &clean_up_child_process;
  sigaction (SIGCHLD, &sigchld_action, NULL);

  /* Now do things, including forking a child process.  */
  /* ...  */

  return 0;
}
```

Note how the signal handler stores the child process's exit status in a global variable, from which the main program can access it. Because the variable is assigned in a signal handler, its type is `sig_atomic_t`.

4 Threads

THREADS, LIKE PROCESSES, ARE A MECHANISM TO ALLOW A PROGRAM to do more than one thing at a time. As with processes, threads appear to run concurrently; the Linux kernel schedules them asynchronously, interrupting each thread from time to time to give others a chance to execute.

Conceptually, a thread exists within a process. Threads are a finer-grained unit of execution than processes. When you invoke a program, Linux creates a new process and in that process creates a single thread, which runs the program sequentially. That thread can create additional threads; all these threads run the same program in the same process, but each thread may be executing a different part of the program at any given time.

We've seen how a program can fork a child process. The child process is initially running its parent's program, with its parent's virtual memory, file descriptors, and so on copied. The child process can modify its memory, close file descriptors, and the like without affecting its parent, and vice versa. When a program creates another thread, though, nothing is copied. The creating and the created thread share the same memory space, file descriptors, and other system resources as the original. If one thread changes the value of a variable, for instance, the other thread subsequently will see the modified value. Similarly, if one thread closes a file descriptor, other threads may not read from or write to that file descriptor. Because a process and all its threads can be executing only one program at a time, if any thread inside a process calls one of the `exec` functions, all the other threads are ended (the new program may, of course, create new threads).

GNU/Linux implements the POSIX standard thread API (known as *pthreads*). All thread functions and data types are declared in the header file `<pthread.h>`. The pthread functions are not included in the standard C library. Instead, they are in `libpthread`, so you should add `-lpthread` to the command line when you link your program.

4.1 Thread Creation

Each thread in a process is identified by a *thread ID*. When referring to thread IDs in C or C++ programs, use the type `pthread_t`.

Upon creation, each thread executes a *thread function*. This is just an ordinary function and contains the code that the thread should run. When the function returns, the thread exits. On GNU/Linux, thread functions take a single parameter, of type `void*`, and have a `void*` return type. The parameter is the *thread argument*: GNU/Linux passes the value along to the thread without looking at it. Your program can use this parameter to pass data to a new thread. Similarly, your program can use the return value to pass data from an exiting thread back to its creator.

The `pthread_create` function creates a new thread. You provide it with the following:

1. A pointer to a `pthread_t` variable, in which the thread ID of the new thread is stored.
2. A pointer to a *thread attribute* object. This object controls details of how the thread interacts with the rest of the program. If you pass `NULL` as the thread attribute, a thread will be created with the default thread attributes. Thread attributes are discussed in Section 4.1.5, "Thread Attributes."
3. A pointer to the thread function. This is an ordinary function pointer, of this type:

   ```
   void* (*) (void*)
   ```

4. A thread argument value of type `void*`. Whatever you pass is simply passed as the argument to the thread function when the thread begins executing.

A call to `pthread_create` returns immediately, and the original thread continues executing the instructions following the call. Meanwhile, the new thread begins executing the thread function. Linux schedules both threads asynchronously, and your program must not rely on the relative order in which instructions are executed in the two threads.

The program in Listing 4.1 creates a thread that prints x's continuously to standard error. After calling `pthread_create`, the main thread prints o's continuously to standard error.

Listing 4.1 **(*thread-create.c*) Create a Thread**

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

/* Prints x's to stderr.  The parameter is unused.  Does not return.  */

void* print_xs (void* unused)
{
  while (1)
    fputc ('x', stderr);
  return NULL;
}

/* The main program.  */

int main ()
{
  pthread_t thread_id;
  /* Create a new thread.  The new thread will run the print_xs
     function.  */
  pthread_create (&thread_id, NULL, &print_xs, NULL);
  /* Print o's continuously to stderr.  */
  while (1)
    fputc ('o', stderr);
  return 0;
}
```

Compile and link this program using the following code:

```
% cc -o thread-create thread-create.c -lpthread
```

Try running it to see what happens. Notice the unpredictable pattern of x's and o's as Linux alternately schedules the two threads.

Under normal circumstances, a thread exits in one of two ways. One way, as illustrated previously, is by returning from the thread function. The return value from the thread function is taken to be the return value of the thread. Alternately, a thread can exit explicitly by calling `pthread_exit`. This function may be called from within the thread function or from some other function called directly or indirectly by the thread function. The argument to `pthread_exit` is the thread's return value.

4.1.1 Passing Data to Threads

The thread argument provides a convenient method of passing data to threads. Because the type of the argument is `void*`, though, you can't pass a lot of data directly via the argument. Instead, use the thread argument to pass a pointer to some structure or array of data. One commonly used technique is to define a structure for each thread function, which contains the "parameters" that the thread function expects.

Using the thread argument, it's easy to reuse the same thread function for many threads. All these threads execute the same code, but on different data.

The program in Listing 4.2 is similar to the previous example. This one creates two new threads, one to print x's and the other to print o's. Instead of printing infinitely, though, each thread prints a fixed number of characters and then exits by returning from the thread function. The same thread function, `char_print`, is used by both threads, but each is configured differently using `struct char_print_parms`.

Listing 4.2 **(*thread-create2*) Create Two Threads**

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

/* Parameters to print_function.  */

struct char_print_parms
{
  /* The character to print.  */
  char character;
  /* The number of times to print it.  */
  int count;
};

/* Prints a number of characters to stderr, as given by PARAMETERS,
   which is a pointer to a struct char_print_parms.  */

void* char_print (void* parameters)
{
  /* Cast the cookie pointer to the right type.  */
  struct char_print_parms* p = (struct char_print_parms*) parameters;
  int i;

  for (i = 0; i < p->count; ++i)
    fputc (p->character, stderr);
  return NULL;
}

/* The main program.  */

int main ()
{
  pthread_t thread1_id;
```

```
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;

  /* Create a new thread to print 30,000 'x's.  */
  thread1_args.character = 'x';
  thread1_args.count = 30000;
  pthread_create (&thread1_id, NULL, &char_print, &thread1_args);

  /* Create a new thread to print 20,000 o's.  */
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print, &thread2_args);

  return 0;
}
```

But wait! The program in Listing 4.2 has a serious bug in it. The main thread (which runs the `main` function) creates the thread parameter structures (`thread1_args` and `thread2_args`) as local variables, and then passes pointers to these structures to the threads it creates. What's to prevent Linux from scheduling the three threads in such a way that `main` finishes executing before either of the other two threads are done? *Nothing!* But if this happens, the memory containing the thread parameter structures will be deallocated while the other two threads are still accessing it.

4.1.2 Joining Threads

One solution is to force `main` to wait until the other two threads are done. What we need is a function similar to `wait` that waits for a thread to finish instead of a process. That function is `pthread_join`, which takes two arguments: the thread ID of the thread to wait for, and a pointer to a `void*` variable that will receive the finished thread's return value. If you don't care about the thread return value, pass `NULL` as the second argument.

Listing 4.3 shows the corrected `main` function for the buggy example in Listing 4.2. In this version, `main` does not exit until both of the threads printing x's and o's have completed, so they are no longer using the argument structures.

Listing 4.3 **Revised *Main* Function for *thread-create2.c***

```
int main ()
{
  pthread_t thread1_id;
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;
```

continues

Listing 4.3 **Continued**

```
  /* Create a new thread to print 30,000 x's.  */
  thread1_args.character = 'x';
  thread1_args.count = 30000;
  pthread_create (&thread1_id, NULL, &char_print, &thread1_args);

  /* Create a new thread to print 20,000 o's.  */
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print, &thread2_args);

  /* Make sure the first thread has finished.  */
  pthread_join (thread1_id, NULL);
  /* Make sure the second thread has finished.  */
  pthread_join (thread2_id, NULL);

  /* Now we can safely return.  */
  return 0;
}
```

The moral of the story: Make sure that any data you pass to a thread by reference is not deallocated, *even by a different thread*, until you're sure that the thread is done with it. This is true both for local variables, which are deallocated when they go out of scope, and for heap-allocated variables, which you deallocate by calling `free` (or using `delete` in C++).

4.1.3 Thread Return Values

If the second argument you pass to `pthread_join` is non-null, the thread's return value will be placed in the location pointed to by that argument. The thread return value, like the thread argument, is of type `void*`. If you want to pass back a single `int` or other small number, you can do this easily by casting the value to `void*` and then casting back to the appropriate type after calling `pthread_join`.[1]

The program in Listing 4.4 computes the *n*th prime number in a separate thread. That thread returns the desired prime number as its thread return value. The main thread, meanwhile, is free to execute other code. Note that the successive division algorithm used in `compute_prime` is quite inefficient; consult a book on numerical algorithims if you need to compute many prime numbers in your programs.

1. Note that this is not portable, and it's up to you to make sure that your value can be cast safely to `void*` and back without losing bits.

Listing 4.4 (*primes.c*) **Compute Prime Numbers in a Thread**

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

/* Compute successive prime numbers (very inefficiently).  Return the
   Nth prime number, where N is the value pointed to by *ARG.  */

void* compute_prime (void* arg)
{
  int candidate = 2;
  int n = *((int*) arg);

  while (1) {
    int factor;
    int is_prime = 1;

    /* Test primality by successive division.  */
    for (factor = 2; factor < candidate; ++factor)
      if (candidate % factor == 0) {
        is_prime = 0;
        break;
      }
    /* Is this the prime number we're looking for?  */
    if (is_prime) {
      if (--n == 0)
        /* Return the desired prime number as the thread return value.  */
        return (void*) candidate;
    }
    ++candidate;
  }
  return NULL;
}

int main ()
{
  pthread_t thread;
  int which_prime = 5000;
  int prime;

  /* Start the computing thread, up to the 5,000th prime number.  */
  pthread_create (&thread, NULL, &compute_prime, &which_prime);
  /* Do some other work here...  */
  /* Wait for the prime number thread to complete, and get the result.  */
  pthread_join (thread, (void*) &prime);
  /* Print the largest prime it computed.  */
  printf("The %dth prime number is %d.\n", which_prime, prime);
  return 0;
}
```

4.1.4 More on Thread IDs

Occasionally, it is useful for a sequence of code to determine which thread is executing it. The `pthread_self` function returns the thread ID of the thread in which it is called. This thread ID may be compared with another thread ID using the `pthread_equal` function.

These functions can be useful for determining whether a particular thread ID corresponds to the current thread. For instance, it is an error for a thread to call `pthread_join` to join itself. (In this case, `pthread_join` would return the error code `EDEADLK`.) To check for this beforehand, you might use code like this:

```
if (!pthread_equal (pthread_self (), other_thread))
  pthread_join (other_thread, NULL);
```

4.1.5 Thread Attributes

Thread attributes provide a mechanism for fine-tuning the behavior of individual threads. Recall that `pthread_create` accepts an argument that is a pointer to a thread attribute object. If you pass a null pointer, the default thread attributes are used to configure the new thread. However, you may create and customize a thread attribute object to specify other values for the attributes.

To specify customized thread attributes, you must follow these steps:

1. Create a `pthread_attr_t` object. The easiest way is simply to declare an automatic variable of this type.
2. Call `pthread_attr_init`, passing a pointer to this object. This initializes the attributes to their default values.
3. Modify the attribute object to contain the desired attribute values.
4. Pass a pointer to the attribute object when calling `pthread_create`.
5. Call `pthread_attr_destroy` to release the attribute object. The `pthread_attr_t` variable itself is not deallocated; it may be reinitialized with `pthread_attr_init`.

A single thread attribute object may be used to start several threads. It is not necessary to keep the thread attribute object around after the threads have been created.

For most GNU/Linux application programming tasks, only one thread attribute is typically of interest (the other available attributes are primarily for specialty real-time programming). This attribute is the thread's *detach state*. A thread may be created as a *joinable thread* (the default) or as a *detached thread*. A joinable thread, like a process, is not automatically cleaned up by GNU/Linux when it terminates. Instead, the thread's exit state hangs around in the system (kind of like a zombie process) until another thread calls `pthread_join` to obtain its return value. Only then are its resources released. A detached thread, in contrast, is cleaned up automatically when it terminates. Because a detached thread is immediately cleaned up, another thread may not synchronize on its completion by using `pthread_join` or obtain its return value.

To set the detach state in a thread attribute object, use `pthread_attr_setdetachstate`. The first argument is a pointer to the thread attribute object, and the second is the desired detach state. Because the joinable state is the default, it is necessary to call this only to create detached threads; pass `PTHREAD_CREATE_DETACHED` as the second argument.

The code in Listing 4.5 creates a detached thread by setting the detach state thread attribute for the thread.

Listing 4.5 **(*detached.c*) Skeleton Program That Creates a Detached Thread**

```
#include <pthread.h>

void* thread_function (void* thread_arg)
{
  /* Do work here...  */
}

int main ()
{
  pthread_attr_t attr;
  pthread_t thread;

  pthread_attr_init (&attr);
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);
  pthread_create (&thread, &attr, &thread_function, NULL);
  pthread_attr_destroy (&attr);

  /* Do work here...  */

  /* No need to join the second thread.  */
  return 0;
}
```

Even if a thread is created in a joinable state, it may later be turned into a detached thread. To do this, call `pthread_detach`. Once a thread is detached, it cannot be made joinable again.

4.2 Thread Cancellation

Under normal circumstances, a thread terminates when it exits normally, either by returning from its thread function or by calling `pthread_exit`. However, it is possible for a thread to request that another thread terminate. This is called *canceling* a thread.

To cancel a thread, call `pthread_cancel`, passing the thread ID of the thread to be canceled. A canceled thread may later be joined; in fact, you should join a canceled thread to free up its resources, unless the thread is detached (see Section 4.1.5, "Thread Attributes"). The return value of a canceled thread is the special value given by `PTHREAD_CANCELED`.

