COURSE TECHNOLOGY
CENGAGE Learning
Professional • Technical • Reference

GNU/LINUX APPLICATION PROGRAMMING

Second Edition

- Focuses on the GNU tools and libraries that make Linux programming possible
- Covers a variety of useful APIs for process management, shared memory, message queues, semaphores, POSIX, file handling, sockets, and more
- Includes a CD-ROM with code snippets for all the detailed APIs and the figures from the book

PROGRAMMING SERIES

M. TIM JONES

GNU/Linux Application Programming

Second Edition

M. Tim Jones

Charles River Media
A part of Course Technology, Cengage Learning

COURSE TECHNOLOGY
CENGAGE Learning™

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States

COURSE TECHNOLOGY
CENGAGE Learning™

GNU/Linux Application Programming, Second Edition

M. Tim Jones

Publisher and General Manager, Course Technology PTR: Stacy L. Hiquet

Associate Director of Marketing: Sarah Panella

Manager of Editorial Services: Heather Talbot

Marketing Manager: Mark Hughes

Acquisitions Editor: Mitzi Koontz

Project Editor: Marta Justak

Technical Reviewer: Jim Lieb

Editorial Services Coordinator: Jen Blaney

Copy Editor: Kevin Kent

Interior Layout Tech: Judy Littlefield

Cover Designer: Tyler Creative Services

Indexer: Joan Green

Proofreader: Kate Shoup

CD-ROM Producer: Brandon Penticuff

Library of Congress Control Number: 2007939373
ISBN-13: 978-1-58450-568-6
ISBN-10: 1-58450-568-0
eISBN-10: 1-58450-610-5

Course Technology
25 Thomson Place
Boston, MA 02210
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at:
international.cengage.com/region

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

For your lifelong learning solutions, visit **courseptr.com**
Visit our corporate website at **cengage.com**

Printed in the United States of America
1 2 3 4 5 6 7 11 10 09 08

This book is dedicated to my wife Jill,
and my children Megan, Elise, and Marc—especially Elise,
who always looks for what's most important in my books.

This page intentionally left blank

Contents

Acknowledgments

My first exposure to open source was in the summer of 1994. I had just come off a project building an operating system kernel for a large geosynchronous communication spacecraft in the Ada language on the MIL-STD-1750A microprocessor. The Ada language was technically very nice, safe, and easily readable. The MIL-STD-1750A processor was old, even by early 1990 standards. (It was a 1970s instruction set architecture designed for military avionics, but was still very elegant in its simplicity.)

I moved on to work on a research satellite to study gamma ray bursts, and on the side, supported the validation of a project called "1750GALS." This project, managed by Oliver Kellogg, consisted of a GCC compiler, assembler, linker, and simulator for the Ada language targeted to the 1750A processor family. Since I had some background in Ada and the 1750A, and the gamma ray burst project was just ramping up, I loaned some time to its validation. Some number of months later, I saw a post in the comp.compilers usenet group, of which a snippet is provided below:

```
`1750GALS', the MIL-STD-1750 Gcc/Assembler/Linker/Simulator, now has a
European FTP home, and an American FTP mirror.
[snip]
Kudos to Pekka Ruuska of VTT Inc. (Pekka.Ruuska@vtt.fi), and M. Tim
Jones of MIT Space Research (mtj@space.mit.edu), whose bugreports made
the toolkit as useable as it now is. Further, Tim Jones kindly set up
the U.S. FTP mirror. [1]
```

I was automatically world famous, and my 15 minutes of fame had begun. An exaggeration, of course, but my time devoted to helping this project was both interesting and worthwhile and introduced me to the growing world of Free Software (which was already 10 years old at this time) and Open Source (whose name would not be coined for another three years).

---

[1] "[announce] 1750GALS now have an FTP home" http://compilers.iecc.com/comparch/article/94-11-043

This second edition is the result not only of many months of hard work, but also of many decades of tireless work by UNIX and GNU tools developers around the world. Since an entire book could be written about the countless number of developers who created and advanced these efforts, I'll whittle it down to four people who (in my opinion) made the largest contributions toward the GNU/Linux operating system:

Dennis Ritchie and Ken Thompson of AT&T Bell Labs built the first UNIX operating system (and subsequent variants) and also the C programming language.

Richard Stallman (father of GNU and the Free Software Foundation) motivated and brought together other free thinkers around the world to build the world-class GNU/Linux operating system.

And last, but not least, Linus Torvalds introduced the Linux kernel and remains the gatekeeper of the kernel source and a major contributor.

I'm also extremely grateful to Jim Lieb, whose wealth of UNIX knowledge and comprehensive review of this text improved it in innumerable ways. Thanks also to Cengage (in particular Jen Blaney and Marta Justak) for making this second edition possible.

FIGURE I.1 Copyright (C) 1999, Free Software Foundation, Inc.
Permission is granted to copy, distribute and/or modify this image under the terms in the GNU General Public License or GNU Free Documentation License.

About the Author

M. Tim Jones is an embedded software architect and the author of numerous books, including *Artificial Intelligence: A Systems Approach*, *AI Application Programming*, *BSD Sockets Programming from a Multilanguage Perspective*, and many articles on a variety of technical subjects. His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a consultant engineer for Emulex Corp. in Longmont, Colorado.

This page intentionally left blank

Reader's Guide

This book was written with GNU/Linux application developers in mind. You'll note that topics such as the Linux kernel or device drivers are absent. This was intentional, and while they're fascinating topics in their own right, they are rarely necessary to develop applications and tools in the GNU/Linux environment.

This book is split into five parts, each focusing on different aspects of GNU/Linux programming. Part I, "Introduction," introduces GNU/Linux for the beginner. It addresses the GNU/Linux architecture, a short introduction to the process model and also licenses, and a brief introduction to open source development and licenses. Linux virtualization is also explored, including models and options in Linux.

Part II, "GNU Tools," concentrates on the necessary tools for GNU/Linux programming. The de facto standard GNU compiler tool chain is explored, along with the GNU `make` automated build system. Building and using libraries (both static and dynamic) are then investigated. Coverage testing and profiling are explored, using the `gcov` and `gprof` utilities, as is application bundling and distribution with `automake` and `autoconf`. Finally, source control is reviewed with some of the popular options on Linux and also data visualization with Gnuplot.

With an introduction to the GNU/Linux architecture and necessary tools for application development, we focus next in Part III, "Application Development Topics," on the most useful of the services available within GNU/Linux. This includes pipes, Sockets programming, dealing with files, both traditional processes and POSIX threads, message queues, semaphores, and finally shared memory management.

In Part IV, "GNU/Linux Shells and Scripting," we move up to application development using shells and scripting languages. Some of the most useful GNU/Linux commands that you'll encounter in programming on GNU/Linux are covered, and there is a tutorial on the Bourne-Again Shell (`bash`). Text processing is explored using two of the most popular string processing languages (`awk` and `sed`). We'll also look at the topic of parser generation using GNU Flex and Bison utilities (`lex` and `yacc`-compatible parser generator). Scripting with Ruby and Python is investigated as well.

In Part V, "Debugging and Testing," debugging is addressed using a variety of different aspects. We investigate some of the unit-testing frameworks that can help in automated regression. The GNU Debugger is introduced, with treatment of the most common commands and techniques. Finally, the topic of code hardening is explored along with a variety of debugging tools and techniques to assist in the development of reliable and secure GNU/Linux applications.

While the book was written with an implicit order in mind, each chapter can be read in isolation, depending upon your needs. Where applicable, references to other chapters are provided if more information is needed on a related topic.

Threads in This Book

This book can be read part-by-part and chapter-by-chapter, but a number of threads run through it that can be followed independently. A reader interested in pursuing a particular aspect of the GNU/Linux operating system can concentrate on the following sets of chapters for a given topic thread.

- GNU/Linux Inter-Process Communication Methods: Chapters 12, 13, 15, 16, 17, and 18.
- Scripting and Text Processing: Chapters 10, 21, 22, 23, 24, 25, 26, and 27.
- Building Efficient and Reliable GNU/Linux Applications: Chapters 5, 29, 31, 33, and 34.
- Multiprocess and Multithreaded Applications: Chapters 14 and 15.
- GNU/Linux Testing and Profiling: Chapters 29, 30, 32, and 33.
- GNU Tools for Application Development: Chapters 5, 6, 8, 9, 30, and 34.
- GNU Tools for Packaging and Distribution: Chapters 6, 8, 21, and 28.

Introduction

GNU/Linux is the Swiss army knife of operating systems. You'll find it in the smallest devices (such as an Apple iPod) to the largest most powerful supercomputers (like IBM's Blue Gene). You'll also find GNU/Linux running on the most diverse architectures, from the older x86 processors to the latest cell processor that powers the PlayStation 3 console.

This book provides the basis for application development on the GNU/Linux operating system. Whether you're developing applications for an iPod or a Blue Gene, this book covers the APIs and concepts that you'll need.

What You'll Find in This Book

This book provides everything that you'll need to develop applications in the GNU/Linux environment. Split into five distinct parts, the book covers GNU tools, topics in application development, shells and scripting, debugging and hardening, and introductory toptics, including the fundamentals of virtualization.

Some of the specific topics that you'll explore include:

- GNU/Linux architecture and virtualization mechanisms.
- GNU Tools such as GCC, `make`, `automake`/`autoconf`, source control systems, the GNU Debugger, and GNUplot.
- Fundamental application development topics such as libraries (static and dynamic), file handling, Pipes, Sockets, programming, and more.
- GNU/Linux process models (including threads) and POSIX IPC mechanisms (message queues, semaphores, and shared memory).
- Shells and scripting basics, from useful GNU/Linux commands to Bash, Ruby, and Python. Text processing is also covered with `sed`, `awk`, and parser construction with `flex` and `bison`.

- Debugging and hardening techniques are also covered, including software testing tools, coverage testing and profiling with GCov and GProf, and memory debugging tools (such as valgrind and others).

Who This Book Is For

This book is written for beginning and intermediate GNU/Linux programmers who want to learn how to develop applications on the GNU/Linux operating system or to extend their knowledge into more advanced areas of development. The book covers tools, APIs, and techniques with many examples illustrating the use of GNU/Linux APIs.

Part I Introduction

Chapter 1: GNU/Linux History
Chapter 2: GNU/Linux Architecture
Chapter 3: Free Software Development
Chapter 4: Linux Virtualization and Emulation

This first part of the book explores a few introductory topics of the GNU/Linux operating system and its development paradigm. This includes a short history of UNIX, GNU, and the GNU/Linux operating system; a quick review of the GNU/Linux architecture; a discussion of the free software (and open source) development paradigm; and finally a short tour of Linux virtualization and emulation solutions.

Chapter 1: GNU/Linux History

The history of GNU/Linux actually started in 1969 with the development of the first UNIX operating system. This chapter discusses the UNIX development history and the motivations (and frustrations) of key developers that led up to the release of the GNU/Linux operating system.

Chapter 2: GNU/Linux Architecture

The deconstruction of the GNU/Linux operating system is the topic of the second chapter. The chapter identifies the major elements of the GNU/Linux operating system and then breaks them down to illustrate how the operating system works at a high level.

Chapter 3: Free Software Development

The free software and open source development paradigms are detailed in this chapter, including some of the licenses that are available for free software. The two major types of open development called free software and open source are discussed, as well as some of the problems that exist within the domains.

Chapter 4: Linux Virtualization and Emulation

Virtualization is a technique that allows multiple operating systems to be executed concurrently on the same machine. The operating systems need not be the same type, nor for the same architecture (x86, PowerPC, and so forth). This chapter explores the basics of virtualization techniques and then introduces a few of the virtualization and emulation options available for Linux.

1 GNU/Linux History

In This Chapter

- UNIX History
- Richard Stallman and the GNU Movement
- Linus Torvalds and the Linux Kernel

Introduction

Before we jump into the technical aspects of GNU/Linux, let's invest a little time in the history of the GNU/Linux operating system (and why we use the term GNU/Linux). We'll review the beginnings of the GNU/Linux operating system by looking at its two primary sources and the two individuals who made it happen.

History of the UNIX Operating System

To understand GNU/Linux, let's first step back to 1969 to look at the history of the UNIX operating system. Although UNIX has existed for over 30 years, it is one of the most flexible and powerful operating systems to have ever been created. A timeline is shown in Figure 1.1.

The goals for UNIX were to provide a multitasking and multiuser operating system that supported application portability. This tradition has continued in all UNIX variants, and given the new perspective of operating system portability (runs on many platforms), UNIX continues to evolve and grow.

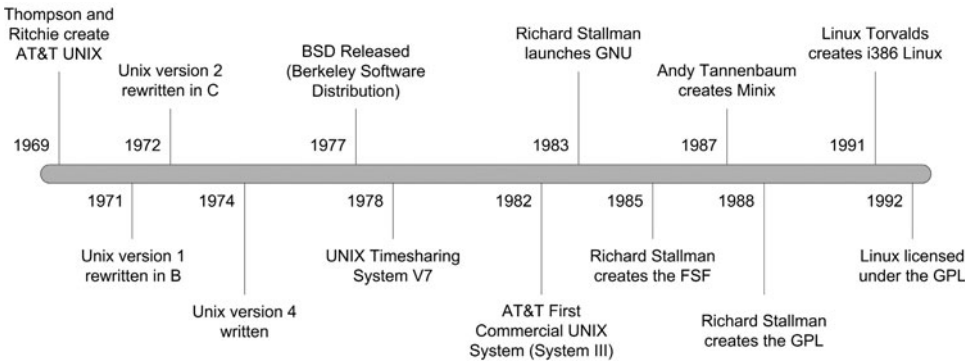

FIGURE 1.1 Timeline of UNIX/Linux and the GNU [RobotWisdom02].

AT&T UNIX

UNIX began as a small research project at AT&T Bell Labs in 1969 for the DEC PDP-7. Dennis Ritchie and Ken Thompson designed and built UNIX as a way to replace the Multics operating system then in use.

After Multics was withdrawn as the operating system at AT&T, Thompson and Ritchie developed UNIX on the PDP-7 in order to play a popular game at the time called *Space Travel* [Unix/Linux History04].

The first useful version of UNIX (version 1) was introduced in late 1971. This version of UNIX was written in the B language (precursor of the C language). It hosted a small number of commands, many of which are still available in UNIX and Linux systems today (such as `cat`, `cp`, `ls`, and `who`). In 1972, UNIX was rewritten in the newly created C language. In the next three years, UNIX continued to evolve, with four new versions produced. In 1979, the Bourne shell was introduced. Its ancestor, the bash shell, is the topic of Chapter 22, "Bourne-Again Shell (Bash)" [Unix History94].

BSD

The BSD (Berkeley Software Distribution) operating system was created as a fork of UNIX at the University of California at Berkeley in 1976. BSD remains not only a strong competitor to GNU/Linux, but in some ways is superior. Many innovations were created in the BSD, including the Sockets network programming paradigm and the variety of IPC mechanisms (addressed in Part III of this book, "Application Development Topics"). Many of the useful applications that we find in GNU/Linux today have their roots in BSD. For example, the vi editor and termcap (which allows programs to deal with displays in a display-agnostic manner) were created by Bill Joy at Berkeley in 1978 [Byte94].

One of the primary differences between BSD and GNU/Linux is in licensing. We'll address this disparity in Chapter 3, "Free Software Development."

GNU/Linux History

The history of GNU/Linux is actually two separate stories that came together to produce a world-class operating system. Richard Stallman created an organization to build a UNIX-like operating system. He had tools, a compiler, and a variety of applications, but he lacked a kernel. Linus Torvalds had a kernel, but no tools or applications for which to make it useful.

A controversial question about GNU/Linux is why it's called *GNU/Linux*, as opposed to the commonly used name *Linux*. The answer is very simple. *Linux* refers to the kernel (or the core of the operating system), which was initially developed by Linus Torvalds. The remaining software—the shells, compiler tool chain, utilities and tools, and plethora of applications—operate above the kernel. Much of this software is GNU software. In fact, the source code that makes up the GNU/Linux operating system dwarfs that of the kernel. Therefore, to call the entire operating system *Linux* is a misnomer, to say the least.

Richard Stallman provides an interesting perspective on this controversy in his article, "Linux and the GNU Project" [Linux/GNU04].

GNU and the Free Software Foundation

Richard Stallman, the father of open source, began the movement in 1983 with a post to the net.unix-wizards Usenet group soliciting help in the development of a free UNIX-compatible operating system [Stallman83]. Stallman's vision was the development of a free (as in freedom) UNIX-like operating system whose source was open and available to anyone.

Even in the 1970s, Stallman was no stranger to open source. He wrote the Emacs editor (1976) and gave the source away to anyone who would send a tape (on which to copy the source) and a return envelope.

The impetus for Stallman to create a free operating system was the fact that a modern computer required a proprietary operating system to do anything useful. These operating systems were closed and not modifiable by end users. In fact, until very recently, it was impossible to buy a PC from a major supplier without having to buy the Windows operating system on it. But through the Free Software Foundation (FSF), Stallman collected hundreds of programmers around the world to help take on the task.

By 1991, Stallman had pulled together many of the elements of a useful operating system. This included a compiler, a shell, and a variety of tools and applications. Work was underway in 1986 to migrate MIT's TRIX kernel, but divisions existed on whether to use TRIX or CMU's Mach microkernel. It was not until 1990 that work began on the official GNU Project kernel [Stallman02].

The Linux Kernel

Our story left off with the development of an operating system by the FSF, but development issues existed with a kernel that would make it complete. In an odd twist of fate, a young programmer by the name of Linus Torvalds announced the development of a "hobby" operating system for i386-based computers. Torvalds wanted to improve on the Minix operating system (which was widely used in the day) and thought a monolithic kernel would be much faster than the microkernel that Minix used. (While this is commonly believed to be true, operating systems such as Carnegie Mellon's Mach and the commercial QNX and Neutrino microkernels provide evidence to the contrary [Montague03].)

Torvalds released his first version of Linux (0.01) in 1991, and then later in the year he released version 0.11, which was a self-hosted release (see Figure 1.2). Torvalds used the freely available GNU tools such as the compiler and the bash shell for this effort. Much like Thompson and Ritchie's first UNIX more than 20 years earlier, it was minimal and not entirely useful. In 1992, Linux 0.96, which supported the X Window System, was released. That year also marked Linux as a GNU software component.

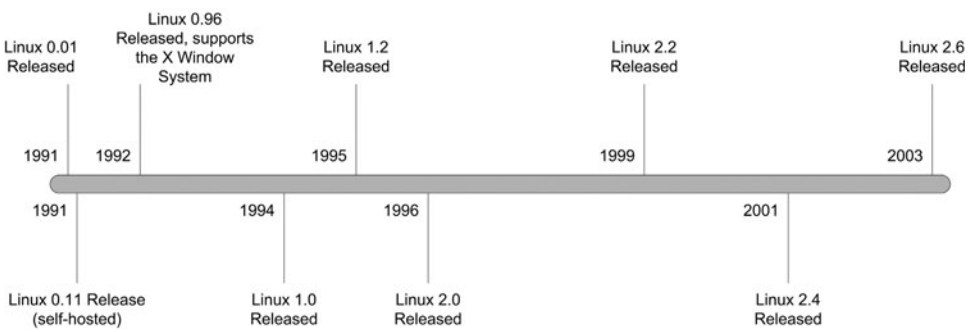

FIGURE 1.2 Linux development timeline [Wikipedia04].

Linux, much like the GNU movement, encompassed not just one person but hundreds (and today thousands) of developers. While Torvalds remains the gatekeeper of Linux, the scope of this monolithic kernel has grown well beyond the scope of one person.

From Figure 1.2, it's important to note why the released minor version numbers are all even. The even minor number represents a stable release, and odd minors represent development versions. Because development releases are usually unstable, it's a good idea to avoid them for production use.

Bringing It Together

The rest, as they say, is history. GNU/Linux moved from an i386 single-CPU operating system to a multiprocessor operating system supporting many processor architectures. Today, GNU/Linux can be found in large supercomputers and small handheld devices. It runs on the x86 family, ARM, PowerPC, Hitachi SuperH, 68K, and many others. This is an important attribute for GNU/Linux as its adoption in the embedded community continues.

GNU/Linux in recent years has become the trailblazer for important new technologies like virtualization. In the old days (1960s IBM mainframe days), special operating systems called *hypervisors* were created to serve as an operating system for other operating systems. Today, Linux has become the hypervisor with projects like Xen and the Kernel Virtual Machine (KVM). In Chapter 4, "Linux Virtualization and Emulation," we'll learn more about virtualization and emulation technologies that are built on Linux.

GNU/Linux has evolved from its humble beginnings to be one of the most scalable, secure, reliable, and highest performing operating systems available. GNU/Linux, when compared to Windows, is less likely to be exploited by hackers [NewsForge04]. When you consider Web servers, the open source Apache HTTP server is far less likely to be hacked than Microsoft's IIS [Wheeler04].

Linux Distributions

In the early days, running a GNU/Linux system was anything but simple. Users sometimes had to modify the kernel and drivers to get the operating system to boot. Today, GNU/Linux distributions provide a simple way to load the operating system and selectively load the plethora of tools and applications. Given the dynamic nature of the kernel with loadable modules, it's simple to configure the operating system dynamically and automatically to take advantage of the peripherals that are available. Projects such as Debian [Debian04] and companies such as Red Hat [RedHat04] and Suse [Suse04] introduced distributions that contained the GNU/Linux operating system and precompiled programs on which to use it. Most distributions typically include over 10,000 packages (applications) with the kernel, making it easy to get what you need. If you don't find what you need, package management systems (like apt) can be used to easily update your OS with the software you need. We'll explore package managers and administration in Chapter 28, "GNU/Linux Administration Basics."

Summary

The history of GNU/Linux is an interesting one because at three levels, it's a story of frustration. Thompson and Ritchie designed the original UNIX as a way to replace the existing Multics operating system. Richard Stallman created the GNU and FSF as a way to create a free operating system that anyone could use, free of proprietary licenses. Linus Torvalds created Linux out of frustration with the Minix [Minix04] operating system that was used primarily as an educational tool at the time. Whatever their motivations, they and countless others around the world succeeded in ways that no one at the time would have ever believed. GNU/Linux today competes with commercial operating systems and offers a real and useful alternative. GNU/Linux is predominantly the operating system for other operating systems (speaking virtually). Even in the embedded systems domain, Linux has begun to dominate and operates in the smallest devices, including smartphones.

References

[Byte94] "Unix at 25" at http://www.byte.com/art/9410/sec8/art3.htm.

[Debian04] Debian Linux at http://www.debian.org.

[Linux/GNU04] "Linux and the GNU Project" at http://www.gnu.org/gnu/linux-and-gnu.html.

[Minix04] Minix Operating System at http://www.minix3.org.

[Montague03] "Why You Should Use a BSD-Style License," Bruce R. Montague, at http://63.249.85.132/open_source_license.htm.

[NewsForge04] "Linux and Windows Security Compared," Stacey Quandt, at http://os.newsforge.com/os/04/05/18/1715247.shtml.

[RedHat04] Red Hat at http://www.redhat.com and http://fedora.redhat.com.

[RobotWisdom02] "Timeline of GNU/Linux and UNIX" at http://www.robotwisdom.com/linux/timeline.html.

[Stallman83] "Initial GNU Announcement" at http://www.gnu.org/gnu/initial-announcement.html.

[Stallman02] "Free as in Freedom," Richard Stallman, O'Reilly & Associates, Inc., 2002.

[Suse04] Suse Linux at http://www.suse.com.

[Unix/Linux History04] "History of UNIX and Linux" at http://www.computerhope.com/history/unix.htm.

[Unix History94] "Unix History" at http://www.english.uga.edu/hc/unixhistoryrev.html.

[Wheeler04] "Why Open Source Software/Free Software," David A. Wheeler, at http://www.dwheeler.com/oss_fs_why.html.

[Wikipedia04] Timeline of Linux Development at http://en.wikipedia.org/wiki/Timeline_of_Linux_development.

2 GNU/Linux Architecture

In This Chapter

- High-Level Architecture
- Architectural Breakdown of Major Kernel Components

Introduction

The GNU/Linux operating system is organized into a number of layers. While understanding the internals of the kernel isn't necessary for application development, knowing how the operating system is organized is important. This chapter looks at the composition of GNU/Linux starting at a very high level and then works its way through the layers.

High-Level Architecture

First, take a high-level look at the GNU/Linux architecture. Figure 2.1 shows the 20,000-foot view of the organization of the GNU/Linux operating system. At the core is the Linux kernel, which mediates access to the underlying hardware resources such as memory, the CPU via the scheduler, and peripherals. The shell (of which there are many different types) provides user access to the kernel. The shell provides command interpretation and the means to load user applications and execute them. Finally, applications are shown that make up the bulk of the GNU/Linux operating system. These applications provide the useful functions for the operating system, such as windowing systems, Web browsers, e-mail programs, language interpreters, and of course, programming and development tools.

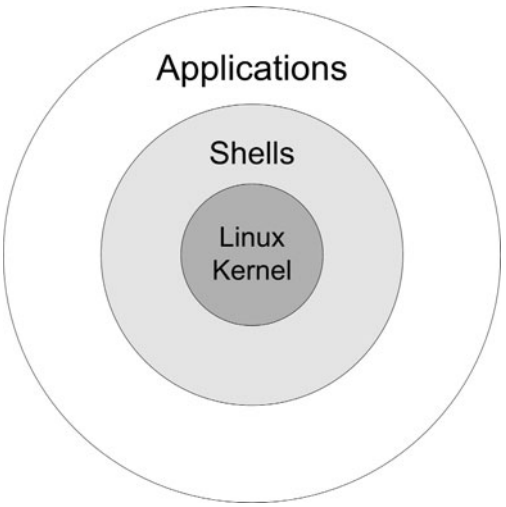

FIGURE 2.1 High-level view of the GNU/Linux operating system.

Within the kernel, you also have the variety of hardware drivers that simplify access to the peripherals (such as the CPU for configuration). Drivers to access the peripherals such as the serial port, display adapter, and network adapter are also found here.

This is a simplistic view, but the next section digs in a little deeper to help you understand the makeup of the Linux kernel.

Linux Kernel Architecture

The GNU/Linux operating system has a layered architecture. The Linux kernel is monolithic and layered also, but with fewer restrictions (dependencies can exist between noncontiguous layers). Figure 2.2 provides one perspective of the GNU/Linux operating system with emphasis on the Linux kernel.

Note here that the operating system has been split into two software sections. At the top is the user space (where you find the tools and applications as well as the GNU C library), and at the bottom is the kernel space where you find the various kernel components. This division also represents address space differences that are important to note. Each process in the user space has its own independent memory region that is not shared. The kernel operates in its own address space, but all elements of the kernel share the space. Therefore, if a component of the kernel makes

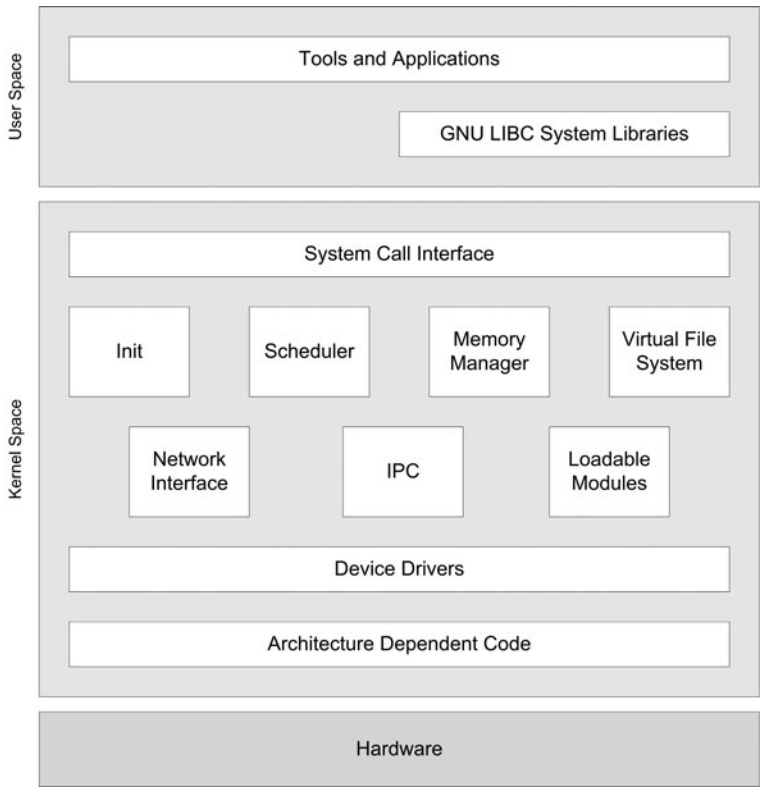

FIGURE 2.2 GNU/Linux operating system architecture.

a bad memory reference, the entire kernel crashes (also known as a *kernel panic*). Finally, the hardware element at the bottom operates in the physical address space (which is mapped to virtual addresses in the kernel).