Often a thread may be in some code that must be executed in an all-or-nothing fashion. For instance, the thread may allocate some resources, use them, and then deallocate them. If the thread is canceled in the middle of this code, it may not have the opportunity to deallocate the resources, and thus the resources will be leaked. To counter this possibility, it is possible for a thread to control whether and when it can be canceled.

A thread may be in one of three states with regard to thread cancellation.

- The thread may be *asynchronously cancelable*. The thread may be canceled at any point in its execution.
- The thread may be *synchronously cancelable*. The thread may be canceled, but not at just any point in its execution. Instead, cancellation requests are queued, and the thread is canceled only when it reaches specific points in its execution.
- A thread may be *uncancelable*. Attempts to cancel the thread are quietly ignored.

When initially created, a thread is synchronously cancelable.

4.2.1 Synchronous and Asynchronous Threads

An asynchronously cancelable thread may be canceled at any point in its execution. A synchronously cancelable thread, in contrast, may be canceled only at particular places in its execution. These places are called *cancellation points*. The thread will queue a cancellation request until it reaches the next cancellation point.

To make a thread asynchronously cancelable, use `pthread_setcanceltype`. This affects the thread that actually calls the function. The first argument should be `PTHREAD_CANCEL_ASYNCHRONOUS` to make the thread asynchronously cancelable, or `PTHREAD_CANCEL_DEFERRED` to return it to the synchronously cancelable state. The second argument, if not null, is a pointer to a variable that will receive the previous cancellation type for the thread. This call, for example, makes the calling thread asynchronously cancelable.

```
pthread_setcanceltype (PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
```

What constitutes a cancellation point, and where should these be placed? The most direct way to create a cancellation point is to call `pthread_testcancel`. This does nothing except process a pending cancellation in a synchronously cancelable thread. You should call `pthread_testcancel` periodically during lengthy computations in a thread function, at points where the thread can be canceled without leaking any resources or producing other ill effects.

Certain other functions are implicitly cancellation points as well. These are listed on the `pthread_cancel` man page. Note that other functions may use these functions internally and thus will indirectly be cancellation points.

4.2.2 Uncancelable Critical Sections

A thread may disable cancellation of itself altogether with the `pthread_setcancelstate` function. Like `pthread_setcanceltype`, this affects the calling thread. The first argument is `PTHREAD_CANCEL_DISABLE` to disable cancellation, or `PTHREAD_CANCEL_ENABLE` to re-enable cancellation. The second argument, if not null, points to a variable that will receive the previous cancellation state. This call, for instance, disables thread cancellation in the calling thread.

```
pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, NULL);
```

Using `pthread_setcancelstate` enables you to implement *critical sections*. A critical section is a sequence of code that must be executed either in its entirety or not at all; in other words, if a thread begins executing the critical section, it must continue until the end of the critical section without being canceled.

For example, suppose that you're writing a routine for a banking program that transfers money from one account to another. To do this, you must add value to the balance in one account and deduct the same value from the balance of another account. If the thread running your routine happened to be canceled at just the wrong time between these two operations, the program would have spuriously increased the bank's total deposits by failing to complete the transaction. To prevent this possibility, place the two operations in a critical section.

You might implement the transfer with a function such as `process_transaction`, shown in Listing 4.6. This function disables thread cancellation to start a critical section before it modifies either account balance.

Listing 4.6 (*critical-section.c*) **Protect a Bank Transaction with a Critical Section**

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>

/* An array of balances in accounts, indexed by account number.  */

float* account_balances;

/* Transfer DOLLARS from account FROM_ACCT to account TO_ACCT.  Return
   0 if the transaction succeeded, or 1 if the balance FROM_ACCT is
   too small.  */

int process_transaction (int from_acct, int to_acct, float dollars)
{
  int old_cancel_state;

  /* Check the balance in FROM_ACCT.  */
  if (account_balances[from_acct] < dollars)
    return 1;
```

continues

Listing 4.6 **Continued**

```
  /* Begin critical section.  */
  pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, &old_cancel_state);
  /* Move the money.  */
  account_balances[to_acct] += dollars;
  account_balances[from_acct] -= dollars;
  /* End critical section.  */
  pthread_setcancelstate (old_cancel_state, NULL);

  return 0;
}
```

Note that it's important to restore the old cancel state at the end of the critical section rather than setting it unconditionally to `PTHREAD_CANCEL_ENABLE`. This enables you to call the `process_transaction` function safely from within another critical section—in that case, your function will leave the cancel state the same way it found it.

4.2.3 When to Use Thread Cancellation

In general, it's a good idea not to use thread cancellation to end the execution of a thread, except in unusual circumstances. During normal operation, a better strategy is to indicate to the thread that it should exit, and then to wait for the thread to exit on its own in an orderly fashion. We'll discuss techniques for communicating with the thread later in this chapter, and in Chapter 5, "Interprocess Communication."

4.3 Thread-Specific Data

Unlike processes, all threads in a single program share the same address space. This means that if one thread modifies a location in memory (for instance, a global variable), the change is visible to all other threads. This allows multiple threads to operate on the same data without the use interprocess communication mechanisms (which are described in Chapter 5).

Each thread has its own call stack, however. This allows each thread to execute different code and to call and return from subroutines in the usual way. As in a single-threaded program, each invocation of a subroutine in each thread has its own set of local variables, which are stored on the stack for that thread.

Sometimes, however, it is desirable to duplicate a certain variable so that each thread has a separate copy. GNU/Linux supports this by providing each thread with a *thread-specific data* area. The variables stored in this area are duplicated for each thread, and each thread may modify its copy of a variable without affecting other threads. Because all threads share the same memory space, thread-specific data may not be accessed using normal variable references. GNU/Linux provides special functions for setting and retrieving values from the thread-specific data area.

You may create as many thread-specific data items as you want, each of type `void*`. Each item is referenced by a key. To create a new key, and thus a new data item for each thread, use `pthread_key_create`. The first argument is a pointer to a `pthread_key_t` variable. That key value can be used by each thread to access its own copy of the corresponding data item. The second argument to `pthread_key_t` is a cleanup function. If you pass a function pointer here, GNU/Linux automatically calls that function when each thread exits, passing the thread-specific value corresponding to that key. This is particularly handy because the cleanup function is called even if the thread is canceled at some arbitrary point in its execution. If the thread-specific value is null, the thread cleanup function is not called. If you don't need a cleanup function, you may pass null instead of a function pointer.

After you've created a key, each thread can set its thread-specific value corresponding to that key by calling `pthread_setspecific`. The first argument is the key, and the second is the `void*` thread-specific value to store. To retrieve a thread-specific data item, call `pthread_getspecific`, passing the key as its argument.

Suppose, for instance, that your application divides a task among multiple threads. For audit purposes, each thread is to have a separate log file, in which progress messages for that thread's tasks are recorded. The thread-specific data area is a convenient place to store the file pointer for the log file for each individual thread.

Listing 4.7 shows how you might implement this. The `main` function in this sample program creates a key to store the thread-specific file pointer and then stores it in `thread_log_key`. Because this is a global variable, it is shared by all threads. When each thread starts executing its thread function, it opens a log file and stores the file pointer under that key. Later, any of these threads may call `write_to_thread_log` to write a message to the thread-specific log file. That function retrieves the file pointer for the thread's log file from thread-specific data and writes the message.

Listing 4.7 **(*tsd.c*) Per-Thread Log Files Implemented with Thread-Specific Data**

```
#include <malloc.h>
#include <pthread.h>
#include <stdio.h>

/* The key used to associate a log file pointer with each thread.  */
static pthread_key_t thread_log_key;

/* Write MESSAGE to the log file for the current thread.  */

void write_to_thread_log (const char* message)
{
  FILE* thread_log = (FILE*) pthread_getspecific (thread_log_key);
  fprintf (thread_log, "%s\n", message);
}

/* Close the log file pointer THREAD_LOG.  */

void close_thread_log (void* thread_log)
```

continues

Listing 4.7 **Continued**

```
{
  fclose ((FILE*) thread_log);
}

void* thread_function (void* args)
{
  char thread_log_filename[20];
  FILE* thread_log;

  /* Generate the filename for this thread's log file.  */
  sprintf (thread_log_filename, "thread%d.log", (int) pthread_self ());
  /* Open the log file.  */
  thread_log = fopen (thread_log_filename, "w");
  /* Store the file pointer in thread-specific data under thread_log_key.  */
  pthread_setspecific (thread_log_key, thread_log);

  write_to_thread_log ("Thread starting.");
  /* Do work here... */

  return NULL;
}

int main ()
{
  int i;
  pthread_t threads[5];

  /* Create a key to associate thread log file pointers in
     thread-specific data.  Use close_thread_log to clean up the file
     pointers.  */
  pthread_key_create (&thread_log_key, close_thread_log);
  /* Create threads to do the work.  */
  for (i = 0; i < 5; ++i)
    pthread_create (&(threads[i]), NULL, thread_function, NULL);
  /* Wait for all threads to finish.  */
  for (i = 0; i < 5; ++i)
    pthread_join (threads[i], NULL);
  return 0;
}
```

Observe that `thread_function` does not need to close the log file. That's because when the log file key was created, `close_thread_log` was specified as the cleanup function for that key. Whenever a thread exits, GNU/Linux calls that function, passing the thread-specific value for the thread log key. This function takes care of closing the log file.

4.3.1 Cleanup Handlers

The cleanup functions for thread-specific data keys can be very handy for ensuring that resources are not leaked when a thread exits or is canceled. Sometimes, though, it's useful to be able to specify cleanup functions without creating a new thread-specific data item that's duplicated for each thread. GNU/Linux provides *cleanup handlers* for this purpose.

A cleanup handler is simply a function that should be called when a thread exits. The handler takes a single `void*` parameter, and its argument value is provided when the handler is registered—this makes it easy to use the same handler function to deallocate multiple resource instances.

A cleanup handler is a temporary measure, used to deallocate a resource only if the thread exits or is canceled instead of finishing execution of a particular region of code. Under normal circumstances, when the thread does not exit and is not canceled, the resource should be deallocated explicitly and the cleanup handler should be removed.

To register a cleanup handler, call `pthread_cleanup_push`, passing a pointer to the cleanup function and the value of its `void*` argument. The call to `pthread_cleanup_push` must be balanced by a corresponding call to `pthread_cleanup_pop`, which unregisters the cleanup handler. As a convenience, `pthread_cleanup_pop` takes an `int` flag argument; if the flag is nonzero, the cleanup action is actually performed as it is unregistered.

The program fragment in Listing 4.8 shows how you might use a cleanup handler to make sure that a dynamically allocated buffer is cleaned up if the thread terminates.

Listing 4.8 **(*cleanup.c*) Program Fragment Demonstrating a Thread Cleanup Handler**

```
#include <malloc.h>
#include <pthread.h>

/* Allocate a temporary buffer.  */

void* allocate_buffer (size_t size)
{
  return malloc (size);
}

/* Deallocate a temporary buffer.  */

void deallocate_buffer (void* buffer)
{
  free (buffer);
}

void do_some_work ()
{
  /* Allocate a temporary buffer.  */
```

continues

Listing 4.8 **Continued**

```
  void* temp_buffer = allocate_buffer (1024);
  /* Register a cleanup handler for this buffer, to deallocate it in
     case the thread exits or is cancelled.  */
  pthread_cleanup_push (deallocate_buffer, temp_buffer);

  /* Do some work here that might call pthread_exit or might be
     cancelled...  */

  /* Unregister the cleanup handler.  Because we pass a nonzero value,
     this actually performs the cleanup by calling
     deallocate_buffer.  */
  pthread_cleanup_pop (1);
}
```

Because the argument to `pthread_cleanup_pop` is nonzero in this case, the cleanup function `deallocate_buffer` is called automatically here and does not need to be called explicitly. In this simple case, we could have used the standard library function `free` directly as our cleanup handler function instead of `deallocate_buffer`.

4.3.2 Thread Cleanup in C++

C++ programmers are accustomed to getting cleanup "for free" by wrapping cleanup actions in object destructors. When the objects go out of scope, either because a block is executed to completion or because an exception is thrown, C++ makes sure that destructors are called for those automatic variables that have them. This provides a handy mechanism to make sure that cleanup code is called no matter how the block is exited.

If a thread calls `pthread_exit`, though, C++ doesn't guarantee that destructors are called for all automatic variables on the thread's stack. A clever way to recover this functionality is to invoke `pthread_exit` at the top level of the thread function by throwing a special exception.

The program in Listing 4.9 demonstrates this. Using this technique, a function indicates its intention to exit the thread by throwing a `ThreadExitException` instead of calling `pthread_exit` directly. Because the exception is caught in the top-level thread function, all local variables on the thread's stack will be destroyed properly as the exception percolates up.

Listing 4.9 **(*cxx-exit.cpp*) Implementing Safe Thread Exit with C++ Exceptions**

```
#include <pthread.h>

class ThreadExitException
{
public:
  /* Create an exception-signaling thread exit with RETURN_VALUE.  */
  ThreadExitException (void* return_value)
    : thread_return_value_ (return_value)
```

```
  {
  }

  /* Actually exit the thread, using the return value provided in the
     constructor.  */
  void* DoThreadExit ()
  {
    pthread_exit (thread_return_value_);
  }

private:
  /* The return value that will be used when exiting the thread.  */
  void* thread_return_value_;
};

void do_some_work ()
{
  while (1) {
    /* Do some useful things here...  */

    if (should_exit_thread_immediately ())
      throw ThreadExitException (/* thread's return value = */ NULL);
  }
}

void* thread_function (void*)
{
  try {
    do_some_work ();
  }
  catch (ThreadExitException ex) {
    /* Some function indicated that we should exit the thread.  */
    ex.DoThreadExit ();
  }
  return NULL;
}
```

4.4 Synchronization and Critical Sections

Programming with threads is very tricky because most threaded programs are concurrent programs. In particular, there's no way to know when the system will schedule one thread to run and when it will run another. One thread might run for a very long time, or the system might switch among threads very quickly. On a system with multiple processors, the system might even schedule multiple threads to run at literally the same time.

Debugging a threaded program is difficult because you cannot always easily reproduce the behavior that caused the problem. You might run the program once and have everything work fine; the next time you run it, it might crash. There's no way to make the system schedule the threads exactly the same way it did before.

The ultimate cause of most bugs involving threads is that the threads are accessing the same data. As mentioned previously, that's one of the powerful aspects of threads, but it can also be dangerous. If one thread is only partway through updating a data structure when another thread accesses the same data structure, chaos is likely to ensue. Often, buggy threaded programs contain a code that will work only if one thread gets scheduled more often—or sooner—than another thread. These bugs are called *race conditions*; the threads are racing one another to change the same data structure.

4.4.1 Race Conditions

Suppose that your program has a series of queued jobs that are processed by several concurrent threads. The queue of jobs is represented by a linked list of `struct job` objects.

After each thread finishes an operation, it checks the queue to see if an additional job is available. If `job_queue` is non-null, the thread removes the head of the linked list and sets `job_queue` to the next job on the list.

The thread function that processes jobs in the queue might look like Listing 4.10.

Listing 4.10 **(*job-queue1.c*) Thread Function to Process Jobs from the Queue**