The rest of this section now looks at each of the elements of the Linux kernel to identify what they do and what capabilities they provide to you as application developers.

GNU/Linux is primarily a monolithic operating system in that the kernel is a single entity. This differs from microkernel operating systems that run a tiny kernel with separate processes (usually running outside of the kernel) providing the capabilities such as networking, filesystem, and memory management. Many microkernel operating systems exist today, including CMU's Mach, Apple's Darwin, Minix, BeOS, Next, QNX/Neutrino, and many others. Which is better is certainly hotly debated, but microkernel architectures have shown themselves to be dynamic and flexible. In fact, GNU/Linux has adopted some microkernel-like features with its loadable kernel module feature.

GNU System Libraries (glibc)

The glibc is a portable library that implements the standard C library functions, including the top half of system calls. An application links with the GNU C library to access common functions in addition to accessing the internals of the Linux kernel. The glibc implements a number of interfaces that are specified in header files. For example, the `stdio.h` header file defines many standard I/O functions (such as `fopen` and `printf`) and also the standard streams that all processes are given (`stdin`, `stdout`, and `stderr`).

When building applications, the GNU compiler automatically resolves symbols to the GNU libc (if possible), which are resolved at runtime using dynamic linking of the libc shared object.

In embedded systems development, use of the standard C libraries can sometimes be problematic. The GCC permits disabling the behavior of automatically resolving symbols to the standard C library by using `-nostdlib`. This permits a developer to rewrite the functions that were used in the standard C library to his own versions.

When a system call is made, a special set of actions occurs to transfer control between the user space (where the application runs) and the kernel space (where the system call is implemented).

System Call Interface

When an application calls a function like `fopen`, it is calling a privileged system call that is implemented in the kernel. The standard C library (glibc) provides a hook to go from the user space call to the kernel where the function is provided. Because this is a useful element to know, let's dig into it further.

A typical system call results in the call of a macro in user space (special assembly sequences to map library calls to system call identifiers). The arguments for the system call are loaded into registers, and a system trap is performed, though newer virtualization processors provide a different means that take advantage of virtualization instructions. This interrupt causes control to pass from the user space to the kernel space where the actual system call is available (vectored through a table called `sys_call_table`). After the call has been performed in the kernel, return to user space is provided by a function called `_ret_from_sys_call`. Registers are loaded properly for a proper stack frame in user space.

In cases where more than scalar arguments are used (such as pointers to storage), copies are performed to migrate the data from user space to kernel space.

The source code for the system calls can be found in the kernel source at `./linux/kernel/sys.c`.

Kernel Components

The kernel mediates access to the system resources (such as interfaces, the CPU, and so on). It also enforces the security of the system and protects users from one another. The kernel is made up of a number of major components, which we'll discuss here.

init

The `init` component is performed upon boot of the Linux kernel. It provides the primary entry point for the kernel in a function called `start_kernel`. This function is very architecture dependent because different processor architectures have different `init` requirements. The `init` also parses and acts upon any options that are passed to the kernel.

After performing hardware and kernel component initialization, the `init` component opens the initial console (`/dev/console`) and starts up the `init` process. This process is the mother of all processes within GNU/Linux and has no parent (unlike all other processes, which have a parent process). After `init` has started, the control over system initialization is performed outside of the kernel proper.

The kernel `init` component can be found in `linux/init` in the Linux kernel source distribution.

Process Scheduler

The Linux kernel provides a preemptible scheduler to manage the processes running in a system. This means that the scheduler permits a process to execute for some duration (an epoch), and if the process has not given up the CPU (by making a system call or calling a function that awaits some resource), then the scheduler temporarily halts the process and schedules another one.

The scheduler can be controlled, for example, by manipulating process priority or chaining the scheduling policy (such as FIFO or round-robin scheduling). The time quantum (or epoch) assigned to processes for their execution can also be manipulated. The timeout used for process scheduling is based upon a variable called jiffies. A *jiffy* is a packet of kernel time that is calculated at `init` based upon the speed of the CPU.

The source for the scheduler (and other core kernel modules such as process control and kernel module support) can be found in `linux/kernel` in the Linux kernel source distribution.

New in the 2.6 Linux kernel is a scheduler that operates in constant time regardless of the number of processes to be scheduled. This new scheduler, called the O(1) scheduler because of its linear time complexity, is ideal for systems with large numbers of tasks.

Memory Manager

The memory manager within Linux is one of the most important core parts of the kernel. It provides physical-to-virtual memory mapping functions (and vice versa) as well as paging and swapping to a physical disk. Because the memory management aspects of Linux are processor dependent, the memory manager works with architecture-dependent code to access the machine's physical memory.

While the kernel maintains its own virtual address space, each process in user space has its own virtual address space that is individual and unique.

The memory manager also provides a swap daemon that implements a demand paging system with a least-recently used replacement policy.

The memory manager component can be found in `linux/mm` of the Linux kernel source distribution.

Elements of user-space memory management are discussed in Chapter 18, "Shared Memory Programming," and Chapter 20, "Other Application Development Topics," of this book.

Virtual File System

The Virtual File System (VFS) is an abstract layer within the Linux kernel that presents a common view of differing filesystems to upper-layer software. Linux supports a large number of individual filesystems, such as ext2, Minix, NFS, and Reiser. Rather than present each of these as a unique filesystem, Linux provides a layer into which filesystems can plug their common functions (such as `open`, `close`, `read`, `write`, `select`, and so on). Therefore, if you needed to open a file on a Reiser journaling filesystem, you could use the same common function `open` as you would on any other filesystem.

The VFS also interfaces to the device drivers to mediate how the data is written to the media. The abstraction here is also useful because it doesn't matter what kind of hard disk (or other media) is present; the VFS presents a common view and therefore simplifies the development of new filesystems. Figure 2.3 illustrates this concept. In fact, multiple filesystems can be present (*mounted*) simultaneously.

The Virtual File System component can be found in `linux/fs` in the Linux kernel source distribution. You also find there a number of subdirectories representing the individual filesystems. For example, `linux/fs/ext3` provides the source for the third extended filesystem.

GNU/Linux provides a variety of filesystems, and each of these provides characteristics that can be used in different scenarios. For example, xfs is very good for streaming very large files (such as audio and video), and Reiser is good at handling large numbers of very small files (< 1 KB). Filesystem characteristics influence performance, and therefore you need to select the filesystem that makes the most sense for your particular application.

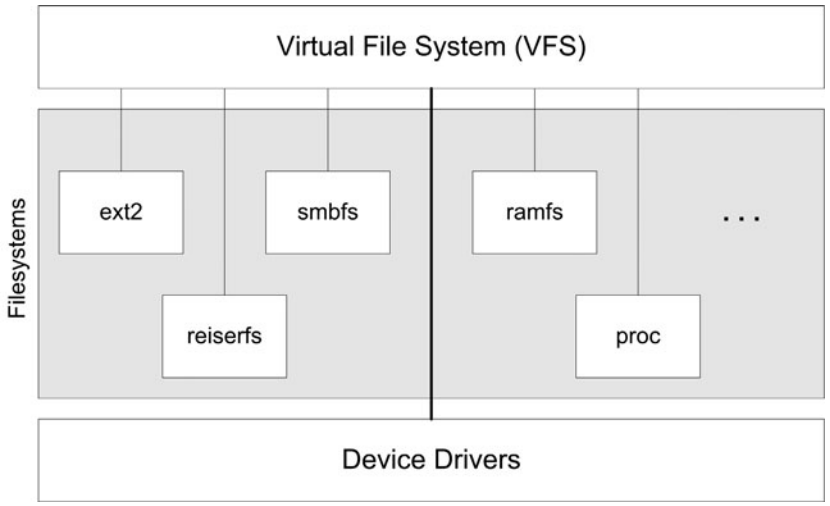

FIGURE 2.3 Abstraction provided by the virtual file system.

The topic of file I/O is discussed in Chapter 11 of this book, "File Handling in GNU/Linux."

Network Interface

The Linux network interface offers a very similar architecture to what you saw with the VFS. The network interface component is made up of three layers that work to abstract the details of networking to higher layers, while presenting a common interface regardless of the underlying protocol or physical medium (see Figure 2.4).

Common interfaces are presented to network protocols and network devices so that the protocol and physical device can be interchanged based upon the actual configuration of the system. As was the case with the VFS, flexibility was a design key.

The network component also provides packet scheduler services for quality of service requirements.

The network interface component can be found in `linux/net` of the Linux kernel source distribution.

The topic of network programming using the BSD Sockets API is discussed in Chapter 13, "Introduction to Sockets Programming," of this book.

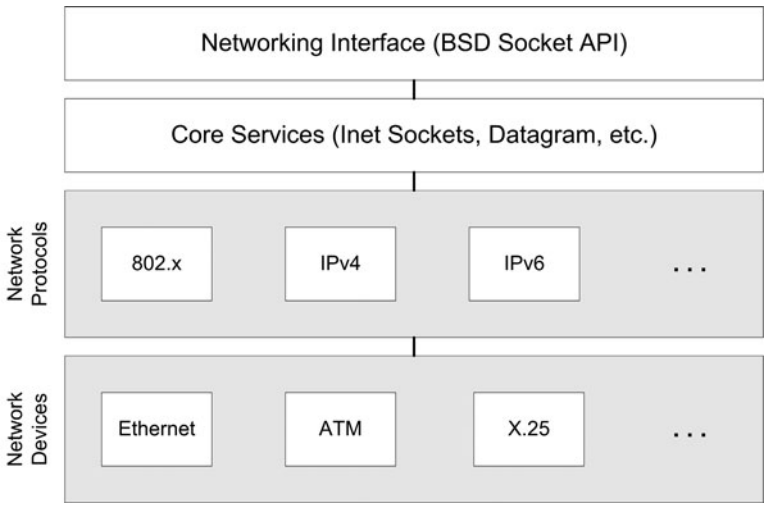

FIGURE 2.4 Network subsystem hierarchy.

Interprocess Communication (IPC)

The IPC component provides the standard System V IPC facilities. This includes semaphores, message queues, and shared memory. Like VFS and the network component, the IPC elements all share a common interface.

The IPC component can be found in `linux/ipc` of the Linux kernel source distribution.

The topic of IPC is detailed within this book. In Chapter 17, "Synchronization with Semaphores," semaphores and the semaphore API are discussed. Chapter 16, "IPC with Message Queues," discusses the message queue API, and Chapter 18 details the shared memory API.

Loadable Modules

Loadable kernel modules are an important element of GNU/Linux as they provide the means to change the kernel dynamically. The footprint for the kernel can therefore be very small, with required modules dynamically loaded as needed. Outside of new drivers, the kernel module component can also be used to extend the Linux kernel with new functionality.

Linux kernel modules are specially compiled with functions for module `init` and cleanup. When installed into the kernel (using the `insmod` tool), the necessary symbols are resolved at runtime in the kernel's address space and connected to the running kernel. Modules can also be removed from the kernel using the `rmmod` tool (and listed using the `lsmod` tool).

Because using loadable modules involves no performance disadvantage, they should be used when possible. Not using loadable modules results in a larger kernel packed with the various drivers that might not even be used.

The source for the kernel side of loadable modules is provided in `linux/kernel` in the Linux kernel source distribution.

Device Drivers

The device drivers component provides the plethora of device drivers that are available. In fact, almost half of the Linux kernel source files are devoted to device drivers. This isn't surprising, given the large number of hardware devices out there, but it does give you a good indication of how much Linux supports.

The source code for the device drivers is provided in `linux/drivers` in the Linux kernel source distribution. The vast majority of source in the Linux kernel exists within drivers. You can find drivers for special devices and also numerous drivers for common devices using different hardware.

Architecture-Dependent Code

At the lowest layer in the kernel stack is architecture-dependent code. Given the variety of hardware platforms that are supported by the Linux kernel, source for the architecture families and processors can be found here. Within the architecture family are common boot support files and other elements that are specific to the given processor family—for example, hardware interfaces such as direct memory access (DMA), memory interfaces for memory management unit (MMU) setup, interrupt handling, and so on. The architecture code also provides board-specific source for popular board vendors.

Hardware

While not part of the Linux kernel, the hardware element (shown at the bottom of the original high-level view of GNU/Linux, Figure 2.1) is important to discuss given the number of processors and processor families that are supported. Today you can find Linux kernels that run on single-address space architectures and those that support an MMU. Processor families including Arm, PowerPC, Intel × 86 (including AMD, Cyrix, and VIA variants), MIPS, Motorola 68K, Sparc, and many others. The Linux kernel can also be found on Microsoft's Xbox (Pentium III), Sega's Dreamcast (Hitachi SuperH), and even the new Cell processor-based PlayStation 3. The kernel source provides information on these supported elements.

The source for the processor and vendor board variants is provided in `linux/arch` in the Linux kernel source distribution.

Summary

The Linux kernel is the core of the GNU/Linux operating system. It is monolithic in nature and defined in a number of layers segregating its necessary elements. The kernel has been designed in such a way that adding new device drivers or protocols is simple, given the uniform interfaces that are available. This chapter provided a very high-level look at the architecture, with discussion of the major elements. References to Linux kernel source were provided where applicable.

Resources

Linux Kernel Archive at http://www.kernel.org.
Linux Kernel Cross-Reference at http://lxr.linux.no/.

3 Free Software Development

In This Chapter

- The Free Software/Open Source Development Paradigm
- Open Source versus Free Software
- Free and Open Software Licenses
- Problems with Free Software

Introduction

Whereas many consider the concept of free software (or open source) something that surfaced with the GNU/Linux operating system, it can actually be traced back to early use in universities and research labs where source was released for others to use, modify, and hopefully improve. The goal of open source is to make source code available to others so that they can identify bugs, create new features, and generally evolve the software. Free software can promote greater reliability and quality through increased use of the software, in addition to greater visibility into how it works.

This chapter discusses the free software development models and introduces some of the licenses that are used to release open source. To keep it fair and balanced, it also discusses some of the common problems with free software.

Open Source versus Free Software

Before discussing free software, this chapter first covers one of the many religious debates that exist. *Open source* was a term coined by Eric Raymond with the creation of the Open Source Initiative (OSI) in 1997. The term *free software* was coined

by Richard Stallman with the release of the GNU Project in 1984 and the founding of the Free Software Foundation in 1985.

While the two terms appear to be similar, and most open source software is released under the GPL (also created by Stallman), the debate is over the motivation for the release of open software. Richard Stallman defines it best: "Open source is a development methodology; free software is a social movement" [Stallman04]. Raymond has also been criticized by many for hijacking the free software movement for his own self-promotion (13 years after it was originally created) [Raymond04].

Anatomy of a Free Software Project

Free software (including open source) is simply a set of useful source code that is licensed under a free software license such as the GNU General Public License (or GPL). Popular Web sites such as SourceForge and Freshmeat provide a means to make free software available to the Internet community. In addition to providing a means for others to find free software, these sites also serve as a meeting place for free software developers. Developers can create new free software projects or join existing projects. This is the essence of free software: developers coming together to build software that is both useful and free to the wider community.

The fact that source code is available means that if something doesn't work the way it should, it can be modified to suit the needs of others. The availability of source also solves the myth of proprietary software, called "security through obscurity." Companies believe that because their software isn't provided in source form, it's more secure because it can't be opened to identify exploits. In fact, what happens in free software is that because it's open, exploits are found and fixed more quickly, making them less likely to be exploited as the distribution of the software widens. Open source is also more likely to uphold higher quality standards because of its openness. If a developer knows that it can (and will) be seen by many, more care will be taken during the coding process. Proprietary software is quickly proving that obscurity does not provide security.

As free software gains in popularity, so does the desire of others who want to help. Free software gains not only in development support but also in documentation, testing, and advertising (typically word of mouth).

Free software has gained so much popularity that even large companies contribute source code to the community. In 2003, IBM donated source code under the Common Public License (in addition to $40 million) to the Eclipse Consortium to help in the development of the Visual Editor Project [zdnet.com03]. IBM has been very supportive of open source and has stated that it is one of the key factors fueling software discovery and innovation around the world [ibm.com04].

Numerous industries have spawned from open source software, not only for companies that produce it, but also those that help to advertise and propagate it. Two such companies act as websites that store open source software and make that software available for download (SourceForge and Freshmeat). SourceForge, at the time of this writing, hosts over 150,000 projects and almost two million registered users. In fact, a fork of the web-based SourceForge site is available as open source itself in the project called GForge.

Open Source Licenses

Now it's time to take a quick look at the free and open source licenses. This section looks at a few different examples that provide different aspects of licensing (ranging from preventing commercial distribution to supporting it).

GPL

The GNU General Public License is one of the most popular licenses used in free software. The GPL provides the user with three basic "rights":

- The right to copy the software and give it away
- The right to change the software
- The right to access the source code

Within these rights is the catch with GPL software. Any changes that are made to GPL software are covered by the GPL and must be made available to others in source form. This makes the GPL unattractive from a commercial perspective because it means that if a company changes GPL software, that source code must be made available to everyone (including competitors). This is what's called the "tainting" effect of GPL software. What "touches" GPL software becomes GPL software (otherwise known as a derivative work, something derived from the GPL).

A variation of the GPL exists called the LGPL (Library GPL, or what is now called the Lesser GPL to indicate the loss of freedom). A library released under the LGPL makes it possible for proprietary software to be linked with the libraries without the tainting effect. For example, the GNU C library is released under the LGPL, allowing proprietary software to be developed on GNU/Linux systems.

Software built into the Linux kernel is automatically GPL, though differences of opinion exist in the case of kernel modules (which can be viewed both ways). The issue of kernel modules has yet to be challenged in court.

At the time of this writing, the latest version of the GPL (version 3) has been introduced with added protections for embedded use of Linux. In one case, a product that used an embedded version of GNU/Linux released source code (in compliance with the GPL), but when users attempted to load new software on the product based upon that released source code, the product detected this and conveniently shut down. [TiVO-isation06] This was against the "spirit" of the GPL, and additions were made to the GPLv3 to protect against this.

Qt Public License

The Qt Public License (QPL) is an oddity in the open source community because it breaks the openness created by other public licenses. The QPL permits two types of licenses: a free license and a commercial license. In the free version, any software that links to the Qt framework must be opened using either the QPL or GPL. Developers could instead purchase a commercial license for Qt, which allows them to build an application using the Qt framework and keep it closed (it's not required to be released to the open source community).

From the perspective of openness, "buying out" of the license makes the Qt framework less useful.

BSD

If one could identify a spectrum of licenses with the GPL on the left, the BSD license would exist on the right. The BSD license offers a more commercial friendly license because a program can be built with BSD-licensed software and not be required to then be BSD licensed itself. The BSD community encourages returning modified source code, but it's not required. Despite this, the BSD UNIX operating system is as advanced, if not more so, as the GNU/Linux operating system.

The issue of the BSD license is what's called *forking* (two or more versions of source code existing from a single source distribution). A commercial incentive exists to fork rather than make your hard work available to your competitors. The BSD UNIX operating system has itself been forked into a number of variants, including FreeBSD, NetBSD, and OpenBSD.

The lack of distribution restrictions defines the primary difference between the BSD and GPL. GPL specifies that if one uses the GPL in a program, then that program becomes a derivative work and therefore is GPL itself. BSD has no concept of a derivative work, and developers are free to modify and keep their changes.

License Summary

Many licenses exist and can be viewed at the Open Source Initiative or in reference [gnu.org04], which also identifies their relation to the GPL and free software.

How one defines freedom determines how one views these licenses. If freedom means access to source code and access to source code that uses the original code, then the GPL is the license to use. If freedom is viewed from a commercial perspective (the freedom to build a product without distributing any changes to the source base), then BSD is a fantastic choice. From our short list, the QPL tries to straddle both extremes. If a commercial license is purchased, then the application using the Qt can be distributed without source. Otherwise, without a commercial license (using the so-called free license), source code must be made available.

You are encouraged to read the available references and license resources discussed at the end of this section. Like anything legal, nothing is really black and white, and a careful review is suggested.

Problems with Open Source Development

It wouldn't be fair to discuss the open source development paradigm without mentioning any of the problems that exist. Open source is a wonderful development paradigm, but it does suffer from many of the same problems as proprietary software development.

Usability/Reliability Ramp

The early days of the GNU/Linux operating system were not for the faint of heart. Installing GNU/Linux was not a simple task, and commonly, source changes were necessary to get a system working. The operating system, after all, was simply a hobby in the early days and did not have the plethora of drivers and hardware support that exists today. New open source projects mirror some of these same issues. Early adoption can be problematic if you are not willing to get your hands dirty. But if an application is truly of interest, just wait a few releases, and someone will make it more usable.

For a demonstration of the simplicity of building the Linux kernel, check out the kernel building section in Chapter 28, "GNU/Linux Administration Basics."

Documentation

Documentation on any software project is one of the last elements to be done. Open source is no different in that respect. Some have claimed that the only real way to make money from open source is to sell documentation (such as what the Free Software Foundation does today, though it's also freely downloadable).

Ego

Like proprietary software, ego plays a large part in the architecture and development direction. Ego is a key reason for the failure of many open source projects, probably more than technical failings, but this is a personal opinion. Related to ego are the conflicts that can arise in open source development. One developer sees an application moving in one direction, while another sees a different path. A common result is forking of an application, which in itself can be beneficial if viewed from the perspective of natural selection.

Fanaticism

The open source movement is filled with fanatically committed people. Phrases such as "GNU zealot" and "Linux zealot" are not uncommon from those on the "outside." Arguments over, for example, which operating system is better mirror many of the political debates to which you've become accustomed. The danger of fanaticism is that you don't see the real issues and focus on what's really important. You can use both Windows and Linux and lead a full and productive life. It's not an either/or argument; it's more about the best tool for the job.

Many of the deeply fervent debates on open source result in arguments such as "open source is better, just because it is." The argument becomes a disjunctive syllogism, such as "Windows or GNU/Linux; definitely not Windows, therefore GNU/Linux." You could argue the merits of the vi editor over Emacs (or vice versa), but ultimately what's most important is that the editor does what you need it to do. Can you operate efficiently using it? From personal experience, I know engineers are aghast that someone would use such an editor as vi. But if you can edit as efficiently in vi as you can in any other editor, why not? Personal preference definitely plays a part in the use of open source software.

Summary

The Free Software movement and open source community have changed the way that people look at pro bono software development. The GNU/Linux operating system, the Apache Web server, and the Python object-oriented scripting language (just to name a few) have resulted from distributed and sometimes ad hoc development around the world. Free and open software licenses have been created to protect free software and maintain it as free, but differences exist depending upon the goal. However, free software development isn't a panacea to today's software development issues, as it does suffer from the same issues of proprietary software development.

References

[ibm.com04] "Innovation Thriving, Depends on Openness to Continue," IBM, 2004.
[gnu.com04] "Various Licenses and Comments about Them" at http://www.gnu.org/philosophy/license-list.html
[Raymond04] Wikipedia: "Eric S. Raymond" at http://en.wikipedia.org/wiki/Eric_Raymond
[Stallman04] "Why 'Free Software' Is Better Than 'Open Source'" at http://www.gnu.org/philosophy/free-software-for-freedom.html
[TiVO-isation06] "GPLv3 issues: TiVO-isation" at http://www.digital-rights.net/?p=548
[zdnet.com03] "IBM Donates Code to Open-Source Project" at http://zdnet.com.com/2100-1104_2-5108886.html

Resources

Developer.com at http://www.developer.com/open/.
Freshmeat at http://www.freshmeat.net.
GForge Content Management System at http://gforge.org/.
GNU Project License List at http://www.gnu.org/licenses/license-list.html.
Open Source Initiative (OSI) at http://www.opensource.org/.
Open Source Technology Group at http://www.ostg.com/.
SourceForge at http://www.sourceforge.net.
"Why Open Source Software/Free Software (OSS/FS)? Look at the Numbers!" at http://www.dwheeler.com/oss_fs_why.html.

This page intentionally left blank

4 Linux Virtualization and Emulation

In This Chapter

- Introduction to Virtualization
- Taxonomy of Virtualization
- Virtualization History
- Open Source Virtualization Methods

Linux is a great operating system, not just for production use, but also for the research and development of cutting-edge operating-system technologies. One of the biggest and most important areas today is that of virtualization. This chapter explores the ideas behind virtualization and then discusses the various means by which virtualization can be accomplished in GNU/Linux.

Introduction

At the time of this writing, virtualization could not be bigger. VMware is moving toward an IPO, Xen has been acquired by Citrix, and a virtualization solution called KVM has finally been integrated directly into Linux (via a loadable module). Linux is at the center of this new revolution and is, therefore, an important technology to understand and know how to exploit.

This chapter starts with a short history of virtualization, as it truly is a technology that is as old as modern computing itself. Then you'll review the options for Linux virtualization and explore how each can be used.

What Is Virtualization?

Virtualization is unfortunately one of those words that when used in isolation creates confusion. This chapter discusses what is called *platform virtualization*, which means that the bare-metal hardware is virtualized, allowing more than one operating system to run concurrently on it. The abstraction of the computer hardware is performed commonly through what is called a hypervisor, or Virtual Machine Monitor (VMM). The hypervisor creates an environment from which guest software (commonly an operating system) can be run (see Figure 4.1).

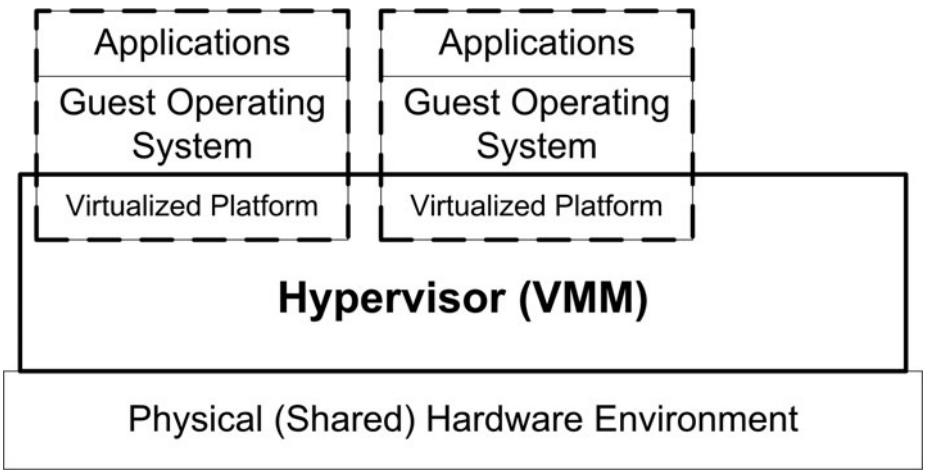

FIGURE 4.1 Graphical depiction of basic platform virtualization.