```
#include <malloc.h>

struct job {
  /* Link field for linked list.  */
  struct job* next;

  /* Other fields describing work to be done... */
};

/* A linked list of pending jobs.  */
struct job* job_queue;

/* Process queued jobs until the queue is empty.  */

void* thread_function (void* arg)
{
  while (job_queue != NULL) {
    /* Get the next available job.  */
    struct job* next_job = job_queue;
    /* Remove this job from the list.  */
    job_queue = job_queue->next;
    /* Carry out the work.  */
    process_job (next_job);
    /* Clean up.  */
    free (next_job);
  }
  return NULL;
}
```

Now suppose that two threads happen to finish a job at about the same time, but only one job remains in the queue. The first thread checks whether `job_queue` is null; finding that it isn't, the thread enters the loop and stores the pointer to the job object in `next_job`. At this point, Linux happens to interrupt the first thread and schedules the second. The second thread also checks `job_queue` and finding it non-null, also assigns the same job pointer to `next_job`. By unfortunate coincidence, we now have two threads executing the same job.

To make matters worse, one thread will unlink the job object from the queue, leaving `job_queue` containing null. When the other thread evaluates `job_queue->next`, a segmentation fault will result.

This is an example of a race condition. Under "lucky" circumstances, this particular schedule of the two threads may never occur, and the race condition may never exhibit itself. Only under different circumstances, perhaps when running on a heavily loaded system (or on an important customer's new multiprocessor server!) may the bug exhibit itself.

To eliminate race conditions, you need a way to make operations *atomic*. An atomic operation is indivisible and uninterruptible; once the operation starts, it will not be paused or interrupted until it completes, and no other operation will take place meanwhile. In this particular example, you want to check `job_queue`; if it's not empty, remove the first job, all as a single atomic operation.

4.4.2 Mutexes

The solution to the job queue race condition problem is to let only one thread access the queue of jobs at a time. Once a thread starts looking at the queue, no other thread should be able to access it until the first thread has decided whether to process a job and, if so, has removed the job from the list.

Implementing this requires support from the operating system. GNU/Linux provides *mutexes*, short for *MUTual EXclusion locks*. A mutex is a special lock that only one thread may lock at a time. If a thread locks a mutex and then a second thread also tries to lock the same mutex, the second thread is *blocked*, or put on hold. Only when the first thread unlocks the mutex is the second thread *unblocked*—allowed to resume execution. GNU/Linux guarantees that race conditions do not occur among threads attempting to lock a mutex; only one thread will ever get the lock, and all other threads will be blocked.

Think of a mutex as the lock on a lavatory door. Whoever gets there first enters the lavatory and locks the door. If someone else attempts to enter the lavatory while it's occupied, that person will find the door locked and will be forced to wait outside until the occupant emerges.

To create a mutex, create a variable of type `pthread_mutex_t` and pass a pointer to it to `pthread_mutex_init`. The second argument to `pthread_mutex_init` is a pointer to a mutex attribute object, which specifies attributes of the mutex. As with pthread_create, if the attribute pointer is null, default attributes are assumed. The mutex variable should be initialized only once. This code fragment demonstrates the declaration and initialization of a mutex variable.

```
pthread_mutex_t mutex;
pthread_mutex_init (&mutex, NULL);
```

Another simpler way to create a mutex with default attributes is to initialize it with the special value PTHREAD_MUTEX_INITIALIZER. No additional call to pthread_mutex_init is necessary. This is particularly convenient for global variables (and, in C++, static data members). The previous code fragment could equivalently have been written like this:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

A thread may attempt to lock a mutex by calling pthread_mutex_lock on it. If the mutex was unlocked, it becomes locked and the function returns immediately. If the mutex was locked by another thread, pthread_mutex_lock blocks execution and returns only eventually when the mutex is unlocked by the other thread. More than one thread may be blocked on a locked mutex at one time. When the mutex is unlocked, only one of the blocked threads (chosen unpredictably) is unblocked and allowed to lock the mutex; the other threads stay blocked.

A call to pthread_mutex_unlock unlocks a mutex. This function should always be called from the same thread that locked the mutex.

Listing 4.11 shows another version of the job queue example. Now the queue is protected by a mutex. Before accessing the queue (either for read or write), each thread locks a mutex first. Only when the entire sequence of checking the queue and removing a job is complete is the mutex unlocked. This prevents the race condition previously described.

Listing 4.11 **(*job-queue2.c*) Job Queue Thread Function, Protected by a Mutex**

```
#include <malloc.h>
#include <pthread.h>

struct job {
  /* Link field for linked list.  */
  struct job* next;

  /* Other fields describing work to be done... */
};

/* A linked list of pending jobs.  */
struct job* job_queue;

/* A mutex protecting job_queue.  */
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
/* Process queued jobs until the queue is empty.  */

void* thread_function (void* arg)
{
  while (1) {
    struct job* next_job;

    /* Lock the mutex on the job queue.  */
    pthread_mutex_lock (&job_queue_mutex);
    /* Now it's safe to check if the queue is empty.  */
    if (job_queue == NULL)
      next_job = NULL;
    else {
      /* Get the next available job.  */
      next_job = job_queue;
      /* Remove this job from the list.  */
      job_queue = job_queue->next;
    }
    /* Unlock the mutex on the job queue because we're done with the
       queue for now.  */
    pthread_mutex_unlock (&job_queue_mutex);

    /* Was the queue empty?  If so, end the thread.  */
    if (next_job == NULL)
      break;

    /* Carry out the work.  */
    process_job (next_job);
    /* Clean up.  */
    free (next_job);
  }
  return NULL;
}
```

All accesses to `job_queue`, the shared data pointer, come between the call to `pthread_mutex_lock` and the call to `pthread_mutex_unlock`. A job object, stored in `next_job`, is accessed outside this region only after that object has been removed from the queue and is therefore inaccessible to other threads.

Note that if the queue is empty (that is, `job_queue` is null), we don't break out of the loop immediately because this would leave the mutex permanently locked and would prevent any other thread from accessing the job queue ever again. Instead, we remember this fact by setting `next_job` to null and breaking out only after unlocking the mutex.

Use of the mutex to lock `job_queue` is not automatic; it's up to you to add code to lock the mutex before accessing that variable and then to unlock it afterward. For example, a function to add a job to the job queue might look like this:

```
void enqueue_job (struct job* new_job)
{
  pthread_mutex_lock (&job_queue_mutex);
```

```
    new_job->next = job_queue;
    job_queue = new_job;
    pthread_mutex_unlock (&job_queue_mutex);
  }
```

4.4.3 Mutex Deadlocks

Mutexes provide a mechanism for allowing one thread to block the execution of another. This opens up the possibility of a new class of bugs, called *deadlocks*. A deadlock occurs when one or more threads are stuck waiting for something that never will occur.

A simple type of deadlock may occur when the same thread attempts to lock a mutex twice in a row. The behavior in this case depends on what kind of mutex is being used. Three kinds of mutexes exist:

- Locking a *fast mutex* (the default kind) will cause a deadlock to occur. An attempt to lock the mutex blocks until the mutex is unlocked. But because the thread that locked the mutex is blocked on the same mutex, the lock cannot ever be released.
- Locking a *recursive mutex* does not cause a deadlock. A recursive mutex may safely be locked many times by the same thread. The mutex remembers how many times `pthread_mutex_lock` was called on it by the thread that holds the lock; that thread must make the same number of calls to `pthread_mutex_unlock` before the mutex is actually unlocked and another thread is allowed to lock it.
- GNU/Linux will detect and flag a double lock on an *error-checking mutex* that would otherwise cause a deadlock. The second consecutive call to `pthread_mutex_lock` returns the failure code `EDEADLK`.

By default, a GNU/Linux mutex is of the fast kind. To create a mutex of one of the other two kinds, first create a mutex attribute object by declaring a `pthread_mutexattr_t` variable and calling `pthread_mutexattr_init` on a pointer to it. Then set the mutex kind by calling `pthread_mutexattr_setkind_np`; the first argument is a pointer to the mutex attribute object, and the second is `PTHREAD_MUTEX_RECURSIVE_NP` for a recursive mutex, or `PTHREAD_MUTEX_ERRORCHECK_NP` for an error-checking mutex. Pass a pointer to this attribute object to `pthread_mutex_init` to create a mutex of this kind, and then destroy the attribute object with `pthread_mutexattr_destroy`.

This code sequence illustrates creation of an error-checking mutex, for instance:

```
pthread_mutexattr_t attr;
pthread_mutex_t mutex;

pthread_mutexattr_init (&attr);
pthread_mutexattr_setkind_np (&attr, PTHREAD_MUTEX_ERRORCHECK_NP);
pthread_mutex_init (&mutex, &attr);
pthread_mutexattr_destroy (&attr);
```

As suggested by the "np" suffix, the recursive and error-checking mutex kinds are specific to GNU/Linux and are not portable. Therefore, it is generally not advised to use them in programs. (Error-checking mutexes can be useful when debugging, though.)

4.4.4 Nonblocking Mutex Tests

Occasionally, it is useful to test whether a mutex is locked without actually blocking on it. For instance, a thread may need to lock a mutex but may have other work to do instead of blocking if the mutex is already locked. Because `pthread_mutex_lock` will not return until the mutex becomes unlocked, some other function is necessary.

GNU/Linux provides `pthread_mutex_trylock` for this purpose. If you call `pthread_mutex_trylock` on an unlocked mutex, you will lock the mutex as if you had called `pthread_mutex_lock`, and `pthread_mutex_trylock` will return zero. However, if the mutex is already locked by another thread, `pthread_mutex_trylock` will not block. Instead, it will return immediately with the error code `EBUSY`. The mutex lock held by the other thread is not affected. You may try again later to lock the mutex.

4.4.5 Semaphores for Threads

In the preceding example, in which several threads process jobs from a queue, the main thread function of the threads carries out the next job until no jobs are left and then exits the thread. This scheme works if all the jobs are queued in advance or if new jobs are queued at least as quickly as the threads process them. However, if the threads work too quickly, the queue of jobs will empty and the threads will exit. If new jobs are later enqueued, no threads may remain to process them. What we might like instead is a mechanism for blocking the threads when the queue empties until new jobs become available.

A *semaphore* provides a convenient method for doing this. A semaphore is a counter that can be used to synchronize multiple threads. As with a mutex, GNU/Linux guarantees that checking or modifying the value of a semaphore can be done safely, without creating a race condition.

Each semaphore has a counter value, which is a non-negative integer. A semaphore supports two basic operations:

- A *wait* operation decrements the value of the semaphore by 1. If the value is already zero, the operation blocks until the value of the semaphore becomes positive (due to the action of some other thread). When the semaphore's value becomes positive, it is decremented by 1 and the wait operation returns.
- A *post* operation increments the value of the semaphore by 1. If the semaphore was previously zero and other threads are blocked in a wait operation on that semaphore, one of those threads is unblocked and its wait operation completes (which brings the semaphore's value back to zero).

Note that GNU/Linux provides two slightly different semaphore implementations. The one we describe here is the POSIX standard semaphore implementation. Use these semaphores when communicating among threads The other implementation, used for communication among processes, is described in Section 5.2, "Process Semaphores." If you use semaphores, include `<semaphore.h>`.

A semaphore is represented by a `sem_t` variable. Before using it, you must initialize it using the `sem_init` function, passing a pointer to the `sem_t` variable. The second parameter should be zero,[2] and the third parameter is the semaphore's initial value. If you no longer need a semaphore, it's good to deallocate it with `sem_destroy`.

To wait on a semaphore, use `sem_wait`. To post to a semaphore, use `sem_post`. A nonblocking wait function, `sem_trywait`, is also provided. It's similar to `pthread_mutex_trylock`—if the wait would have blocked because the semaphore's value was zero, the function returns immediately, with error value `EAGAIN`, instead of blocking.

GNU/Linux also provides a function to retrieve the current value of a semaphore, `sem_getvalue`, which places the value in the `int` variable pointed to by its second argument. You should not use the semaphore value you get from this function to make a decision whether to post to or wait on the semaphore, though. To do this could lead to a race condition: Another thread could change the semaphore's value between the call to `sem_getvalue` and the call to another semaphore function. Use the atomic post and wait functions instead.

Returning to our job queue example, we can use a semaphore to count the number of jobs waiting in the queue. Listing 4.12 controls the queue with a semaphore. The function `enqueue_job` adds a new job to the queue.

Listing 4.12 **(*job-queue3.c*) Job Queue Controlled by a Semaphore**

```
#include <malloc.h>
#include <pthread.h>
#include <semaphore.h>

struct job {
  /* Link field for linked list.  */
  struct job* next;

  /* Other fields describing work to be done... */
};

/* A linked list of pending jobs.  */
struct job* job_queue;

/* A mutex protecting job_queue.  */
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
```

2. A nonzero value would indicate a semaphore that can be shared across processes, which is not supported by GNU/Linux for this type of semaphore.