Note the similarities here between a hypervisor, which creates an environment for an operating system, and an operating system, which creates an environment for applications. From this perspective, and as you can see in the short history section, virtualization is certainly not a new technique, but instead a tried and true one with many applications.

Because the hypervisor abstracts the resources for each of the guest operating systems, the failure of one guest operating system does not mean that other guest operating systems fail. Each guest is isolated from the others.

This is one of the first advantages of virtualization. While a computer can be better utilized with multiple guests, their isolation means that one's failing does not impact another.

Short History of Virtualization

The history of virtualization is interesting and worthy of more study than is covered here. The term *virtualization* began being used in the 1960s with IBM's CP-40 research operating system (the starting point to the popular System/360 family of mainframe computers). The CP-40 (see Figure 4.2) provided the ability to run multiple instances of an operating system (in the early days, this was the Cambridge Monitor System, or CMS).

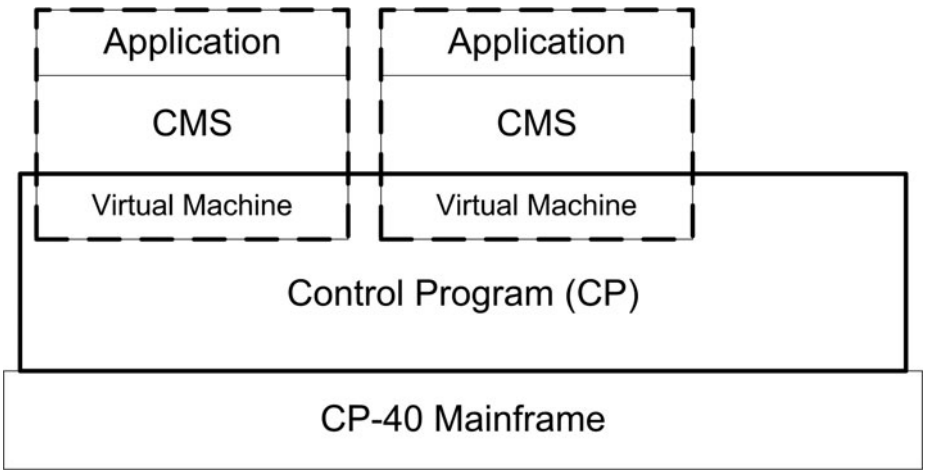

FIGURE 4.2 Early virtualization with IBM's CP-40.

Note the similarities this system has to the basic virtualization platform in Figure 4.1. The Control Program was the first hypervisor, with virtual machines implementing the fully virtualized platforms for isolating separate operating systems.

The CP-40 implemented multiple forms of virtualization that at this time were purely research oriented. The first, as shown in Figure 4.2, was the concept of a virtual machine. In those days, this was called *time-sharing* because the system was shared by multiple users who were each unaware of the others' existence on the CP-40 platform. Up to 14 individual virtual machines could be supported on a single CP.

Another innovation in the CP-40 (demonstrated also in the University of Manchester's Atlas computer and IBM's experimental M44/44X) was the concept of paging. Paging allows memory to be paged in and out of the available physical memory, allowing a much larger address space to be represented than is physically available. Paging transparently moves blocks of memory (called pages) between

central (local) storage and auxiliary (non-local) storage. Local storage is addressed as virtual memory, abstracting the physical memory of the system. This allows a page of virtual memory to exist in physical memory or auxiliary memory. Pages are moved out of local memory based upon a replacement algorithm (commonly least-recently used).

One of the most popular virtualization platforms is IBM's System/370 mainframe. While you might not have access to this platform, it might interest you to know that you can virtualize an entire System/370 mainframe or even the latest 64-bit z/Architecture on Linux with the open source Hercules emulator.

While Linux is the focus of this book, it's useful at times to run other operating systems (such as Darwin, ReactOS, or BeOS). Virtualization allows two or more different operating systems to coexist on the same host at the same time (see Figure 4.3).

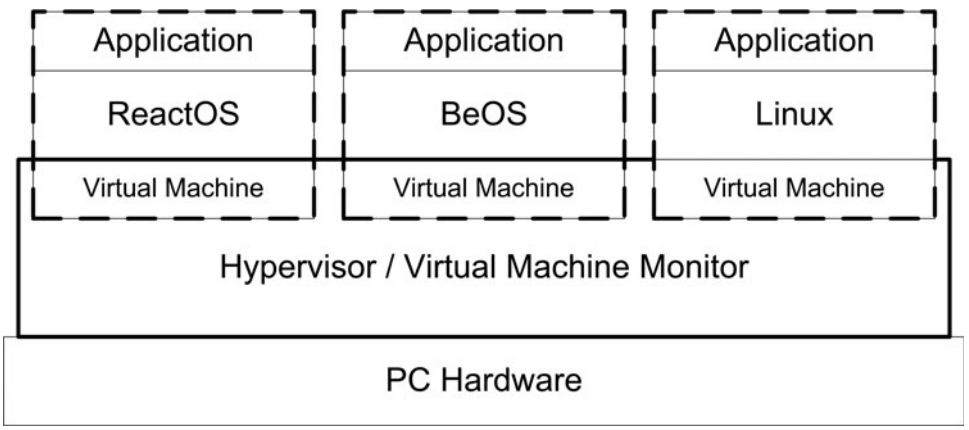

FIGURE 4.3 Virtualization can support multiple operating systems.

In addition to CPU and memory, virtualization has also been applied to other computing system objects. For example, storage systems have been (and are currently) virtualized. While IBM introduced the concepts in a 1978 patent, the term Redundant Array of Inexpensive Disks (RAID) was not coined until 1987 by the University of California. Conceptually, RAID is the virtualization of disks to abstract them into other entities. For example, two disks can be virtualized into one through mirroring, essentially keeping two copies of storage on two disks for reliability (if one disk fails, the data is still available on the other). Multiple disks can also

be viewed as multiple logical disks through striping, which increases performance through parallel access to multiple independent disks (see Figure 4.4).

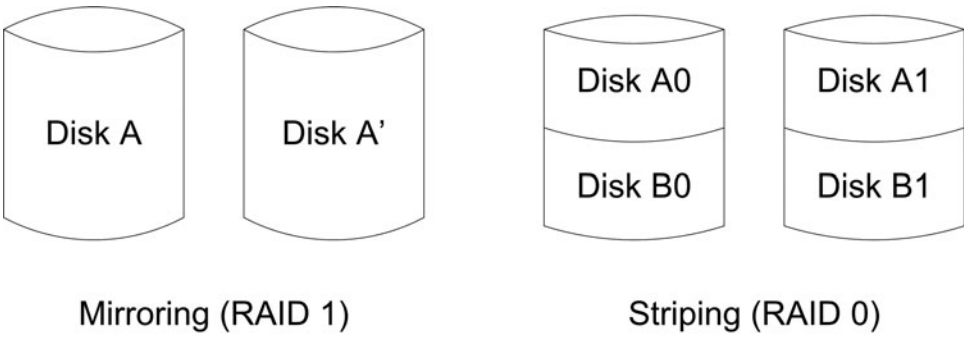

FIGURE 4.4 Disk virtualization with RAID levels.

Other virtualization possibilities exist such as breaking a single physical volume into multiple logical disks or, vice versa, taking multiple physical disks and making them appear as one large volume.

Linux is a great operating system on which to try virtualization technologies. In addition to a variety of solutions for platform virtualization, you also find standard solutions for storage virtualization (supporting all useful RAID levels).

Today, virtualization is the new big thing. With numerous methods to achieve platform virtualization, Linux supports an approach that meets any need. In the next section, you explore the methods by which virtualization is implemented and then review and see demonstrated some of the options for Linux.

What's the Point?

Modern virtualization is important for many reasons. From a developer's perspective, virtualization allows you to develop within a virtualized kernel. If your code causes the kernel to crash, it simply causes that virtualized platform to crash, but your machine (and host operating system) continues to run. It can also be simpler to debug kernels that are part of a virtualized platform, as you have greater control over that environment.

Virtualization also allows the development of software for other architectures in a simpler fashion. For example, it's not always necessary for the software that's being virtualized to be of the same instruction set as the host. You could, for example, virtualize a Linux kernel for a PowerPC on an x86 host.

But the real value of virtualization is as old as its first use in multi-user systems. Servers in the enterprise used to support a small number of applications, but in most cases, the server was dormant much of the time. Servers can be better utilized if more applications (and operating systems) can run on them. Platform virtualization makes this possible, with added reliability. If a virtual server crashes, no other virtual server goes down with it. Therefore, the servers are better utilized, and fewer servers are required, meaning less expense.

Another interesting advantage of virtualization is migration. A virtualized server can be stopped, migrated to a new physical server, and then restarted. This allows a particular server to be removed from the working set (for maintenance) while the application remains running (with a small delay for migration). Related to migration is checkpointing. When Linux is migrated, its state is stored in its entirety in a file. This file can then be migrated, or can be saved to represent the OS, applications, and all state at that point in time.

Virtualization Taxonomy

If there's one thing that GNU/Linux is known for, it's options. If you need to build an application using a scripting language, you have many choices. You need support for a particular network card, a driver exists, and in most cases, it simply works. Virtualization is no different. Linux supports a number of different virtualization methods, each with its own advantages and disadvantages. In this section, you can see the more popular virtualization approaches and then look at some of the options that are available to you.

Full Virtualization

Full virtualization refers to the method by which a virtual machine instance is created for a guest operating system to allow it to run unmodified. The virtual machine simulates some portion of the underlying hardware, trapping calls when necessary to arbitrate within the hypervisor (Virtual Machine Monitor). This form of virtualization was shown in Figure 4.1.

Full virtualization requires that all operating systems be targeted to the same processor architecture, which is a common constraint in all but experimental or developmental scenarios. Each operating system is unmodified and is unaware that it is executing on a virtualized platform. Further, full virtualization can run on hardware with virtualization support.

The biggest issue in virtualization is how to handle special privileged-mode instructions. These must be trapped appropriately and allowed to be handled by the hypervisor. So in an unmodified operating system (without virtualization hardware), these calls must be found and converted to hypervisor calls. This is called code scanning, and the result is a modification of the original code to insert hypervisor calls where privileged trap instructions were found.

Paravirtualization

The definition of paravirtualization has changed given recent announcements. The early method of paravirtualization required modifications to the guest operating system so that it was aware of the fact that it was being virtualized. This meant higher performance (as code scanning was not necessary) but limited the operating systems that could be virtualized (see Figure 4.5).

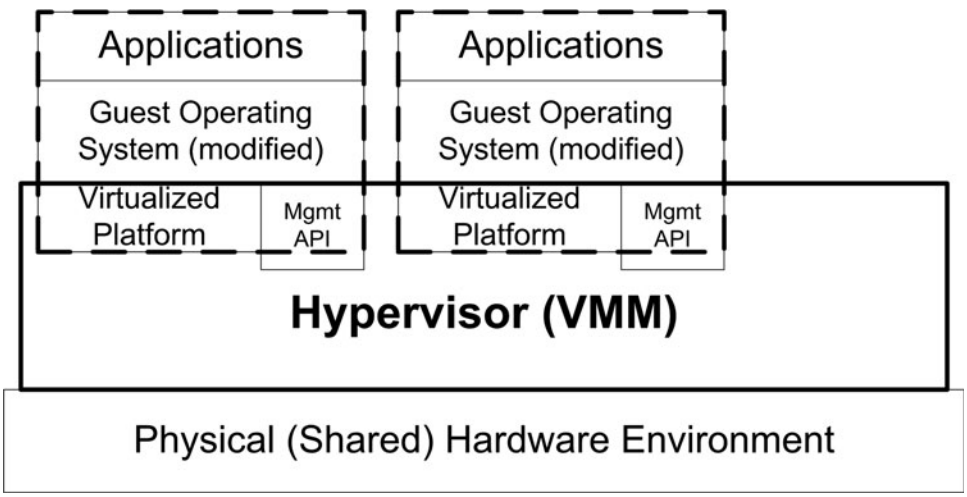

FIGURE 4.5 Early paravirtualization solution.

But today, the definition of paravirtualization has changed. Given hardware support for virtualization, operating system modifications are no longer necessary. This means that all operating systems can be virtualized because the virtualization is transparent to each guest operating system (see Figure 4.6).

As you will see shortly, Linux itself is the ideal hypervisor, making it possible to virtualize guest operating systems on top of Linux (including Linux guest operating systems).

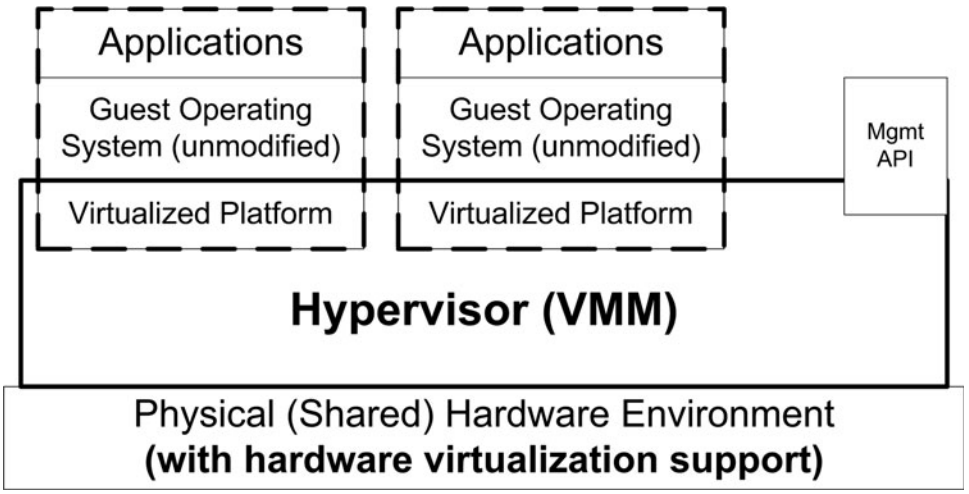

FIGURE 4.6 Current paravirtualization solution.

Emulation

An older technique, though still popular because of its versatility, is called emulation. With emulation, you have not just one hypervisor, but potentially many servicing each virtualized platform. The hypervisor emulates the entire platform, providing the means to emulate a processor different from that of the host. For example, with emulation, you can run a Linux kernel targeted for PowerPC on an x86 host.

Emulation is obviously not a new idea; in fact, it has been used since the early days of computing. Further, some other examples of emulation that are not specific to operating systems are useful to understand.

System Emulation

One fascinating example of a full system emulator is MAME, or Multiple Arcade Machine Emulator. This software emulates a large number of arcade machines, including not only their CPUs, video, and sound hardware, but also control devices such as joysticks and sliders. It's an amazing example of an emulator because of the vast number of processors and hardware configurations that are emulated.

The ROMs (or storage device for the games) are typically emulated as a file in the host operating system, which is a common method for any emulator (including the Linux virtualization solutions). As of this writing, over 3,600 individual games can be emulated on MAME.

Language VM

While not the first language to make use of a virtual machine, Java is the most popular (see Figure 4.7). Early examples of bytecode execution include the UCSD p-System, which was the portable execution environment for UCSD Pascal. Forth is another language that relied on emulation, typically on a simple stack-based machine.

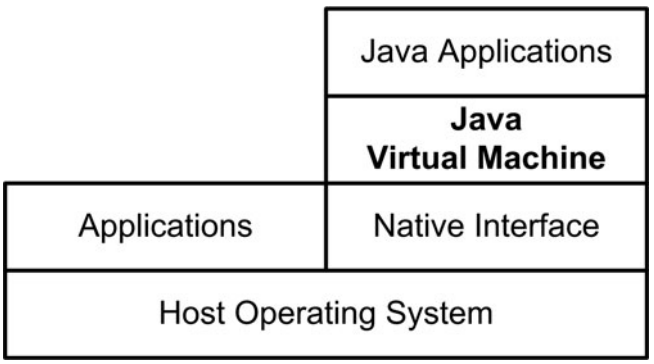

FIGURE 4.7 The Java Virtual Machine (JVM) as the virtualization environment for Java applications.

Java is emulated, but still has very good performance and great portability. Java and older Pascal implementations are not the only ones that are emulated. In fact, many of the modern scripting languages (such as Ruby or Python) are interpreted rather than being languages that execute directly on the host processor.

Specialized Emulators

One final example of an emulator is a popular one in embedded development. Prior to hardware being available, embedded developers commonly relied on instruction set simulators (ISS). These emulators/simulators allow code to be executed on simulated hardware for preverification. These simulators commonly support cycle accuracy, simulating the processor exactly and allowing performance evaluation to be performed.

Operating System Virtualization

One final virtualization method to be discussed is called operating system virtualization. This is different from the methods discussed so far because it's more about isolating services at the operating system layer than virtualizing two or more operating

systems on a given platform. For example, in OS virtualization, a single operating system exists with multiple user-space environments (see Figure 4.8).

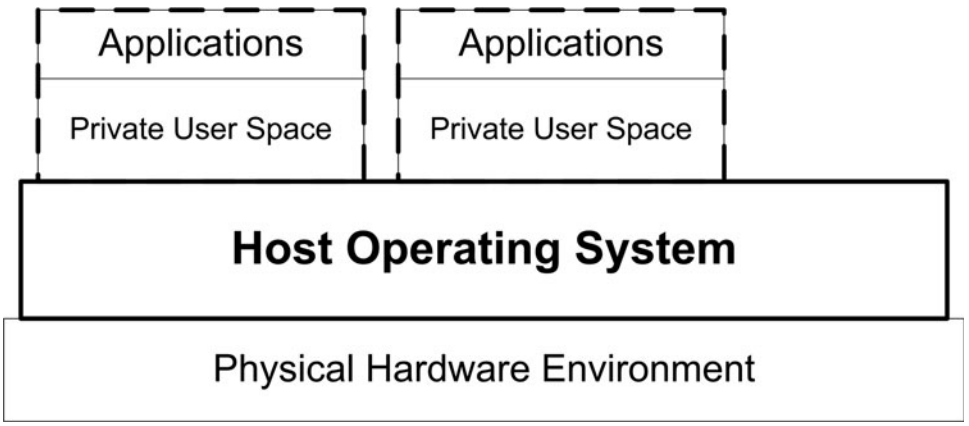

FIGURE 4.8 Operating system virtualization focuses on virtualizing user-space environments.

Operating system virtualization is very popular for server sharing by multiple users. It has the advantage of being fast (multiple kernels aren't virtualized), but is not as reliable as other virtualization schemes. For example, if one user space causes the system to crash, then all user spaces crash because they rely on the same kernel.

Hardware-Assisted Virtualization

Virtualization is such a popular technique and brings so many advantages to computing that processors are now being introduced with instruction sets that accelerate it. Modern CPUs support efficient ways of mediating between privileged modes of execution (to support user-mode and kernel-mode software, for example). In the same vein, new CPUs include instructions to mediate between guest operating systems and hypervisors (as hypervisors are more privileged than guest operating systems). Two CPU vendors are producing processors with virtualization support and are explored in this section.

Many of the virtualization techniques, such as full and paravirtualization schemes, take advantage of these CPUs. Emulators are also now incorporating support for virtualization instruction sets in search of better efficiency.

x86 Virtualization

Intel provides the VT-x (and VT-i) processors with extensions to support virtualization. AMD provides the AMD-V x86 architecture with extensions for virtualiza-

tion. Each provides roughly the same capabilities, so I'll speak generally about what's necessary for x86 virtualization and how it's achieved.

In x86 architectures, four rings represent privileged levels of execution. Ring 0 is the most privileged and is used to execute the operating system. Ring 3, the least privileged, is where user applications are run (with rings 1 and 2 rarely being used). Therefore, operating systems expect to be running in ring 0. Also, numerous instructions surface in the privileged state of the CPU. While these would be usable in a hypervisor, they should not be executed in guest operating systems. One solution is to provide rings for guest operating systems, and another is to alias the rings so that they can be swapped in and out based upon the virtual machine being run. The Intel CPU provides a set of rings for the hypervisor while another set of rings exists for the guest virtual machines.

Virtualization-aware CPUs also add new instructions to manage control between the hypervisor and guest operating systems. For example, the AMD-V processor introduces a new instruction called `VMRUN` that migrates the processor between virtualization modes. For example, the `VMRUN` instruction can migrate the CPU and its state from the Hypervisor to the guest operating system, and back. Another instruction called `VMMCALL` allows the hypervisor and a guest operating system to communicate directly. AMD also introduced the concept of application-space IDs, which provide control over TLBs (as the guest operating systems never touch the real page tables managed by the CPU).

As virtualization is an important technology, the ability to efficiently execute hypervisors and quickly transition between guest operating systems is critical. CPU vendors are now doing their part to improve this process.

You can tell if your CPU has virtualization support by checking `procinfo` *within the* `/proc` *filesystem. The flags line within* `proc-info` *should have either* `vmx` *if you have an Intel processor that's virtualization aware or* `svm` *if you have an AMD processor that's virtualization aware. You can easily check this with the following command line:*

```
$ grep svm /proc/cpuinfo
```

Open Source Virtualization Solutions

You have numerous open source solutions for virtualization on Linux. In this section, I'll show you how to use two. QEMU is more accessible, because you don't need special hardware to use it. KVM requires a platform with a virtualization-aware processor, so you'll need a bit more to use it.

QEMU

QEMU is a platform virtualization application that allows you to virtualize an operating system on top of Linux. QEMU virtualizes an entire PC system including peripherals such as the display, sound hardware, and NIC. QEMU also is equipped with an accelerator that speeds up PC emulation on x86 physical hosts. If your host is x86, it's very worthwhile to enable this.

How QEMU Works

QEMU is a full system emulator, which means that it creates (virtually) a PC within an application and then emulates the desired operating system on it (see Figure 4.9). QEMU offers a couple of ways to emulate the operating system, depending upon the instruction set being emulated. If the code is the same as the host machine, then the process of emulation is much simpler (because the instruction sets match). QEMU permits execution of the guest code directly on the CPU (without emulating it), making it more of a paravirtualization solution (but with an emulated PC environment).

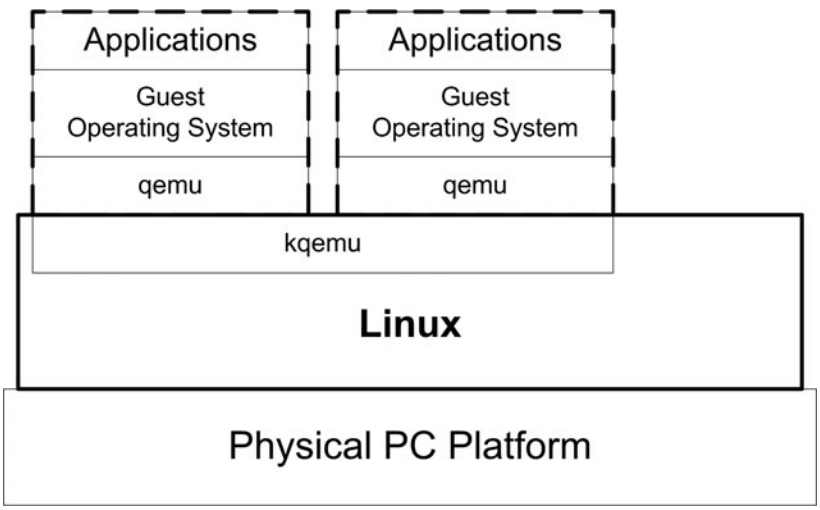

FIGURE 4.9 The QEMU approach to virtualization through emulation.

QEMU can also emulate other architectures on x86, for example. One example is emulating a Linux kernel built for PowerPC on an x86 host system. Emulators of this type are traditionally very slow, but QEMU emulates in a much more efficient way through a process called dynamic translation. In dynamic translation, translated code (code from the guest that's been translated to run on the host CPU) is

cached for use. This means that a block of translated code can be reused without retranslating. QEMU also takes a very smart approach to translation by employing compiler techniques within the translator to quickly and efficiently translate instruction sequences (called micro-operations) from one instruction set to another.

Now have a look at the process for emulating an operating system on Linux.

Installing and Emulating a Guest OS

The process of installing and emulating a guest OS on top of Linux is surprisingly simple. The process of building and installing won't be covered here, but it's the standard configure/make/make install procedure that exists for most Linux software.

The first step after building and installing QEMU is to install the QEMU accelerator. This is a kernel module that is inserted into the kernel using the `insmod` command:

```
$ insmod kqemu.ko
$
```

This permits much faster emulation of x86 code on an x86 host system.

The process of emulating an operating system begins with an installation. Just like installing a new operating system on a computer, you must first install your new operating system within the QEMU environment. To emulate an operating system, you begin with a hard disk onto which the guest operating system is installed. In virtualization, you use a shared hard disk, so your hard disk for QEMU is actually a file within the filesystem of the host operating system.

Creating this disk uses a special command in QEMU called `qemu-img`. This utility can be used to create images and also convert them between image types. QEMU supports a number of image types, but the most useful is called qcow (or QEMU Copy On Write). This is a compressed image type that consumes only what is used. For example, a 4 GB qcow image file can be created, but it consumes only 16 KB when empty. As files are added to the emulated disk (qcow file), it grows accordingly. The following creates a qcow image that's 256 MB in size:

```
$ qemu-img create qcow disk.img 256M
Formating 'disk.img', ftm=qcow, size=262144 kB
$
```

With the newly created hard disk image, the next step is to install the desired operating system. This happens very much like installing a new operating system on a standard PC, except that you control it from the command line. Most operating system images are loaded from a CD-ROM in what's called an ISO format. ISO (or ISO9660) is the standard format for CD-ROMs. ISO images for operating systems

can be downloaded from any Linux distribution website. But it doesn't have to be GNU/Linux; you could grab BSD images or any of the other operating systems that are available.

With the ISO image downloaded, the next step is to use QEMU to boot the CD-ROM image in order to install it to the emulated hard disk. This is performed with the following command line:

```
$ qemu -hda disk.img -cdrom image.iso -boot d
```

This command line specifies the hard disk image (that was previously created), the ISO file, and what to boot. The `-boot` option specifies where to boot, in this case `d` means the CD-ROM (`n` would be network, `a` the floppy, and `c` the hard disk). This command boots (emulates) the image on the emulated CD-ROM. The purpose of the CD-ROM is to install the operating system image onto the hard disk. QEMU emulates the entire platform, including the hard disk (represented a file in the host operating system) and how to deal with a simulated CD-ROM (in addition to other devices).

After this command is executed, the CD-ROM image is emulated and this results in the typical install process in a new window (which represents the emulated PC platform). When the install process is complete, the emulation platform ends. The emulated hard disk has now been populated (through the install process) with a bootable OS image. This can now be booted with QEMU, but this time you can specify that the hard disk should be used for the boot process:

```
$ qemu -hda disk.img -boot c
```

This creates a new window for the QEMU guest OS instance and proceeds to boot the newly installed operating system. This same process can be used to install any operating system, from Linux to Windows.

Booting from Emulated Floppy Disks

QEMU can also be used to boot smaller operating systems, such as those that reside on a single floppy disk. QEMU provides a special option to boot a floppy disk image. The example that follows illustrates QEMU booting a live floppy disk image (`fda`), specifying a floppy disk boot (`-boot a`), and disabling boot signature checking.

```
$ qemu -fda aros.bin -boot a -no-fd-bootchk
$
```

This example booted the AROS operating system, which is compatible with the AmigaOS 3.1 OS (at least at the API level). This is one of the greatest features of QEMU, the ability to try new operating systems without having to install them directly on your system. The AROS workbench screen is shown in Figure 4.10, as emulated on a standard GNU/Linux desktop.