```
/* A semaphore counting the number of jobs in the queue.  */
sem_t job_queue_count;

/* Perform one-time initialization of the job queue.  */

void initialize_job_queue ()
{
  /* The queue is initially empty.  */
  job_queue = NULL;
  /* Initialize the semaphore which counts jobs in the queue.  Its
     initial value should be zero.  */
  sem_init (&job_queue_count, 0, 0);
}

/* Process queued jobs until the queue is empty.  */

void* thread_function (void* arg)
{
  while (1) {
    struct job* next_job;

    /* Wait on the job queue semaphore.  If its value is positive,
       indicating that the queue is not empty, decrement the count by
       1.  If the queue is empty, block until a new job is enqueued.  */
    sem_wait (&job_queue_count);

    /* Lock the mutex on the job queue.  */
    pthread_mutex_lock (&job_queue_mutex);
    /* Because of the semaphore, we know the queue is not empty.  Get
       the next available job.  */
    next_job = job_queue;
    /* Remove this job from the list.  */
    job_queue = job_queue->next;
    /* Unlock the mutex on the job queue because we're done with the
       queue for now.  */
    pthread_mutex_unlock (&job_queue_mutex);

    /* Carry out the work.  */
    process_job (next_job);
    /* Clean up.  */
    free (next_job);
  }
  return NULL;
}

/* Add a new job to the front of the job queue.  */

void enqueue_job (/* Pass job-specific data here...  */)
{
  struct job* new_job;
```

continues

Listing 4.12 **Continued**

```
  /* Allocate a new job object.  */
  new_job = (struct job*) malloc (sizeof (struct job));
  /* Set the other fields of the job struct here...  */

  /* Lock the mutex on the job queue before accessing it.  */
  pthread_mutex_lock (&job_queue_mutex);
  /* Place the new job at the head of the queue.  */
  new_job->next = job_queue;
  job_queue = new_job;

  /* Post to the semaphore to indicate that another job is available.  If
     threads are blocked, waiting on the semaphore, one will become
     unblocked so it can process the job.  */
  sem_post (&job_queue_count);

  /* Unlock the job queue mutex.  */
  pthread_mutex_unlock (&job_queue_mutex);
}
```

Before taking a job from the front of the queue, each thread will first wait on the semaphore. If the semaphore's value is zero, indicating that the queue is empty, the thread will simply block until the semaphore's value becomes positive, indicating that a job has been added to the queue.

The `enqueue_job` function adds a job to the queue. Just like `thread_function`, it needs to lock the queue mutex before modifying the queue. After adding a job to the queue, it posts to the semaphore, indicating that a new job is available. In the version shown in Listing 4.12, the threads that process the jobs never exit; if no jobs are available for a while, all the threads simply block in `sem_wait`.

4.4.6 Condition Variables

We've shown how to use a mutex to protect a variable against simultaneous access by two threads and how to use semaphores to implement a shared counter. A *condition variable* is a third synchronization device that GNU/Linux provides; with it, you can implement more complex conditions under which threads execute.

Suppose that you write a thread function that executes a loop infinitely, performing some work on each iteration. The thread loop, however, needs to be controlled by a flag: The loop runs only when the flag is set; when the flag is not set, the loop pauses.

Listing 4.13 shows how you might implement this by spinning in a loop. During each iteration of the loop, the thread function checks that the flag is set. Because the flag is accessed by multiple threads, it is protected by a mutex. This implementation may be correct, but it is not efficient. The thread function will spend lots of CPU whenever the flag is not set, checking and rechecking the flag, each time locking and unlocking the mutex. What you really want is a way to put the thread to sleep when the flag is not set, until some circumstance changes that might cause the flag to become set.

Listing 4.13 **(*spin-condvar.c*) A Simple Condition Variable Implementation**

```
#include <pthread.h>

int thread_flag;
pthread_mutex_t thread_flag_mutex;

void initialize_flag ()
{
  pthread_mutex_init (&thread_flag_mutex, NULL);
  thread_flag = 0;
}

/* Calls do_work repeatedly while the thread flag is set; otherwise
   spins.  */

void* thread_function (void* thread_arg)
{
  while (1) {
    int flag_is_set;

    /* Protect the flag with a mutex lock.  */
    pthread_mutex_lock (&thread_flag_mutex);
    flag_is_set = thread_flag;
    pthread_mutex_unlock (&thread_flag_mutex);

    if (flag_is_set)
      do_work ();
    /* Else don't do anything.  Just loop again.  */
  }
  return NULL;
}

/* Sets the value of the thread flag to FLAG_VALUE.  */

void set_thread_flag (int flag_value)
{
  /* Protect the flag with a mutex lock.  */
  pthread_mutex_lock (&thread_flag_mutex);
  thread_flag = flag_value;
  pthread_mutex_unlock (&thread_flag_mutex);
}
```

A condition variable enables you to implement a condition under which a thread executes and, inversely, the condition under which the thread is blocked. As long as every thread that potentially changes the sense of the condition uses the condition variable properly, Linux guarantees that threads blocked on the condition will be unblocked when the condition changes.

As with a semaphore, a thread may *wait* on a condition variable. If thread A waits on a condition variable, it is blocked until some other thread, thread B, signals the same condition variable. Unlike a semaphore, a condition variable has no counter or memory; thread A must wait on the condition variable *before* thread B signals it. If thread B signals the condition variable before thread A waits on it, the signal is lost, and thread A blocks until some other thread signals the condition variable again.

This is how you would use a condition variable to make the previous sample more efficient:

- The loop in `thread_function` checks the flag. If the flag is not set, the thread waits on the condition variable.
- The `set_thread_flag` function signals the condition variable after changing the flag value. That way, if `thread_function` is blocked on the condition variable, it will be unblocked and will check the condition again.

There's one problem with this: There's a race condition between checking the flag value and signaling or waiting on the condition variable. Suppose that `thread_function` checked the flag and found that it was not set. At that moment, the Linux scheduler paused that thread and resumed the main one. By some coincidence, the main thread is in `set_thread_flag`. It sets the flag and then signals the condition variable. Because no thread is waiting on the condition variable at the time (remember that `thread_function` was paused before it could wait on the condition variable), the signal is lost. Now, when Linux reschedules the other thread, it starts waiting on the condition variable and may end up blocked forever.

To solve this problem, we need a way to lock the flag and the condition variable together with a single mutex. Fortunately, GNU/Linux provides exactly this mechanism. Each condition variable must be used in conjunction with a mutex, to prevent this sort of race condition. Using this scheme, the thread function follows these steps:

1. The loop in `thread_function` locks the mutex and reads the flag value.
2. If the flag is set, it unlocks the mutex and executes the work function.
3. If the flag is not set, it atomically unlocks the mutex and waits on the condition variable.

The critical feature here is in step 3, in which GNU/Linux allows you to unlock the mutex and wait on the condition variable atomically, without the possibility of another thread intervening. This eliminates the possibility that another thread may change the flag value and signal the condition variable in between `thread_function`'s test of the flag value and wait on the condition variable.

A condition variable is represented by an instance of pthread_cond_t. Remember that each condition variable should be accompanied by a mutex. These are the functions that manipulate condition variables:

- pthread_cond_init initializes a condition variable. The first argument is a pointer to a pthread_cond_t instance. The second argument, a pointer to a condition variable attribute object, is ignored under GNU/Linux.

  The mutex must be initialized separately, as described in Section 4.4.2, "Mutexes."
- pthread_cond_signal signals a condition variable. A single thread that is blocked on the condition variable will be unblocked. If no other thread is blocked on the condition variable, the signal is ignored. The argument is a pointer to the pthread_cond_t instance.

  A similar call, pthread_cond_broadcast, unblocks *all* threads that are blocked on the condition variable, instead of just one.
- pthread_cond_wait blocks the calling thread until the condition variable is signaled. The argument is a pointer to the pthread_cond_t instance. The second argument is a pointer to the pthread_mutex_t mutex instance.

  When pthread_cond_wait is called, the mutex must already be locked by the calling thread. That function atomically unlocks the mutex and blocks on the condition variable. When the condition variable is signaled and the calling thread unblocks, pthread_cond_wait automatically reacquires a lock on the mutex.

Whenever your program performs an action that may change the sense of the condition you're protecting with the condition variable, it should perform these steps. (In our example, the condition is the state of the thread flag, so these steps must be taken whenever the flag is changed.)

1. Lock the mutex accompanying the condition variable.
2. Take the action that may change the sense of the condition (in our example, set the flag).
3. Signal or broadcast the condition variable, depending on the desired behavior.
4. Unlock the mutex accompanying the condition variable.

Listing 4.14 shows the previous example again, now using a condition variable to protect the thread flag. Note that in thread_function, a lock on the mutex is held before checking the value of thread_flag. That lock is automatically released by pthread_cond_wait before blocking and is automatically reacquired afterward. Also note that set_thread_flag locks the mutex before setting the value of thread_flag and signaling the mutex.

Listing 4.14 (*condvar.c*) **Control a Thread Using a Condition Variable**

```
#include <pthread.h>

int thread_flag;
pthread_cond_t thread_flag_cv;
pthread_mutex_t thread_flag_mutex;

void initialize_flag ()
{
  /* Initialize the mutex and condition variable.  */
  pthread_mutex_init (&thread_flag_mutex, NULL);
  pthread_cond_init (&thread_flag_cv, NULL);
  /* Initialize the flag value.  */
  thread_flag = 0;
}

/* Calls do_work repeatedly while the thread flag is set; blocks if
   the flag is clear.  */

void* thread_function (void* thread_arg)
{
  /* Loop infinitely.  */
  while (1) {
    /* Lock the mutex before accessing the flag value.  */
    pthread_mutex_lock (&thread_flag_mutex);
    while (!thread_flag)
      /* The flag is clear.  Wait for a signal on the condition
         variable, indicating that the flag value has changed.  When the
         signal arrives and this thread unblocks, loop and check the
         flag again.  */
      pthread_cond_wait (&thread_flag_cv, &thread_flag_mutex);
    /* When we've gotten here, we know the flag must be set.  Unlock
       the mutex.  */
    pthread_mutex_unlock (&thread_flag_mutex);
    /* Do some work.  */
    do_work ();
  }
  return NULL;
}

/* Sets the value of the thread flag to FLAG_VALUE.  */

void set_thread_flag (int flag_value)
{
  /* Lock the mutex before accessing the flag value.  */
  pthread_mutex_lock (&thread_flag_mutex);
  /* Set the flag value, and then signal in case thread_function is
     blocked, waiting for the flag to become set.  However,
     thread_function can't actually check the flag until the mutex is
     unlocked.  */
```

```
  thread_flag = flag_value;
  pthread_cond_signal (&thread_flag_cv);
  /* Unlock the mutex.  */
  pthread_mutex_unlock (&thread_flag_mutex);
}
```

The condition protected by a condition variable can be arbitrarily complex. However, before performing any operation that may change the sense of the condition, a mutex lock should be required, and the condition variable should be signaled afterward.

A condition variable may also be used without a condition, simply as a mechanism for blocking a thread until another thread "wakes it up." A semaphore may also be used for that purpose. The principal difference is that a semaphore "remembers" the wake-up call even if no thread was blocked on it at the time, while a condition variable discards the wake-up call unless some thread is actually blocked on it at the time. Also, a semaphore delivers only a single wake-up per post; with `pthread_cond_broadcast`, an arbitrary and unknown number of blocked threads may be awoken at the same time.

4.4.7 Deadlocks with Two or More Threads

Deadlocks can occur when two (or more) threads are each blocked, waiting for a condition to occur that only the other one can cause. For instance, if thread A is blocked on a condition variable waiting for thread B to signal it, and thread B is blocked on a condition variable waiting for thread A to signal it, a deadlock has occurred because neither thread will ever signal the other. You should take care to avoid the possibility of such situations because they are quite difficult to detect.

One common error that can cause a deadlock involves a problem in which more than one thread is trying to lock the same set of objects. For example, consider a program in which two different threads, running two different thread functions, need to lock the same two mutexes. Suppose that thread A locks mutex 1 and then mutex 2, and thread B happens to lock mutex 2 before mutex 1. In a sufficiently unfortunate scheduling scenario, Linux may schedule thread A long enough to lock mutex 1, and then schedule thread B, which promptly locks mutex 2. Now neither thread can progress because each is blocked on a mutex that the other thread holds locked.

This is an example of a more general deadlock problem, which can involve not only synchronization objects such as mutexes, but also other resources, such as locks on files or devices. The problem occurs when multiple threads try to lock the same set of resources in different orders. The solution is to make sure that all threads that lock more than one resource lock them in the same order.

4.5 GNU/Linux Thread Implementation

The implementation of POSIX threads on GNU/Linux differs from the thread implementation on many other UNIX-like systems in an important way: on GNU/Linux, threads are implemented as processes. Whenever you call `pthread_create` to create a new thread, Linux creates a new process that runs that thread. However, this process is not the same as a process you would create with `fork`; in particular, it shares the same address space and resources as the original process rather than receiving copies.

The program `thread-pid` shown in Listing 4.15 demonstrates this. The program creates a thread; both the original thread and the new one call the `getpid` function and print their respective process IDs and then spin infinitely.

Listing 4.15 **(*thread-pid*) Print Process IDs for Threads**

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void* thread_function (void* arg)
{
  fprintf (stderr, "child thread pid is %d\n", (int) getpid ());
  /* Spin forever.  */
  while (1);
  return NULL;
}

int main ()
{
  pthread_t thread;
  fprintf (stderr, "main thread pid is %d\n", (int) getpid ());
  pthread_create (&thread, NULL, &thread_function, NULL);
  /* Spin forever.  */
  while (1);
  return 0;
}
```

Run the program in the background, and then invoke `ps x` to display your running processes. Don't forget to kill the `thread-pid` program afterward—it consumes lots of CPU doing nothing. Here's what the output might look like:

```
% cc thread-pid.c -o thread-pid -lpthread
% ./thread-pid &
[1] 14608
main thread pid is 14608
child thread pid is 14610
% ps x
  PID TTY      STAT   TIME COMMAND
14042 pts/9    S      0:00 bash
14608 pts/9    R      0:01 ./thread-pid
```

```
14609 pts/9    S      0:00 ./thread-pid
14610 pts/9    R      0:01 ./thread-pid
14611 pts/9    R      0:00 ps x
% kill 14608
[1]+  Terminated              ./thread-pid
```

Job Control Notification in the Shell

The lines starting with `[1]` are from the shell. When you run a program in the background, the shell assigns a job number to it—in this case, 1—and prints out the program's pid. If a background job terminates, the shell reports that fact the next time you invoke a command.

Notice that there are three processes running the `thread-pid` program. The first of these, with pid 14608, is the main thread in the program; the third, with pid 14610, is the thread we created to execute `thread_function`.

How about the second thread, with pid 14609? This is the "manager thread," which is part of the internal implementation of GNU/Linux threads. The manager thread is created the first time a program calls `pthread_create` to create a new thread.

4.5.1 Signal Handling

Suppose that a multithreaded program receives a signal. In which thread is the signal handler invoked? The behavior of the interaction between signals and threads varies from one UNIX-like system to another. In GNU/Linux, the behavior is dictated by the fact that threads are implemented as processes.

Because each thread is a separate process, and because a signal is delivered to a particular process, there is no ambiguity about which thread receives the signal. Typically, signals sent from outside the program are sent to the process corresponding to the main thread of the program. For instance, if a program forks and the child process execs a multithreaded program, the parent process will hold the process id of the main thread of the child process's program and will use that process id to send signals to its child. This is generally a good convention to follow yourself when sending signals to a multithreaded program.

Note that this aspect of GNU/Linux's implementation of pthreads is at variance with the POSIX thread standard. Do not rely on this behavior in programs that are meant to be portable.

Within a multithreaded program, it is possible for one thread to send a signal specifically to another thread. Use the `pthread_kill` function to do this. Its first parameter is a thread ID, and its second parameter is a signal number.

4.5.2 The *clone* System Call

Although GNU/Linux threads created in the same program are implemented as separate processes, they share their virtual memory space and other resources. A child process created with `fork`, however, gets copies of these items. How is the former type of process created?

The Linux `clone` system call is a generalized form of `fork` and `pthread_create` that allows the caller to specify which resources are shared between the calling process and the newly created process. Also, `clone` requires you to specify the memory region for the execution stack that the new process will use. Although we mention `clone` here to satisfy the reader's curiosity, that system call should not ordinarily be used in programs. Use `fork` to create new processes or `pthread_create` to create threads.

4.6 Processes Vs. Threads

For some programs that benefit from concurrency, the decision whether to use processes or threads can be difficult. Here are some guidelines to help you decide which concurrency model best suits your program:

- All threads in a program must run the same executable. A child process, on the other hand, may run a different executable by calling an `exec` function.
- An errant thread can harm other threads in the same process because threads share the same virtual memory space and other resources. For instance, a wild memory write through an uninitialized pointer in one thread can corrupt memory visible to another thread.

  An errant process, on the other hand, cannot do so because each process has a copy of the program's memory space.
- Copying memory for a new process adds an additional performance overhead relative to creating a new thread. However, the copy is performed only when the memory is changed, so the penalty is minimal if the child process only reads memory.
- Threads should be used for programs that need fine-grained parallelism. For example, if a problem can be broken into multiple, nearly identical tasks, threads may be a good choice. Processes should be used for programs that need coarser parallelism.
- Sharing data among threads is trivial because threads share the same memory. (However, great care must be taken to avoid race conditions, as described previously.) Sharing data among processes requires the use of IPC mechanisms, as described in Chapter 5. This can be more cumbersome but makes multiple processes less likely to suffer from concurrency bugs.

5 Interprocess Communication

Chapter 3, "Processes," discussed the creation of processes and showed how one process can obtain the exit status of a child process. That's the simplest form of communication between two processes, but it's by no means the most powerful. The mechanisms of Chapter 3 don't provide any way for the parent to communicate with the child except via command-line arguments and environment variables, nor any way for the child to communicate with the parent except via the child's exit status. None of these mechanisms provides any means for communicating with the child process while it is actually running, nor do these mechanisms allow communication with a process outside the parent-child relationship.

This chapter describes means for interprocess communication that circumvent these limitations. We will present various ways for communicating between parents and children, between "unrelated" processes, and even between processes on different machines.

Interprocess communication (IPC) is the transfer of data among processes. For example, a Web browser may request a Web page from a Web server, which then sends HTML data. This transfer of data usually uses sockets in a telephone-like connection. In another example, you may want to print the filenames in a directory using a command such as `ls | lpr`. The shell creates an `ls` process and a separate `lpr` process, connecting the two with a *pipe*, represented by the "|" symbol. A pipe permits one-way communication between two related processes. The `ls` process writes data into the pipe, and the `lpr` process reads data from the pipe.

In this chapter, we discuss five types of interprocess communication:

- Shared memory permits processes to communicate by simply reading and writing to a specified memory location.
- Mapped memory is similar to shared memory, except that it is associated with a file in the filesystem.
- Pipes permit sequential communication from one process to a related process.
- FIFOs are similar to pipes, except that unrelated processes can communicate because the pipe is given a name in the filesystem.
- Sockets support communication between unrelated processes even on different computers.

These types of IPC differ by the following criteria:

- Whether they restrict communication to related processes (processes with a common ancestor), to unrelated processes sharing the same filesystem, or to any computer connected to a network
- Whether a communicating process is limited to only write data or only read data
- The number of processes permitted to communicate
- Whether the communicating processes are synchronized by the IPC—for example, a reading process halts until data is available to read

In this chapter, we omit discussion of IPC permitting communication only a limited number of times, such as communicating via a child's exit value.

5.1 Shared Memory

One of the simplest interprocess communication methods is using shared memory. Shared memory allows two or more processes to access the same memory as if they all called `malloc` and were returned pointers to the same actual memory. When one process changes the memory, all the other processes see the modification.

5.1.1 Fast Local Communication

Shared memory is the fastest form of interprocess communication because all processes share the same piece of memory. Access to this shared memory is as fast as accessing a process's nonshared memory, and it does not require a system call or entry to the kernel. It also avoids copying data unnecessarily.

Because the kernel does not synchronize accesses to shared memory, you must provide your own synchronization. For example, a process should not read from the memory until after data is written there, and two processes must not write to the same memory location at the same time. A common strategy to avoid these race conditions is to use semaphores, which are discussed in the next section. Our illustrative programs, though, show just a single process accessing the memory, to focus on the shared memory mechanism and to avoid cluttering the sample code with synchronization logic.

5.1.2 The Memory Model

To use a shared memory segment, one process must allocate the segment. Then each process desiring to access the segment must attach the segment. After finishing its use of the segment, each process detaches the segment. At some point, one process must deallocate the segment.

Understanding the Linux memory model helps explain the allocation and attachment process. Under Linux, each process's virtual memory is split into pages. Each process maintains a mapping from its memory addresses to these virtual memory pages, which contain the actual data. Even though each process has its own addresses, multiple processes' mappings can point to the same page, permitting sharing of memory. Memory pages are discussed further in Section 8.8, "The `mlock` Family: Locking Physical Memory," of Chapter 8, "Linux System Calls."

Allocating a new shared memory segment causes virtual memory pages to be created. Because all processes desire to access the same shared segment, only one process should allocate a new shared segment. Allocating an existing segment does not create new pages, but it does return an identifier for the existing pages. To permit a process to use the shared memory segment, a process attaches it, which adds entries mapping from its virtual memory to the segment's shared pages. When finished with the segment, these mapping entries are removed. When no more processes want to access these shared memory segments, exactly one process must deallocate the virtual memory pages.

All shared memory segments are allocated as integral multiples of the system's *page size*, which is the number of bytes in a page of memory. On Linux systems, the page size is 4KB, but you should obtain this value by calling the `getpagesize` function.

5.1.3 Allocation

A process allocates a shared memory segment using `shmget` ("SHared Memory GET"). Its first parameter is an integer key that specifies which segment to create. Unrelated processes can access the same shared segment by specifying the same key value. Unfortunately, other processes may have also chosen the same fixed key, which could lead to conflict. Using the special constant `IPC_PRIVATE` as the key value guarantees that a brand new memory segment is created.

Its second parameter specifies the number of bytes in the segment. Because segments are allocated using pages, the number of actually allocated bytes is rounded up to an integral multiple of the page size.

The third parameter is the bitwise or of flag values that specify options to `shmget`. The flag values include these:

- `IPC_CREAT`—This flag indicates that a new segment should be created. This permits creating a new segment while specifying a key value.
- `IPC_EXCL`—This flag, which is always used with `IPC_CREAT`, causes `shmget` to fail if a segment key is specified that already exists. Therefore, it arranges for the calling process to have an "exclusive" segment. If this flag is not given and the key of an existing segment is used, `shmget` returns the existing segment instead of creating a new one.
- Mode flags—This value is made of 9 bits indicating permissions granted to owner, group, and world to control access to the segment. Execution bits are ignored. An easy way to specify permissions is to use the constants defined in `<sys/stat.h>` and documented in the section 2 `stat` man page.[1] For example, `S_IRUSR` and `S_IWUSR` specify read and write permissions for the owner of the shared memory segment, and `S_IROTH` and `S_IWOTH` specify read and write permissions for others.

For example, this invocation of `shmget` creates a new shared memory segment (or access to an existing one, if `shm_key` is already used) that's readable and writeable to the owner but not other users.

```
int segment_id = shmget (shm_key, getpagesize (),
                         IPC_CREAT | S_IRUSR |  S_IWUSER);
```

If the call succeeds, `shmget` returns a segment identifier. If the shared memory segment already exists, the access permissions are verified and a check is made to ensure that the segment is not marked for destruction.

5.1.4 Attachment and Detachment

To make the shared memory segment available, a process must use `shmat`, "SHared Memory ATtach." Pass it the shared memory segment identifier `SHMID` returned by `shmget`. The second argument is a pointer that specifies where in your process's address space you want to map the shared memory; if you specify NULL, Linux will choose an available address. The third argument is a flag, which can include the following:

- `SHM_RND` indicates that the address specified for the second parameter should be rounded down to a multiple of the page size. If you don't specify this flag, you must page-align the second argument to `shmat` yourself.
- `SHM_RDONLY` indicates that the segment will be only read, not written.

1. These permission bits are the same as those used for files. They are described in Section 10.3, "File System Permissions."

If the call succeeds, it returns the address of the attached shared segment. Children created by calls to `fork` inherit attached shared segments; they can detach the shared memory segments, if desired.

When you're finished with a shared memory segment, the segment should be detached using `shmdt` ("SHared Memory DeTach"). Pass it the address returned by `shmat`. If the segment has been deallocated and this was the last process using it, it is removed. Calls to `exit` and any of the `exec` family automatically detach segments.

5.1.5 Controlling and Deallocating Shared Memory

The `shmctl` ("SHared Memory ConTroL") call returns information about a shared memory segment and can modify it. The first parameter is a shared memory segment identifier.

To obtain information about a shared memory segment, pass `IPC_STAT` as the second argument and a pointer to a `struct shmid_ds`.

To remove a segment, pass `IPC_RMID` as the second argument, and pass NULL as the third argument. The segment is removed when the last process that has attached it finally detaches it.

Each shared memory segment should be explicitly deallocated using `shmctl` when you're finished with it, to avoid violating the systemwide limit on the total number of shared memory segments. Invoking `exit` and `exec` detaches memory segments but does not deallocate them.

See the `shmctl` man page for a description of other operations you can perform on shared memory segments.

5.1.6 An Example Program

The program in Listing 5.1 illustrates the use of shared memory.

Listing 5.1 **(*shm.c*) Exercise Shared Memory**

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main ()
{
  int segment_id;
  char* shared_memory;
  struct shmid_ds shmbuffer;
  int segment_size;
  const int shared_segment_size = 0x6400;

  /* Allocate a shared memory segment.  */
  segment_id = shmget (IPC_PRIVATE, shared_segment_size,
                       IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
```

continues

Listing 5.1 **Continued**

```
  /* Attach the shared memory segment.  */
  shared_memory = (char*) shmat (segment_id, 0, 0);
  printf ("shared memory attached at address %p\n", shared_memory);
  /* Determine the segment's size.  */
  shmctl (segment_id, IPC_STAT, &shmbuffer);
  segment_size = shmbuffer.shm_segsz;
  printf ("segment size: %d\n", segment_size);
  /* Write a string to the shared memory segment.  */
  sprintf (shared_memory, "Hello, world.");
  /* Detach the shared memory segment.  */
  shmdt (shared_memory);

  /* Reattach the shared memory segment, at a different address.  */
  shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);
  printf ("shared memory reattached at address %p\n", shared_memory);
  /* Print out the string from shared memory.  */
  printf ("%s\n", shared_memory);
  /* Detach the shared memory segment.  */
  shmdt (shared_memory);

  /* Deallocate the shared memory segment.  */
  shmctl (segment_id, IPC_RMID, 0);

  return 0;
}
```

5.1.7 Debugging

The `ipcs` command provides information on interprocess communication facilities, including shared segments. Use the `-m` flag to obtain information about shared memory. For example, this code illustrates that one shared memory segment, numbered 1627649, is in use:

```
% ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 1627649    user       640        25600      0
```

If this memory segment was erroneously left behind by a program, you can use the `ipcrm` command to remove it.