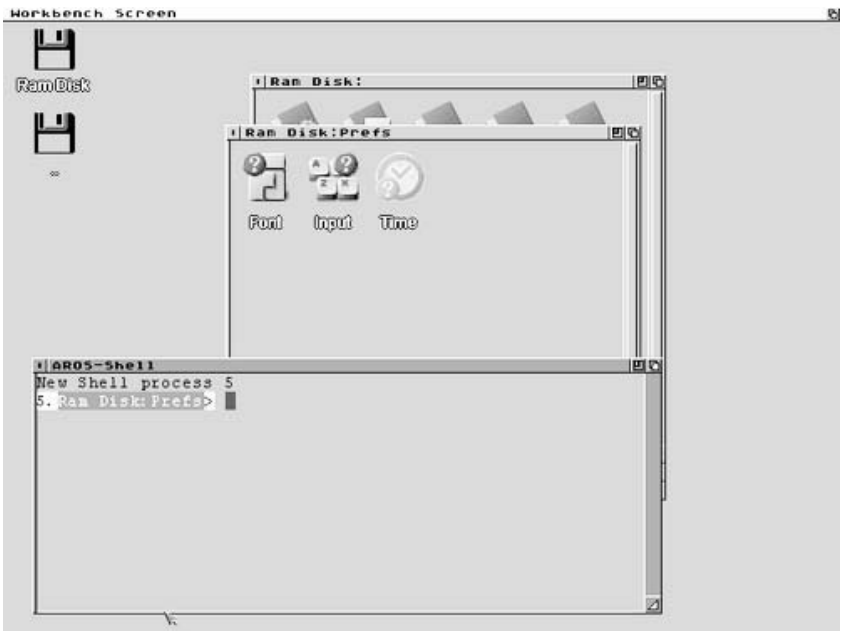

FIGURE 4.10 Booting a floppy image with QEMU.

Another portable PC emulator is called Bochs. This emulator, like QEMU, can run on a variety of operating systems providing an emulation of common PC peripherals (display, BIOS, disks, memory, etc.). Like QEMU, Bochs can support a number of guest operating systems including DOS, Windows, BSD, and, of course, Linux.

KVM

A recent addition to the virtualization arena is called KVM, which is an acronym for Kernel Virtual Machine. KVM is a paravirtualization solution that in essence turns the Linux host operating system into a hypervisor. This was an ideal solution because as it turns out the Linux kernel is an ideal base for a hypervisor.

KVM exists as a loadable module that extends a file within the proc filesystem to support virtualization. Each virtualized kernel exists as a single process in the process space of the host operating system (in this case, the hypervisor). KVM relies on QEMU for the platform virtualization and also relies on a virtualization-aware processor. This can presently be viewed as an issue, because virtualization-aware CPUs are not in widespread use. But it won't be long before these CPUs are commonplace.

Installing and virtualizing a new guest operating system is very similar to the process demonstrated with QEMU; in fact, KVM relies on QEMU. For example, the first step is to create a hard disk image with `qemu-img`:

```
$ qemu-img create -f qcow disk.img 4G
```

Then, a new OS can be installed into the emulated disk using `kvm` (the KVM image):

```
$ kvm -m 384 -cdrom newguestos.iso -hda disk.img -boot d
```

After the guest operating system is installed, it can be executed as:

```
$ kvm -m 384 -hda disk.img
```

Note here that the `-m` option specifies that the virtualized platform has 384 MB of memory available. Because KVM relies on hardware support for virtualization, it is faster than QEMU.

Summary

Virtualization is not a new technology, but it has made a huge comeback in recent years given its benefits in server environments. You can also find great benefits in virtualization, not just for running two operating systems simultaneously (for application access), but also from a development perspective. Kernel development became much simpler with the introduction of virtualization because a kernel crash doesn't require the reboot time. Instead, the virtualized kernel can be restarted, in seconds. Virtualization is one technology that can fundamentally change the landscape of computing, and Linux is at the center.

Part II GNU Tools

This part of the book focuses on GNU tools. Because a plethora of tools are available, the focus here is primarily on those that are necessary to build, segment, test and profile, and finally distribute applications.

Chapter 5: The GNU Compiler Toolchain

The GNU compiler toolchain (known as *GCC*) is the standard compiler on GNU/Linux systems (it is, after all, an acronym for *GNU Compiler Collection*). This chapter addresses compiling C programs for native systems, but GCC provides a front end for a number of different languages and back ends for almost any processor architecture you can think of.

Chapter 6: Building Software with GNU `make`

The GNU `make` utility provides a way to automatically build software based upon defined source files, source paths, and dependencies. But that's not all! The `make` utility is a general utility that can be used for a variety of tasks that have ordered dependencies. This chapter looks at the typical—and some not so typical—uses.

Chapter 7: Building and Using Libraries

Software libraries allow you to collect and compile software (objects) into a single entity. In this chapter you'll explore the methods for creating both static and dynamic libraries as well as the API functions that allow applications to build and use dynamic (shared) libraries.

Chapter 8: Building Packages with automake/autoconf

Next, this chapter takes you back to the topic of application building with a look at `automake` and `autoconf`. These tools can be used to automatically create build files for `make` based upon the given architecture and available tools. In this process, `autoconf` and `automake` can determine if the given system has the necessary elements (such as tools or libraries) to build an application correctly.

Chapter 9: Source Control in GNU/Linux

Source control is the process in which source can be managed and tracked over time. Source control is a key aspect of software development, and GNU/Linux provides many options to help protect your source. This chapter introduces the major source control paradigms and then explores the various applications available, such as CVS, Subversion, Arch, and Git.

Chapter 10: Data Visualization with Gnuplot

Visualizing data is important in any field and GNU/Linux provides an endless number of options to help organize and then visualize your data. This chapter presents a number of options and then explores the various ways that data can be portrayed and reduced. Common tools like GNUplot are covered as well as some of the more complex options like GNU Octave.

5 The GNU Compiler Toolchain

In This Chapter

- A Review of the Compilation Process
- Introduction to Common GCC Patterns
- Using the GCC Warning Options
- Using the GCC Optimizer
- Architectural Specification to GCC
- Related Tools such as `size` and `objdump`

Introduction

The GNU Compiler Collection (otherwise known as GCC) is a compiler and set of utilities to build binaries from high-level source code. GCC is not only the de facto standard compiler on GNU/Linux, but it's also the standard for embedded systems development. This is because GCC supports so many different target architectures. For example, the use in this chapter concentrates on host-based development (building software for the platform on which you are compiling), but if you were cross-compiling (building for a different target), then GCC provides for 40 different architecture families. Examples include x86, RS6000, Arm, PowerPC, and many others. GCC can also be used on over 40 different host systems (such as Linux, Solaris, Windows, or the Next operating system).

GCC also supports a number of other languages outside of standard C. You can compile for C++, Ada, Java, Objective-C, FORTRAN, Pascal, and three dialects of the C language.

This chapter looks at some of the basic features of GCC and some of the more advanced ones (including optimization). It also looks at some of the related tools within GCC that are useful in image construction (such as `size`, `objcopy`, and others).

This chapter addresses the 3.2.2 version of GCC. This is the default version for Red Hat 9.0. Newer versions of GCC now exist, but the details explored here remain compatible.

Introduction to Compilation

The GNU compiler involves a number of different stages in the process of building an object. These stages can be filtered down to four: preprocessing, compiling, assembling, and linking (see Figure 5.1).

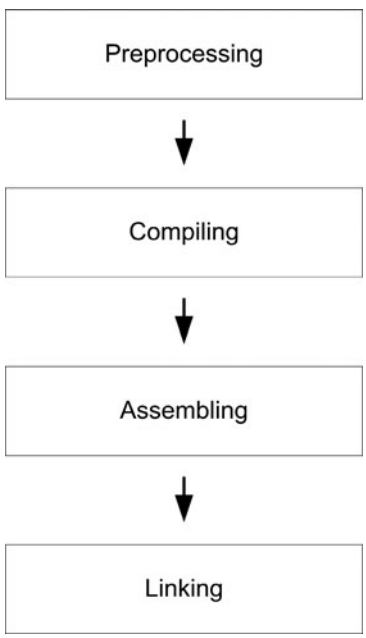

FIGURE 5.1 The stages of compilation.

The preprocessing, compiling, and assembling stages are commonly collected together into one phase, but they're shown as independent here to illustrate some of the capabilities of GCC. Table 5.1 identifies the input files and files that result.

TABLE 5.1 Compilation Stages with Inputs and Outputs

Stage	Input	Output	GCC Example
Preprocessing	`*.c`	`*.i`	`gcc -E test.c -o test.i`
Compiling	`*.i`	`*.s`	`GCC -S test.i -o test.s`
Assembling	`*.s`	`*.o`	`GCC -c test.s -o test.o`
Linking		`*.o`	`*  GCC test.o -o test`

In the preprocessing stage, the source file (`*.c`) is preprocessed with the include files (`.h` headers). At this stage, directives such as `#ifdef`, `#include`, and `#define` are resolved. The result is an intermediate file. Usually, this file isn't externally generated at all, but it is shown here for completeness. With the source file now preprocessed, it can be compiled into assembly in the compiling stage (`*.s`). The assembly file is then converted into machine instructions in the assembling stage, resulting in an object file (`*.o`). Finally, the machine code is linked together (potentially with other machine code objects or object libraries) into an executable binary.

That's enough preliminaries. Now it's time to dig into GCC and see the variety of ways it can be used. What follows first looks at a number of patterns that illustrate GCC in use and then explores some of the most useful options of GCC. This includes options for debugging, enabling various warnings, and optimizing. Then you investigate a number of GNU tools that are related to GCC.

Patterns for GCC (Compile, Compile, and Link)

The simplest example from which to begin is the compilation of a C source file to an image. In this example, the entire source necessary is contained within the single file, so you use GCC as follows:

```
$ GCC test.c -o test
```

Here you compile the `test.c` file and place the resulting executable image in a file called `test` (using the `-o` output option). If instead you wanted just the object file for the source, you'd use the `-c` flag, as follows:

```
$ GCC -c test.c
```

By default, the resulting object is named `test.o`, but you could force the output of the object to `newtest.o`, as shown here:

```
$ GCC -c test.c -o newtest.o
```

Most programs you develop involve more than one file. GCC handles this easily on the command line as shown here:

```
$ GCC -o image first.c second.c third.c
```

Here you compile three source files (`first.c`, `second.c`, and `third.c`) and link them together into the executable named `image`.

In all examples where an executable results, all C programs require a function called `main`. This is the main entry point for the program and should appear once in all the files to be compiled and linked together. When you are simply compiling a source file to an object, the link phase is not yet performed, and therefore the `main` function is not necessary.

Useful Options

In many cases, you keep your header files in a directory that's separate from where you keep your source files. Consider an example where the source is kept in a subdirectory called `./src` and at the same level is a directory where the include files are kept, `./inc`. You can tell GCC that the headers are provided there while compiling within the `./src` subdirectory as shown here:

```
$ gcc test.c -I../inc -o test
```

You could specify numerous include subdirectories using multiple `-I` specs:

```
$ gcc test.c -I../inc -I../../inc2 -o test
```

Here you specify another include subdirectory called `inc2` that is two directories up from the current directory.

For configuration of software, you can specify symbolic constants on the compile line. For example, defining a symbolic constant in the source or header as

```
#define TEST_CONFIGURATION
```

can be just as easily defined on the command line using the `-D` option as shown here:

```
$ gcc -DTEST_CONFIGURATION test.c -o test
```

The advantage to specifying this on the command line is that you need not modify any source to change its behavior (as specified by the symbolic constant).

One final useful option provides you with the means to emit a source and assembly interspersed listing. Consider the following command line:

```
$ gcc -c -g -Wa,-ahl,-L test.c
```

Most interesting in this command is the `-Wa` option, which passes the subsequent options to the assembler stage to intersperse the C source with assembly.

Compiler Warnings

Whereas the GCC compiler aborts the compilation process if an error is detected, the discovery of warnings indicates potential problems that should be fixed, though the result might still be a working executable. GCC provides a very rich warning system, but it must be enabled to take advantage of the full spectrum of warnings that can be detected.

The most common use of GCC for finding common warnings is the `-Wall` option. This turns on "all" warnings of a given type, which consists of the most generally encountered issues in applications. Its use is this:

```
$ gcc -Wall test.c -o test
```

A synonym for `-Wall` is `-all-warnings`. Table 5.2 lists the plethora of warning options that are enabled within `-Wall`.

TABLE 5.2 Warning Options Enabled in `-Wall`

Option	Purpose
`unused-function`	Warn of undefined but declared static function.
`unused-label`	Warn of declared but unused label.
`unused-parameter`	Warn of unused function argument.
`unused-variable`	Warn of unused locally declared variable.
`unused-value`	Warn of computed but unused value.
`format`	Verify that the format strings of `printf` and so on have valid arguments based upon the types specified in the format string.
`implicit-int`	Warn when a declaration doesn't specify a type.
`implicit-function-declaration`	Warn of a function being used prior to its declaration.

→

Option	Purpose
`char-subscripts`	Warn if an array is subscripted by a `char` (a common error considering that the type is signed).
`missing-braces`	Warn if an aggregate initializer is not fully bracketed.
`parentheses`	Warn of omissions of `()`s if they could be ambiguous.
`return-type`	Warn of function declarations that default to `int` or functions that lack a return, which note a return type.
`sequence-point`	Warn of code elements that are suspicious (such as `a[i] = c[i++];`).
`switch`	In `switch` statements that lack a default, warn of missing cases that would be present in the `switch` argument.
`strict-aliasing`	Use strictest rules for aliasing of variables (such as trying to alias a `void*` to a `double`).
`unknown-pragmas`	Warn of `#pragma` directives that are not recognized.
`uninitialized`	Warn of variables that are used but not initialized (enabled only with `-O2` optimization level).