```
% ipcrm shm 1627649
```

5.1.8 Pros and Cons

Shared memory segments permit fast bidirectional communication among any number of processes. Each user can both read and write, but a program must establish and follow some protocol for preventing race conditions such as overwriting information before it is read. Unfortunately, Linux does not strictly guarantee exclusive access even if you create a new shared segment with `IPC_PRIVATE`.

Also, for multiple processes to use a shared segment, they must make arrangements to use the same key.

5.2 Processes Semaphores

As noted in the previous section, processes must coordinate access to shared memory. As we discussed in Section 4.4.5, "Semaphores for Threads," in Chapter 4, "Threads," semaphores are counters that permit synchronizing multiple threads. Linux provides a distinct alternate implementation of semaphores that can be used for synchronizing processes (called process semaphores or sometimes System V semaphores). Process semaphores are allocated, used, and deallocated like shared memory segments. Although a single semaphore is sufficient for almost all uses, process semaphores come in sets. Throughout this section, we present system calls for process semaphores, showing how to implement single binary semaphores using them.

5.2.1 Allocation and Deallocation

The calls `semget` and `semctl` allocate and deallocate semaphores, which is analogous to `shmget` and `shmctl` for shared memory. Invoke `semget` with a key specifying a semaphore set, the number of semaphores in the set, and permission flags as for `shmget`; the return value is a semaphore set identifier. You can obtain the identifier of an existing semaphore set by specifying the right key value; in this case, the number of semaphores can be zero.

Semaphores continue to exist even after all processes using them have terminated. The last process to use a semaphore set must explicitly remove it to ensure that the operating system does not run out of semaphores. To do so, invoke `semctl` with the semaphore identifier, the number of semaphores in the set, `IPC_RMID` as the third argument, and any `union semun` value as the fourth argument (which is ignored). The effective user ID of the calling process must match that of the semaphore's allocator (or the caller must be `root`). Unlike shared memory segments, removing a semaphore set causes Linux to deallocate immediately.

Listing 5.2 presents functions to allocate and deallocate a binary semaphore.

Listing 5.2 (*sem_all_deall.c*) **Allocating and Deallocating a Binary Semaphore**

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>

/* We must define union semun ourselves.  */

union semun {
  int val;
  struct semid_ds *buf;
  unsigned short int *array;
  struct seminfo *__buf;
};

/* Obtain a binary semaphore's ID, allocating if necessary.  */

int binary_semaphore_allocation (key_t key, int sem_flags)
{
  return semget (key, 1, sem_flags);
}

/* Deallocate a binary semaphore.  All users must have finished their
   use.  Returns -1 on failure.  */

int binary_semaphore_deallocate (int semid)
{
  union semun ignored_argument;
  return semctl (semid, 1, IPC_RMID, ignored_argument);
}
```

5.2.2 Initializing Semaphores

Allocating and initializing semaphores are two separate operations. To initialize a semaphore, use `semctl` with zero as the second argument and `SETALL` as the third argument. For the fourth argument, you must create a `union semun` object and point its `array` field at an array of unsigned short values. Each value is used to initialize one semaphore in the set.

Listing 5.3 presents a function that initializes a binary semaphore.

Listing 5.3 (*sem_init.c*) **Initializing a Binary Semaphore**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
/* We must define union semun ourselves.  */

union semun {
  int val;
  struct semid_ds *buf;
  unsigned short int *array;
  struct seminfo *__buf;
};

/* Initialize a binary semaphore with a value of 1.  */

int binary_semaphore_initialize (int semid)
{
  union semun argument;
  unsigned short values[1];
  values[0] = 1;
  argument.array = values;
  return semctl (semid, 0, SETALL, argument);
}
```

5.2.3 Wait and Post Operations

Each semaphore has a non-negative value and supports wait and post operations. The `semop` system call implements both operations. Its first parameter specifies a semaphore set identifier. Its second parameter is an array of `struct sembuf` elements, which specify the operations you want to perform. The third parameter is the length of this array.

The fields of `struct sembuf` are listed here:

- `sem_num` is the semaphore number in the semaphore set on which the operation is performed.
- `sem_op` is an integer that specifies the semaphore operation.

  If `sem_op` is a positive number, that number is added to the semaphore value immediately.

  If `sem_op` is a negative number, the absolute value of that number is subtracted from the semaphore value. If this would make the semaphore value negative, the call blocks until the semaphore value becomes as large as the absolute value of `sem_op` (because some other process increments it).

  If `sem_op` is zero, the operation blocks until the semaphore value becomes zero.
- `sem_flg` is a flag value. Specify `IPC_NOWAIT` to prevent the operation from blocking; if the operation would have blocked, the call to `semop` fails instead. If you specify `SEM_UNDO`, Linux automatically undoes the operation on the semaphore when the process exits.

Listing 5.4 illustrates wait and post operations for a binary semaphore.

Listing 5.4 (*sem_pv.c*) **Wait and Post Operations for a Binary Semaphore**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

/* Wait on a binary semaphore.  Block until the semaphore value is positive, then
   decrement it by 1.  */

int binary_semaphore_wait (int semid)
{
  struct sembuf operations[1];
  /* Use the first (and only) semaphore.  */
  operations[0].sem_num = 0;
  /* Decrement by 1.  */
  operations[0].sem_op = -1;
  /* Permit undo'ing.  */
  operations[0].sem_flg = SEM_UNDO;

  return semop (semid, operations, 1);
}

/* Post to a binary semaphore: increment its value by 1.
   This returns immediately.  */

int binary_semaphore_post (int semid)
{
  struct sembuf operations[1];
  /* Use the first (and only) semaphore.  */
  operations[0].sem_num = 0;
  /* Increment by 1.  */
  operations[0].sem_op = 1;
  /* Permit undo'ing.  */
  operations[0].sem_flg = SEM_UNDO;

  return semop (semid, operations, 1);
}
```

Specifying the `SEM_UNDO` flag permits dealing with the problem of terminating a process while it has resources allocated through a semaphore. When a process terminates, either voluntarily or involuntarily, the semaphore's values are automatically adjusted to "undo" the process's effects on the semaphore. For example, if a process that has decremented a semaphore is killed, the semaphore's value is incremented.

5.2.4 Debugging Semaphores

Use the command `ipcs -s` to display information about existing semaphore sets. Use the `ipcrm sem` command to remove a semaphore set from the command line. For example, to remove the semaphore set with identifier 5790517, use this line:

```
% ipcrm sem 5790517
```

5.3 Mapped Memory

Mapped memory permits different processes to communicate via a shared file. Although you can think of mapped memory as using a shared memory segment with a name, you should be aware that there are technical differences. Mapped memory can be used for interprocess communication or as an easy way to access the contents of a file.

Mapped memory forms an association between a file and a process's memory. Linux splits the file into page-sized chunks and then copies them into virtual memory pages so that they can be made available in a process's address space. Thus, the process can read the file's contents with ordinary memory access. It can also modify the file's contents by writing to memory. This permits fast access to files.

You can think of mapped memory as allocating a buffer to hold a file's entire contents, and then reading the file into the buffer and (if the buffer is modified) writing the buffer back out to the file afterward. Linux handles the file reading and writing operations for you.

There are uses for memory-mapped files other than interprocess communication. Some of these are discussed in Section 5.3.5, "Other Uses for `mmap`."

5.3.1 Mapping an Ordinary File

To map an ordinary file to a process's memory, use the `mmap` ("Memory MAPped," pronounced "em-map") call. The first argument is the address at which you would like Linux to map the file into your process's address space; the value NULL allows Linux to choose an available start address. The second argument is the length of the map in bytes. The third argument specifies the protection on the mapped address range. The protection consists of a bitwise "or" of `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC`, corresponding to read, write, and execution permission, respectively. The fourth argument is a flag value that specifies additional options. The fifth argument is a file descriptor opened to the file to be mapped. The last argument is the offset from the beginning of the file from which to start the map. You can map all or part of the file into memory by choosing the starting offset and length appropriately.

The flag value is a bitwise "or" of these constraints:

- `MAP_FIXED`—If you specify this flag, Linux uses the address you request to map the file rather than treating it as a hint. This address must be page-aligned.
- `MAP_PRIVATE`—Writes to the memory range should not be written back to the attached file, but to a private copy of the file. No other process sees these writes. This mode may not be used with `MAP_SHARED`.

- MAP_SHARED—Writes are immediately reflected in the underlying file rather than buffering writes. Use this mode when using mapped memory for IPC. This mode may not be used with MAP_PRIVATE.

If the call succeeds, it returns a pointer to the beginning of the memory. On failure, it returns MAP_FAILED.

When you're finished with a memory mapping, release it by using munmap. Pass it the start address and length of the mapped memory region. Linux automatically unmaps mapped regions when a process terminates.

5.3.2 Example Programs

Let's look at two programs to illustrate using memory-mapped regions to read and write to files. The first program, Listing 5.5, generates a random number and writes it to a memory-mapped file. The second program, Listing 5.6, reads the number, prints it, and replaces it in the memory-mapped file with double the value. Both take a command-line argument of the file to map.

Listing 5.5 **(*mmap-write.c*) Write a Random Number to a Memory-Mapped File**

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

/* Return a uniformly random number in the range [low,high].  */

int random_range (unsigned const low, unsigned const high)
{
  unsigned const range = high - low + 1;
  return low + (int) (((double) range) * rand () / (RAND_MAX + 1.0));
}

int main (int argc, char* const argv[])
{
  int fd;
  void* file_memory;

  /* Seed the random number generator.  */
  srand (time (NULL));

  /* Prepare a file large enough to hold an unsigned integer.  */
  fd = open (argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
  lseek (fd, FILE_LENGTH+1, SEEK_SET);
```

```
  write (fd, "", 1);
  lseek (fd, 0, SEEK_SET);

  /* Create the memory mapping.  */
  file_memory = mmap (0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd, 0);
  close (fd);
  /* Write a random integer to memory-mapped area.  */
  sprintf((char*) file_memory, "%d\n", random_range (-100, 100));
  /* Release the memory (unnecessary because the program exits).  */
  munmap (file_memory, FILE_LENGTH);

  return 0;
}
```

The `mmap-write` program opens the file, creating it if it did not previously exist. The third argument to `open` specifies that the file is opened for reading and writing. Because we do not know the file's length, we use `lseek` to ensure that the file is large enough to store an integer and then move back the file position to its beginning.

The program maps the file and then closes the file descriptor because it's no longer needed. The program then writes a random integer to the mapped memory, and thus the file, and unmaps the memory. The `munmap` call is unnecessary because Linux would automatically unmap the file when the program terminates.

Listing 5.6 **(*mmap-read.c*) Read an Integer from a Memory-Mapped File, and Double It**

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

int main (int argc, char* const argv[])
{
  int fd;
  void* file_memory;
  int integer;

  /* Open the file.  */
  fd = open (argv[1], O_RDWR, S_IRUSR | S_IWUSR);
  /* Create the memory mapping.  */
  file_memory = mmap (0, FILE_LENGTH, PROT_READ | PROT_WRITE,
                      MAP_SHARED, fd, 0);
  close (fd);
```

continues

Listing 5.6 **Continued**

```
  /* Read the integer, print it out, and double it.  */
  scanf (file_memory, "%d", &integer);
  printf ("value: %d\n", integer);
  sprintf ((char*) file_memory, "%d\n", 2 * integer);
  /* Release the memory (unnecessary because the program exits).  */
  munmap (file_memory, FILE_LENGTH);

  return 0;
}
```

The `mmap-read` program reads the number out of the file and then writes the doubled value to the file. First, it opens the file and maps it for reading and writing. Because we can assume that the file is large enough to store an unsigned integer, we need not use `lseek`, as in the previous program. The program reads and parses the value out of memory using `sscanf` and then formats and writes the double value using `sprintf`.

Here's an example of running these example programs. It maps the file `/tmp/integer-file`.

```
% ./mmap-write /tmp/integer-file
% cat /tmp/integer-file
42
% ./mmap-read /tmp/integer-file
value: 42
% cat /tmp/integer-file
84
```

Observe that the text `42` was written to the disk file without ever calling `write`, and was read back in again without calling `read`. Note that these sample programs write and read the integer as a string (using `sprintf` and `sscanf`) for demonstration purposes only—there's no need for the contents of a memory-mapped file to be text. You can store and retrieve arbitrary binary in a memory-mapped file.

5.3.3 Shared Access to a File

Different processes can communicate using memory-mapped regions associated with the same file. Specify the `MAP_SHARED` flag so that any writes to these regions are immediately transferred to the underlying file and made visible to other processes. If you don't specify this flag, Linux may buffer writes before transferring them to the file.

Alternatively, you can force Linux to incorporate buffered writes into the disk file by calling `msync`. Its first two parameters specify a memory-mapped region, as for `munmap`. The third parameter can take these flag values:

- `MS_ASYNC`—The update is scheduled but not necessarily run before the call returns.
- `MS_SYNC`—The update is immediate; the call to `msync` blocks until it's done. `MS_SYNC` and `MS_ASYNC` may not both be used.

- MS_INVALIDATE—All other file mappings are invalidated so that they can see the updated values.

For example, to flush a shared file mapped at address mem_addr of length mem_length bytes, call this:

```
msync (mem_addr, mem_length, MS_SYNC | MS_INVALIDATE);
```

As with shared memory segments, users of memory-mapped regions must establish and follow a protocol to avoid race conditions. For example, a semaphore can be used to prevent more than one process from accessing the mapped memory at one time. Alternatively, you can use fcntl to place a read or write lock on the file, as described in Section 8.3, "fcntl: Locks and Other File Operations," in Chapter 8.

5.3.4 Private Mappings

Specifying MAP_PRIVATE to mmap creates a copy-on-write region. Any write to the region is reflected only in this process's memory; other processes that map the same file won't see the changes. Instead of writing directly to a page shared by all processes, the process writes to a private copy of this page. All subsequent reading and writing by the process use this page.

5.3.5 Other Uses for *mmap*

The mmap call can be used for purposes other than interprocess communications. One common use is as a replacement for read and write. For example, rather than explicitly reading a file's contents into memory, a program might map the file into memory and scan it using memory reads. For some programs, this is more convenient and may also run faster than explicit file I/O operations.

One advanced and powerful technique used by some programs is to build data structures (ordinary struct instances, for example) in a memory-mapped file. On a subsequent invocation, the program maps that file back into memory, and the data structures are restored to their previous state. Note, though, that pointers in these data structures will be invalid unless they all point within the same mapped region of memory and unless care is taken to map the file back into the same address region that it occupied originally.

Another handy technique is to map the special /dev/zero file into memory. That file, which is described in Section 6.5.2, "/dev/zero," of Chapter 6, "Devices," behaves as if it were an infinitely long file filled with 0 bytes. A program that needs a source of 0 bytes can mmap the file /dev/zero. Writes to /dev/zero are discarded, so the mapped memory may be used for any purpose. Custom memory allocators often map /dev/zero to obtain chunks of preinitialized memory.

5.4 Pipes

A *pipe* is a communication device that permits unidirectional communication. Data written to the "write end" of the pipe is read back from the "read end." Pipes are serial devices; the data is always read from the pipe in the same order it was written. Typically, a pipe is used to communicate between two threads in a single process or between parent and child processes.

In a shell, the symbol | creates a pipe. For example, this shell command causes the shell to produce two child processes, one for `ls` and one for `less`:

```
% ls | less
```

The shell also creates a pipe connecting the standard output of the `ls` subprocess with the standard input of the `less` process. The filenames listed by `ls` are sent to `less` in exactly the same order as if they were sent directly to the terminal.

A pipe's data capacity is limited. If the writer process writes faster than the reader process consumes the data, and if the pipe cannot store more data, the writer process blocks until more capacity becomes available. If the reader tries to read but no data is available, it blocks until data becomes available. Thus, the pipe automatically synchronizes the two processes.

5.4.1 Creating Pipes

To create a pipe, invoke the `pipe` command. Supply an integer array of size 2. The call to `pipe` stores the reading file descriptor in array position 0 and the writing file descriptor in position 1. For example, consider this code:

```
int pipe_fds[2];
int read_fd;
int write_fd;

pipe (pipe_fds);
read_fd = pipe_fds[0];
write_fd = pipe_fds[1];
```

Data written to the file descriptor `read_fd` can be read back from `write_fd`.

5.4.2 Communication Between Parent and Child Processes

A call to `pipe` creates file descriptors, which are valid only within that process and its children. A process's file descriptors cannot be passed to unrelated processes; however, when the process calls `fork`, file descriptors are copied to the new child process. Thus, pipes can connect only related processes.

In the program in Listing 5.7, a `fork` spawns a child process. The child inherits the pipe file descriptors. The parent writes a string to the pipe, and the child reads it out. The sample program converts these file descriptors into `FILE*` streams using `fdopen`. Because we use streams rather than file descriptors, we can use the higher-level standard C library I/O functions such as `printf` and `fgets`.

Listing 5.7 **(*pipe.c*) Using a Pipe to Communicate with a Child Process**

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

/* Write COUNT copies of MESSAGE to STREAM, pausing for a second
   between each.  */

void writer (const char* message, int count, FILE* stream)
{
  for (; count > 0; --count) {
    /* Write the message to the stream, and send it off immediately.  */
    fprintf (stream, "%s\n", message);
    fflush (stream);
    /* Snooze a while.  */
    sleep (1);
  }
}

/* Read random strings from the stream as long as possible.  */

void reader (FILE* stream)
{
  char buffer[1024];
  /* Read until we hit the end of the stream.  fgets reads until
     either a newline or the end-of-file.  */
  while (!feof (stream)
         && !ferror (stream)
         && fgets (buffer, sizeof (buffer), stream) != NULL)
    fputs (buffer, stdout);
}

int main ()
{
  int fds[2];
  pid_t pid;

  /* Create a pipe.  File descriptors for the two ends of the pipe are
     placed in fds.  */
  pipe (fds);
  /* Fork a child process.  */
  pid = fork ();
  if (pid == (pid_t) 0) {
    FILE* stream;
    /* This is the child process.  Close our copy of the write end of
       the file descriptor.  */
    close (fds[1]);
    /* Convert the read file descriptor to a FILE object, and read
       from it.  */
    stream = fdopen (fds[0], "r");
    reader (stream);
```

continues

Listing 5.7 **Continued**

```
    close (fds[0]);
  }
  else {
    /* This is the parent process.  */
    FILE* stream;
    /* Close our copy of the read end of the file descriptor.  */
    close (fds[0]);
    /* Convert the write file descriptor to a FILE object, and write
       to it.  */
    stream = fdopen (fds[1], "w");
    writer ("Hello, world.", 5, stream);
    close (fds[1]);
  }

  return 0;
}
```

At the beginning of `main`, `fds` is declared to be an integer array with size 2. The `pipe` call creates a pipe and places the read and write file descriptors in that array. The program then forks a child process. After closing the read end of the pipe, the parent process starts writing strings to the pipe. After closing the write end of the pipe, the child reads strings from the pipe.

Note that after writing in the `writer` function, the parent flushes the pipe by calling `fflush`. Otherwise, the string may not be sent through the pipe immediately.

When you invoke the command `ls | less`, two forks occur: one for the `ls` child process and one for the `less` child process. Both of these processes inherit the pipe file descriptors so they can communicate using a pipe. To have unrelated processes communicate, use a FIFO instead, as discussed in Section 5.4.5, "FIFOs."

5.4.3 Redirecting the Standard Input, Output, and Error Streams

Frequently, you'll want to create a child process and set up one end of a pipe as its standard input or standard output. Using the `dup2` call, you can equate one file descriptor with another. For example, to redirect a process's standard input to a file descriptor `fd`, use this line:

```
dup2 (fd, STDIN_FILENO);
```

The symbolic constant `STDIN_FILENO` represents the file descriptor for the standard input, which has the value 0. The call closes standard input and then reopens it as a duplicate of `fd` so that the two may be used interchangeably. Equated file descriptors share the same file position and the same set of file status flags. Thus, characters read from `fd` are not reread from standard input.

The program in Listing 5.8 uses `dup2` to send the output from a pipe to the `sort` command.[2] After creating a pipe, the program forks. The parent process prints some strings to the pipe. The child process attaches the read file descriptor of the pipe to its standard input using `dup2`. It then executes the `sort` program.

Listing 5.8 **(*dup2.c*) Redirect Output from a Pipe with *dup2***

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

int main ()
{
  int fds[2];
  pid_t pid;

  /* Create a pipe.  File descriptors for the two ends of the pipe are
     placed in fds.  */
  pipe (fds);
  /* Fork a child process.  */
  pid = fork ();
  if (pid == (pid_t) 0) {
    /* This is the child process.  Close our copy of the write end of
       the file descriptor.  */
    close (fds[1]);
    /* Connect the read end of the pipe to standard input.  */
    dup2 (fds[0], STDIN_FILENO);
    /* Replace the child process with the "sort" program.  */
    execlp ("sort", "sort", 0);
  }
  else {
    /* This is the parent process.  */
    FILE* stream;
    /* Close our copy of the read end of the file descriptor.  */
    close (fds[0]);
    /* Convert the write file descriptor to a FILE object, and write
       to it.  */
    stream = fdopen (fds[1], "w");
    fprintf (stream, "This is a test.\n");
    fprintf (stream, "Hello, world.\n");
    fprintf (stream, "My dog has fleas.\n");
    fprintf (stream, "This program is great.\n");
    fprintf (stream, "One fish, two fish.\n");
    fflush (stream);
    close (fds[1]);
    /* Wait for the child process to finish.  */
    waitpid (pid, NULL, 0);
  }

  return 0;
}
```

2. `sort` reads lines of text from standard input, sorts them into alphabetical order, and prints them to standard output.

5.4.4 *popen* and *pclose*

A common use of pipes is to send data to or receive data from a program being run in a subprocess. The `popen` and `pclose` functions ease this paradigm by eliminating the need to invoke `pipe`, `fork`, `dup2`, `exec`, and `fdopen`.

Compare Listing 5.9, which uses `popen` and `pclose`, to the previous example (Listing 5.8).

Listing 5.9 **(*popen.c*) Example Using *popen***

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

int main ()
{
  FILE* stream = popen ("sort", "w");
  fprintf (stream, "This is a test.\n");
  fprintf (stream, "Hello, world.\n");
  fprintf (stream, "My dog has fleas.\n");
  fprintf (stream, "This program is great.\n");
  fprintf (stream, "One fish, two fish.\n");
  return pclose (stream);
}
```

The call to `popen` creates a child process executing the `sort` command, replacing calls to `pipe`, `fork`, `dup2`, and `execlp`. The second argument, `"w"`, indicates that this process wants to write to the child process. The return value from `popen` is one end of a pipe; the other end is connected to the child process's standard input. After the writing finishes, `pclose` closes the child process's stream, waits for the process to terminate, and returns its status value.

The first argument to `popen` is executed as a shell command in a subprocess running `/bin/sh`. The shell searches the `PATH` environment variable in the usual way to find programs to execute. If the second argument is `"r"`, the function returns the child process's standard output stream so that the parent can read the output. If the second argument is `"w"`, the function returns the child process's standard input stream so that the parent can send data. If an error occurs, `popen` returns a null pointer.

Call `pclose` to close a stream returned by `popen`. After closing the specified stream, `pclose` waits for the child process to terminate.

5.4.5 FIFOs

A *first-in, first-out (FIFO)* file is a pipe that has a name in the filesystem. Any process can open or close the FIFO; the processes on either end of the pipe need not be related to each other. FIFOs are also called *named pipes*.

You can make a FIFO using the `mkfifo` command. Specify the path to the FIFO on the command line. For example, create a FIFO in `/tmp/fifo` by invoking this:

```
% mkfifo /tmp/fifo
% ls -l /tmp/fifo
prw-rw-rw-    1 samuel   users           0 Jan 16 14:04 /tmp/fifo
```

The first character of the output from `ls` is `p`, indicating that this file is actually a FIFO (named pipe). In one window, read from the FIFO by invoking the following:

```
% cat < /tmp/fifo
```

In a second window, write to the FIFO by invoking this:

```
% cat > /tmp/fifo
```

Then type in some lines of text. Each time you press Enter, the line of text is sent through the FIFO and appears in the first window. Close the FIFO by pressing Ctrl+D in the second window. Remove the FIFO with this line:

```
% rm /tmp/fifo
```

Creating a FIFO

Create a FIFO programmatically using the `mkfifo` function. The first argument is the path at which to create the FIFO; the second parameter specifies the pipe's owner, group, and world permissions, as discussed in Chapter 10, "Security," Section 10.3, "File System Permissions." Because a pipe must have a reader and a writer, the permissions must include both read and write permissions. If the pipe cannot be created (for instance, if a file with that name already exists), `mkfifo` returns –1. Include `<sys/types.h>` and `<sys/stat.h>` if you call `mkfifo`.

Accessing a FIFO

Access a FIFO just like an ordinary file. To communicate through a FIFO, one program must open it for writing, and another program must open it for reading. Either low-level I/O functions (`open`, `write`, `read`, `close`, and so on, as listed in Appendix B, "Low-Level I/O") or C library I/O functions (`fopen`, `fprintf`, `fscanf`, `fclose`, and so on) may be used.

For example, to write a buffer of data to a FIFO using low-level I/O routines, you could use this code:

```
int fd = open (fifo_path, O_WRONLY);
write (fd, data, data_length);
close (fd);
```

To read a string from the FIFO using C library I/O functions, you could use this code:

```
FILE* fifo = fopen (fifo_path, "r");
fscanf (fifo, "%s", buffer);
fclose (fifo);
```

A FIFO can have multiple readers or multiple writers. Bytes from each writer are written atomically up to a maximum size of `PIPE_BUF` (4KB on Linux). Chunks from simultaneous writers can be interleaved. Similar rules apply to simultaneous reads.

Differences from Windows Named Pipes

Pipes in the Win32 operating systems are very similar to Linux pipes. (Refer to the Win32 library documentation for technical details about these.) The main differences concern named pipes, which, for Win32, function more like sockets. Win32 named pipes can connect processes on separate computers connected via a network. On Linux, sockets are used for this purpose. Also, Win32 allows multiple reader-writer connections on a named pipe without interleaving data, and pipes can be used for two-way communication.[3]

5.5 Sockets

A *socket* is a bidirectional communication device that can be used to communicate with another process on the same machine or with a process running on other machines. Sockets are the only interprocess communication we'll discuss in this chapter that permit communication between processes on different computers. Internet programs such as Telnet, rlogin, FTP, talk, and the World Wide Web use sockets.

For example, you can obtain the WWW page from a Web server using the Telnet program because they both use sockets for network communications.[4] To open a connection to a WWW server at `www.codesourcery.com`, use `telnet www.codesourcery.com 80`. The magic constant 80 specifies a connection to the Web server programming running `www.codesourcery.com` instead of some other process. Try typing `GET /` after the connection is established. This sends a message through the socket to the Web server, which replies by sending the home page's HTML source and then closing the connection—for example:

```
% telnet www.codesourcery.com 80
Trying 206.168.99.1...
Connected to merlin.codesourcery.com (206.168.99.1).
Escape character is '^]'.
GET /
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
...
```

3. Note that only Windows NT can create a named pipe; Windows 9*x* programs can form only client connections.

4. Usually, you'd use `telnet` to connect a Telnet server for remote logins. But you can also use `telnet` to connect to a server of a different kind and then type comments directly at it.

5.5.1 Socket Concepts

When you create a socket, you must specify three parameters: communication style, namespace, and protocol.

A communication style controls how the socket treats transmitted data and specifies the number of communication partners. When data is sent through a socket, it is packaged into chunks called *packets*. The communication style determines how these packets are handled and how they are addressed from the sender to the receiver.

- *Connection* styles guarantee delivery of all packets in the order they were sent. If packets are lost or reordered by problems in the network, the receiver automatically requests their retransmission from the sender.

  A connection-style socket is like a telephone call: The addresses of the sender and receiver are fixed at the beginning of the communication when the connection is established.

- *Datagram* styles do not guarantee delivery or arrival order. Packets may be lost or reordered in transit due to network errors or other conditions. Each packet must be labeled with its destination and is not guaranteed to be delivered. The system guarantees only "best effort," so packets may disappear or arrive in a different order than shipping.

  A datagram-style socket behaves more like postal mail. The sender specifies the receiver's address for each individual message.

A socket namespace specifies how *socket addresses* are written. A socket address identifies one end of a socket connection. For example, socket addresses in the "local namespace" are ordinary filenames. In "Internet namespace," a socket address is composed of the Internet address (also known as an *Internet Protocol address* or *IP address*) of a host attached to the network and a port number. The port number distinguishes among multiple sockets on the same host.

A protocol specifies how data is transmitted. Some protocols are TCP/IP, the primary networking protocols used by the Internet; the AppleTalk network protocol; and the UNIX local communication protocol. Not all combinations of styles, namespaces, and protocols are supported.

5.5.2 System Calls

Sockets are more flexible than previously discussed communication techniques. These are the system calls involving sockets:

`socket`—Creates a socket

`closes`—Destroys a socket

`connect`—Creates a connection between two sockets

`bind`—Labels a server socket with an address

`listen`—Configures a socket to accept conditions

`accept`—Accepts a connection and creates a new socket for the connection

Sockets are represented by file descriptors.

Creating and Destroying Sockets

The `socket` and `close` functions create and destroy sockets, respectively. When you create a socket, specify the three socket choices: namespace, communication style, and protocol. For the namespace parameter, use constants beginning with `PF_` (abbreviating "protocol families"). For example, `PF_LOCAL` or `PF_UNIX` specifies the local namespace, and `PF_INET` specifies Internet namespaces. For the communication style parameter, use constants beginning with `SOCK_`. Use `SOCK_STREAM` for a connection-style socket, or use `SOCK_DGRAM` for a datagram-style socket.

The third parameter, the protocol, specifies the low-level mechanism to transmit and receive data. Each protocol is valid for a particular namespace-style combination. Because there is usually one best protocol for each such pair, specifying 0 is usually the correct protocol. If `socket` succeeds, it returns a file descriptor for the socket. You can read from or write to the socket using `read`, `write`, and so on, as with other file descriptors. When you are finished with a socket, call `close` to remove it.

Calling *connect*

To create a connection between two sockets, the client calls `connect`, specifying the address of a server socket to connect to. A *client* is the process initiating the connection, and a *server* is the process waiting to accept connections. The client calls `connect` to initiate a connection from a local socket to the server socket specified by the second argument. The third argument is the length, in bytes, of the address structure pointed to by the second argument. Socket address formats differ according to the socket namespace.

Sending Information

Any technique to write to a file descriptor can be used to write to a socket. See Appendix B for a discussion of Linux's low-level I/O functions and some of the issues surrounding their use. The `send` function, which is specific to the socket file descriptors, provides an alternative to `write` with a few additional choices; see the man page for information.

5.5.3 Servers

A server's life cycle consists of the creation of a connection-style socket, binding an address to its socket, placing a call to `listen` that enables connections to the socket, placing calls to `accept` incoming connections, and then closing the socket. Data isn't read and written directly via the server socket; instead, each time a program accepts a new connection, Linux creates a separate socket to use in transferring data over that connection. In this section, we introduce `bind`, `listen`, and `accept`.

An address must be bound to the server's socket using `bind` if a client is to find it. Its first argument is the socket file descriptor. The second argument is a pointer to a socket address structure; the format of this depends on the socket's address family. The third argument is the length of the address structure, in bytes. When an address is bound to a connection-style socket, it must invoke `listen` to indicate that it is a server. Its first argument is the socket file descriptor. The second argument specifies how many pending connections are queued. If the queue is full, additional connections will be rejected. This does not limit the total number of connections that a server can handle; it limits just the number of clients attempting to connect that have not yet been accepted.

A server accepts a connection request from a client by invoking `accept`. The first argument is the socket file descriptor. The second argument points to a socket address structure, which is filled with the client socket's address. The third argument is the length, in bytes, of the socket address structure. The server can use the client address to determine whether it really wants to communicate with the client. The call to accept creates a new socket for communicating with the client and returns the corresponding file descriptor. The original server socket continues to accept new client connections. To read data from a socket without removing it from the input queue, use `recv`. It takes the same arguments as `read`, plus an additional `FLAGS` argument. A flag of `MSG_PEEK` causes data to be read but not removed from the input queue.

5.5.4 Local Sockets

Sockets connecting processes on the same computer can use the local namespace represented by the synonyms `PF_LOCAL` and `PF_UNIX`. These are called *local sockets* or *UNIX-domain sockets*. Their socket addresses, specified by filenames, are used only when creating connections.

The socket's name is specified in `struct sockaddr_un`. You must set the `sun_family` field to `AF_LOCAL`, indicating that this is a local namespace. The `sun_path` field specifies the filename to use and may be, at most, 108 bytes long. The actual length of `struct sockaddr_un` should be computed using the `SUN_LEN` macro. Any filename can be used, but the process must have directory write permissions, which permit adding files to the directory. To connect to a socket, a process must have read permission for the file. Even though different computers may share the same filesystem, only processes running on the same computer can communicate with local namespace sockets.

The only permissible protocol for the local namespace is 0.

Because it resides in a file system, a local socket is listed as a file. For example, notice the initial `s`:

```
% ls -l /tmp/socket
srwxrwx--x    1 user   group    0 Nov 13 19:18 /tmp/socket
```

Call `unlink` to remove a local socket when you're done with it.

5.5.5 An Example Using Local Namespace Sockets

We illustrate sockets with two programs. The server program, in Listing 5.10, creates a local namespace socket and listens for connections on it. When it receives a connection, it reads text messages from the connection and prints them until the connection closes. If one of these messages is "quit," the server program removes the socket and ends. The `socket-server` program takes the path to the socket as its command-line argument.

Listing 5.10 **(*socket-server.c*) Local Namespace Socket Server**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

/* Read text from the socket and print it out.  Continue until the
   socket closes.  Return nonzero if the client sent a "quit"
   message, zero otherwise.  */

int server (int client_socket)
{
  while (1) {
    int length;
    char* text;

    /* First, read the length of the text message from the socket.  If
       read returns zero, the client closed the connection.  */
    if (read (client_socket, &length, sizeof (length)) == 0)
      return 0;
    /* Allocate a buffer to hold the text.  */
    text = (char*) malloc (length);
    /* Read the text itself, and print it.  */

    read (client_socket, text, length);
    printf ("%s\n", text);
    /* Free the buffer.  */
    free (text);
    /* If the client sent the message "quit," we're all done.  */
    if (!strcmp (text, "quit"))
      return 1;
  }
}

int main (int argc, char* const argv[])
{
const char* const socket_name = argv[1];
```

```
  int socket_fd;
  struct sockaddr_un name;
  int client_sent_quit_message;

  /* Create the socket.  */
  socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
  /* Indicate that this is a server.  */
  name.sun_family = AF_LOCAL;
  strcpy (name.sun_path, socket_name);
  bind (socket_fd, &name, SUN_LEN (&name));
  /* Listen for connections.  */
  listen (socket_fd, 5);

  /* Repeatedly accept connections, spinning off one server() to deal
     with each client.  Continue until a client sends a "quit" message.  */
  do {
    struct sockaddr_un client_name;
    socklen_t client_name_len;
    int client_socket_fd;

    /* Accept a connection.  */
    client_socket_fd = accept (socket_fd, &client_name, &client_name_len);
    /* Handle the connection.  */
    client_sent_quit_message = server (client_socket_fd);
    /* Close our end of the connection.  */
    close (client_socket_fd);
  }
  while (!client_sent_quit_message);

  /* Remove the socket file.  */
  close (socket_fd);
  unlink (socket_name);

  return 0;
}
```

The client program, in Listing 5.11, connects to a local namespace socket and sends a message. The name path to the socket and the message are specified on the command line.

Listing 5.11 **(*socket-client.c*) Local Namespace Socket Client**

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
```

continues

Listing 5.11 **Continued**

```
/* Write TEXT to the socket given by file descriptor SOCKET_FD.  */

void write_text (int socket_fd, const char* text)
{
  /* Write the number of bytes in the string, including
     NUL-termination.  */
  int length = strlen (text) + 1;
  write (socket_fd, &length, sizeof (length));
  /* Write the string.  */
  write (socket_fd, text, length);
}

int main (int argc, char* const argv[])
{
  const char* const socket_name = argv[1];
  const char* const message = argv[2];
  int socket_fd;
  struct sockaddr_un name;

  /* Create the socket.  */
  socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
  /* Store the server's name in the socket address.  */
  name.sun_family = AF_LOCAL;
  strcpy (name.sun_path, socket_name);
  /* Connect the socket.  */
  connect (socket_fd, &name, SUN_LEN (&name));
  /* Write the text on the command line to the socket.  */
  write_text (socket_fd, message);
  close (socket_fd);
  return 0;
}
```

Before the client sends the message text, it sends the length of that text by sending the bytes of the integer variable `length`. Likewise, the server reads the length of the text by reading from the socket into an integer variable. This allows the server to allocate an appropriately sized buffer to hold the message text before reading it from the socket.

To try this example, start the server program in one window. Specify a path to a socket—for example, `/tmp/socket`.

```
% ./socket-server /tmp/socket
```

In another window, run the client a few times, specifying the same socket path plus messages to send to the client:

```
% ./socket-client /tmp/socket "Hello, world."
% ./socket-client /tmp/socket "This is a test."
```

The server program receives and prints these messages. To close the server, send the message "quit" from a client:

```
% ./socket-client /tmp/socket "quit"
```

The server program terminates.

5.5.6 Internet-Domain Sockets

UNIX-domain sockets can be used only for communication between two processes on the same computer. *Internet-domain sockets*, on the other hand, may be used to connect processes on different machines connected by a network.

Sockets connecting processes through the Internet use the Internet namespace represented by `PF_INET`. The most common protocols are TCP/IP. The *Internet Protocol (IP)*, a low-level protocol, moves packets through the Internet, splitting and rejoining the packets, if necessary. It guarantees only "best-effort" delivery, so packets may vanish or be reordered during transport. Every participating computer is specified using a unique IP number. The *Transmission Control Protocol (TCP)*, layered on top of IP, provides reliable connection-ordered transport. It permits telephone-like connections to be established between computers and ensures that data is delivered reliably and in order.

> **DNS Names**
>
> Because it is easier to remember names than numbers, the *Domain Name Service (DNS)* associates names such as `www.codesourcery.com` with computers' unique IP numbers. DNS is implemented by a worldwide hierarchy of name servers, but you don't need to understand DNS protocols to use Internet host names in your programs.

Internet socket addresses contain two parts: a machine and a port number. This information is stored in a `struct sockaddr_in` variable. Set the `sin_family` field to `AF_INET` to indicate that this is an Internet namespace address. The `sin_addr` field stores the Internet address of the desired machine as a 32-bit integer IP number. A *port number* distinguishes a given machine's different sockets. Because different machines store multibyte values in different byte orders, use `htons` to convert the port number to *network byte order*. See the man page for `ip` for more information.

To convert human-readable hostnames, either numbers in standard dot notation (such as 10.0.0.1) or DNS names (such as `www.codesourcery.com`) into 32-bit IP numbers, you can use `gethostbyname`. This returns a pointer to the `struct hostent` structure; the `h_addr` field contains the host's IP number. See the sample program in Listing 5.12.

Listing 5.12 illustrates the use of Internet-domain sockets. The program obtains the home page from the Web server whose hostname is specified on the command line.

Listing 5.12 (*socket-inet.c*) **Read from a WWW Server**

```
#include <stdlib.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string.h>

/* Print the contents of the home page for the server's socket.
   Return an indication of success.  */

void get_home_page (int socket_fd)
{
  char buffer[10000];
  ssize_t number_characters_read;

  /* Send the HTTP GET command for the home page.  */
  sprintf (buffer, "GET /\n");
  write (socket_fd, buffer, strlen (buffer));
  /* Read from the socket.  The call to read may not
  return all the data at one time, so keep
  trying until we run out.  */
  while (1) {
    number_characters_read = read (socket_fd, buffer, 10000);
    if (number_characters_read == 0)
      return;
    /* Write the data to standard output.  */
    fwrite (buffer, sizeof (char), number_characters_read, stdout);
  }
}

int main (int argc, char* const argv[])
{
  int socket_fd;
  struct sockaddr_in name;
  struct hostent* hostinfo;

  /* Create the socket.  */
  socket_fd = socket (PF_INET, SOCK_STREAM, 0);
  /* Store the server's name in the socket address.  */
  name.sin_family = AF_INET;
  /* Convert from strings to numbers.  */
  hostinfo = gethostbyname (argv[1]);
  if (hostinfo == NULL)
    return 1;
  else
    name.sin_addr = *((struct in_addr *) hostinfo->h_addr);
  /* Web servers use port 80.  */
  name.sin_port = htons (80);
```

```
  /* Connect to the Web server  */
  if (connect (socket_fd, &name, sizeof (struct sockaddr_in)) == -1) {
    perror ("connect");
    return 1;
  }
  /* Retrieve the server's home page.  */
  get_home_page (socket_fd);

  return 0;
}
```

This program takes the hostname of the Web server on the command line (not a URL—that is, without the "http://"). It calls `gethostbyname` to translate the hostname into a numerical IP address and then connects a stream (TCP) socket to port 80 on that host. Web servers speak the *Hypertext Transport Protocol (HTTP)*, so the program issues the HTTP `GET` command and the server responds by sending the text of the home page.

Standard Port Numbers

By convention, Web servers listen for connections on port 80. Most Internet network services are associated with a standard port number. For example, secure Web servers that use SSL listen for connections on port 443, and mail servers (which speak SMTP) use port 25.

On GNU/Linux systems, the associations between protocol/service names and standard port numbers are listed in the file `/etc/services`. The first column is the protocol or service name. The second column lists the port number and the connection type: `tcp` for connection-oriented, or `udp` for datagram.

If you implement custom network services using Internet-domain sockets, use port numbers greater than 1024.

For example, to retrieve the home page from the Web site `www.codesourcery.com`, invoke this:

```
% ./socket-inet www.codesourcery.com
<html>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">

...
```

5.5.7 Socket Pairs

As we saw previously, the `pipe` function creates two file descriptors for the beginning and end of a pipe. Pipes are limited because the file descriptors must be used by related processes and because communication is unidirectional. The `socketpair` function creates two file descriptors for two connected sockets on the same computer. These file descriptors permit two-way communication between related processes.

Its first three parameters are the same as those of the socket call: They specify the domain, connection style, and protocol. The last parameter is a two-integer array, which is filled with the file descriptions of the two sockets, similar to pipe. When you call socketpair, you must specify PF_LOCAL as the domain.