Note that most options also have a negative form, so that they can be disabled (if on by default or covered by an aggregate option such as `-Wall`). For example, if you wanted to enable `-Wall` but disable the `unused` warning set, you could specify this as follows:

```
$ gcc -Wall -Wno-unused test.c -o test
```

Numerous other warnings can be enabled outside of `-Wall`. Table 5.3 provides a list of some of the more useful options and their descriptions.

One final warning option that can be very useful is `-Werror`. This option specifies that instead of simply issuing a warning if one is detected, the compiler instead treats all warnings as errors and aborts the compilation process. This can be very useful to ensure the highest quality code and is therefore recommended.

TABLE 5.3 Other Useful Warning Options Not Enabled in `-Wall`

Option	Purpose
`cast-align`	Warn whenever a pointer is cast and the required alignment is increased.
`sign-compare`	Warn if a signed vs. unsigned compare could yield an incorrect result.
`missing-prototypes`	Warn if a global function is used without a previous prototype definition.
`packed`	Warn if a structure is provided with the `packed` attribute and no packing occurs.
`padded`	Warn if a structure is padded to align it (resulting in a larger structure).
`unreachable-code`	Warn if code is found that can never be executed.
`inline`	Warn if a function marked as inline could not be inlined.
`disabled-optimization`	Warn that the optimizer was not able to perform a given optimization (required too much time or resources to perform).

GCC Optimizer

The job of the optimizer is essentially to do one of three potentially orthogonal tasks. It can optimize the code to make it faster and smaller, it can optimize the code to make it faster but potentially larger, or it can simply reduce the size of the code but potentially make it slower. Luckily, you have control over the optimizer to instruct it to do what you really want.

While the GCC optimizer does a good job of code optimization, it can sometimes result in larger or slower images (the opposite of what you might be after). It's important to test your image to ensure that you're getting what you expect. When you don't get what you expect, changing the options you provide to the optimizer can usually remedy the situation.

This section looks at the various mechanisms to optimize code using GCC.

In its simplest form, GCC provides a number of levels of optimization that can be enabled. The `-O` (oh) option permits the specification of five different optimization levels, listed in Table 5.4.

TABLE 5.4 Optimization Settings and Descriptions

Optimization Level	Description
`-O0`	No optimization (the default level).
`-O`, `-O1`	Tries to reduce both compilation time and image size.
`-O2`	More optimizations than `-O1`, but only those that don't increase size over speed (or vice versa).
`-Os`	Optimize for resulting image size (all `-O2`, except for those that increase size).
`-O3`	Even more optimizations (`-O2`, plus a couple more).

Enabling the optimizer simply entails specifying the given optimization level on the GCC command line. For example, in the following command line, you instruct the optimizer to focus on reducing the size of the resulting image:

```
$ gcc -Os test.c -o test
```

Note that you can specify different optimization levels for each file that is to make up an image. Certain optimizations (not contained within the optimization levels) require all files to be compiled with the option if one is compiled with it, but none of those are addressed here.

Now it's time to dig into the optimization levels and see what each does and also identify the individual optimizations that are provided.

-O0 Optimization

With `-O0` optimization (or no optimizer spec specified at all), the compiler simply generates code that provides the expected results and is easily debuggable within a source code debugger (such as the GNU Debugger, `gdb`). The compiler is also much faster when not optimizing, as the optimizer is not invoked at all.

-O1 Optimization (-O)

In the first level of optimization, the optimizer's goal is to compile as quickly as possible and also to reduce the resulting code size and execution time. Compilation might take more time with `-O1` (over `-O0`), but depending upon the source being compiled, this is usually not noticeable.

The individual optimizations in `-O1` are shown in Table 5.5.

TABLE 5.5 Optimizations Available in `-O1`

Optimization Level	Description
`defer-pop`	Defer popping function `args` from stack until necessary.
`thread-jumps`	Perform jump threading optimizations (to avoid jumps to `jumps`).
`branch-probabilities`	Use branch profiling to optimize branches.
`cprop-registers`	Perform a register copy-propagation optimization pass.
`guess-branch-probability`	Enable guessing of branch probabilities.
`omit-frame-pointer`	Do not generate stack frames (if possible).

The `-O1` optimization is usually a safe level if you still desire to safely debug the resulting image.

When you are specifying optimizations explicitly, the `-f` option is used to identify them. For example, to enable the `defer-pop` optimization, you simply define this as `-fdefer-pop`. If the option is enabled via an optimization level and you want it turned off, simply use the negative form `-fno-defer-pop`.

-O2 Optimization

The second optimization level provides even more optimizations (while including those in `-O1`) but does not include any optimizations that trade speed for space (or vice versa). The optimizations that are present in `-O2` are listed in Table 5.6.

Note that Table 5.6 lists only those optimizations that are unique to `-O2`; it doesn't list the `-O1` optimizations. You can assume that `-O2` is the collection of optimizations shown in Tables 5.5 and 5.6.

TABLE 5.6 Optimizations Available in `-O2`

Optimization	Description
`align-loops`	Align the start of loops.
`align-jumps`	Align the labels that are only reachable by jumps.
`align-labels`	Align all labels.
`align-functions`	Align the beginning of functions.
`optimize-sibling-calls`	Optimize sibling and tail recursive calls.

→

Optimization	Description
`cse-follow-jumps`	When performing CSE, follow jumps to their targets.
`cse-skip-blocks`	When performing CSE, follow conditional jumps.
`gcse`	Perform global common subexpression elimination.
`expensive-optimizations`	Perform a set of expensive optimizations.
`strength-reduce`	Perform strength reduction optimizations.
`rerun-cse-after-loop`	Rerun CSE after loop optimizations.
`rerun-loop-opt`	Rerun the loop optimizer twice.
`caller-saves`	Enable register saving around function calls.
`force-mem`	Copy memory operands into registers before using.
`peephole2`	Enable an `rtl` peephole pass before `sched2`.
`regmove`	Enable register move optimizations.
`strict-aliasing`	Assume that strict aliasing rules apply.
`delete-null-pointer-checks`	Delete useless null pointer checks.
`reorder-blocks`	Reorder basic blocks to improve code placement.
`schedule-insns`	Reschedule instructions before register allocation.
`schedule-insns2`	Reschedule instructions after register allocation.

-Os Optimization

The `-Os` optimization level simply disables some `-O2` optimizations that would otherwise increase the size of the resulting image. Those optimizations that are disabled for `-Os` (that do appear in `-O2`) are `-falign-labels`, `-falign-jumps`, `-falign-labels`, and `-falign-functions`. Each of these has the potential to increase the size of the resulting image, and therefore they are disabled to help build a smaller executable.

-O3 Optimization

The `-O3` optimization level is the highest level of optimization provided by GCC. In addition to those optimizations provided in `-O2`, this level also includes those shown in Table 5.7.

TABLE 5.7 Optimizations Enabled in `-O3` (Above `-O2`)

Optimization	Description
`-finline-functions`	Inline simple functions into the calling function.
`-frename-registers`	Optimize register allocation for architectures with large numbers of registers (makes debugging difficult).

Architectural Optimizations

Whereas standard optimization levels can provide meaningful improvements on software performance and code size, specifying the target architecture can also be very useful. The `-mcpu` option tells the compiler to generate instructions for the CPU type as specified. For the standard x86 target, Table 5.8 lists some of the options.

TABLE 5.8 Architectures (CPUs) Supported for x86

Target CPU	`-mcpu=`
i386 DX/SX/CX/EX/SO	`i386`
i486 DX/SX/DX2/SL/SX2/DX4	`i486`
487	`i486`
Pentium	`pentium`
Pentium MMX	`pentium-mmx`
Pentium Pro	`pentiumpro`
Pentium II	`pentium2`
Celeron	`pentium2`
Pentium III	`pentium3`
Pentium IV	`pentium4`
Via C3	`c3`
Winchip 2	`winchip2`
Winchip C6-2	`winchip-c6`
AMD K5	`i586`
AMD K6	`k6`
AMD K6 II	`k6-2`

→

Target CPU	-mcpu=
AMD K6 III	k6-3
AMD Athlon	athlon
AMD Athlon 4	athlon
AMD Athlon XP/MP	athlon
AMD Duron	athlon
AMD Tbird	athlon-tbird

So if you were compiling specifically for the Intel Celeron architecture, you'd use the following command line:

```
$ gcc -mcpu=pentium2 test.c -o test
```

Of course, combining the `-mcpu` option with an optimization level can lead to additional performance benefits. One very important point to note is that after you compile for a given CPU, the software might not run on another. Therefore, if you're more interested in an image running on a variety of CPUs, allowing the compiler to pick the default (i386) supports any of the x86 architectures.

Debugging Options

If you want to debug your code with a symbolic debugger, you can specify the `-g` flag to produce debugging information in the image for GDB. The `-g` option can specify an argument to specify which format to produce. To request debugging information to be produced using the `dwarf-2` format, you would provide the option as follows:

```
$ gcc -gdwarf-2 test.c -o test
```

Other Tools

This final section takes a look at some of the other GNU tools (usually called *binutils*) that help you in the development process.

First, how can you identify how large your executable image or intermediate object is? The `size` utility emits the text size (instruction count) and also the `data` and `bss` segments. Consider this example:

```
$ size test.o
text    data      bss     dec      hex filename
789     256       4       1049     419 test.o
$
```

Here you request the size of your intermediate object file, `test.o`. You find that the `text` size (instructions and constants) is 789 bytes, the `data` segment is 256 bytes, and the `bss` segment (which is automatically initialized to zero) is 4 bytes. If you want more detailed information on the image, you can use the `objdump` utility. You can explore the symbol table of the image or object using the `-syms` argument, as follows:

```
$ objdump -syms test.o
```

This results in a list of symbols available in the object, their type (`text`, `bss`, `data`), lengths, offset, and so on. You can also disassemble the image using the `–disassemble` argument, as follows:

```
# objdump –disassemble test.o
```

This provides a list of the functions found in the object, along with the instructions that were generated for each by GCC.

Finally, the `nm` utility can also be used to understand the symbols that are present in an object file. This utility lists not only each symbol but also detailed information on the type of the symbol. Numerous other options are available, which can be found in the `nm` main page.

Summary

In this chapter, you explored the GCC compiler and some of the related tools. You investigated some of the commonly used patterns with GCC and looked over details of the use of the warning options. You also reviewed the various optimization levels provided by GCC in addition to the architectural specifier that provides even greater optimization. Finally, you reviewed a few tools that relate to the GCC products, such as `size`, `objdump`, and `nm`.

This page intentionally left blank

6 Building Software with GNU `make`

by Curtis Nottberg

In This Chapter

- Compiling C
- Basic Makefile
- Makefile Constructs
- Dependency Tracking

Introduction

Creating a binary in a compiled language often involves lots of steps to compile all of the source files into object code and then invoke the linker to put the object code modules together into an executable. The necessary steps can all be performed by hand, but this becomes tedious very quickly. Another solution is to write a shell script to perform the commands each time. This is a better solution but has a number of drawbacks for larger projects and tends to be hard to maintain over the life of a project. Building software has a number of unique requirements that justify the development of a tool that is specifically targeted at automating the software build process. The developers of UNIX recognized this requirement early on and developed a utility named `make` to solve the problem. This chapter is an introduction to GNU `make`, the open source implementation of the `make` utility commonly used in Linux software development.

The `make` *utility was originally developed at Bell Labs in 1977 by Dr. Stuart Feldman. Dr. Feldman was part of the original team that developed UNIX at Bell Labs and also wrote the first Fortran 77 compiler.*

A Sample Project

The approach used in this chapter will be to introduce a simple sample project and then show how to build the project starting with a command-line solution and progress to a fairly complete GNU `make` implementation. The examples in this chapter will show various ways to build a project consisting of four source files. The diagram shown in Figure 6.1 illustrates the directory layout of the project.

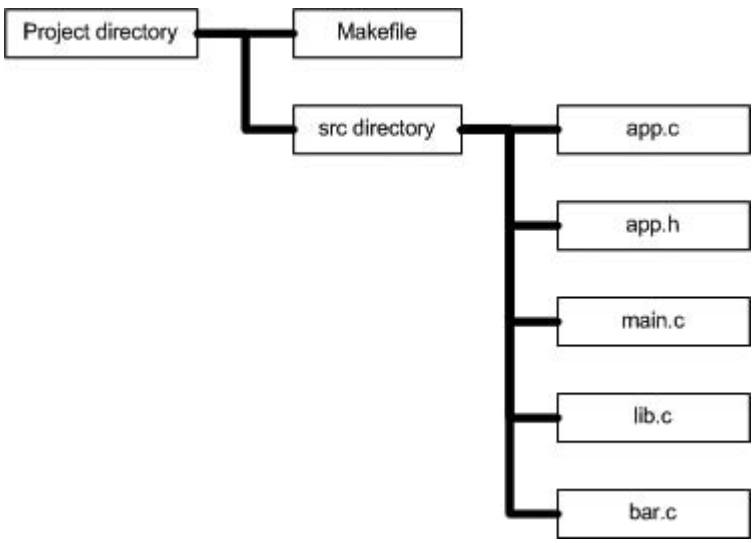

FIGURE 6.1 Directory structure of sample project.

Compiling by Hand

The simplicity of the sample project makes it very easy to compile by hand. Executing the following command in the top-level project directory generates the application named `appexp` with a single command.

```
gcc -o appexp src/main.c src/app.c src/bar.c src/lib.c
```

This command runs the GCC wrapper program that invokes the preprocessor, compiler, and linker to turn these four c-files into an executable. The single command approach is acceptable for such a simple project, but for a larger project this would be impractical. The following set of commands breaks the compilation into the incremental steps commonly seen in a more complicated build process.

```
gcc -c -o main.o src/main.c
gcc -c -o app.o src/app.c
gcc -c -o bar.o src/bar.c
gcc -c -o lib.o src/lib.c
gcc -o appexp main.o app.o bar.o lib.o
```

The first four commands in this series turn the c-files in the `src` directory into object files in the top-level directory. The last command invokes the linker to combine the four generated object files into an executable.

A Build Script

Typing any of the commands described in the previous section would get pretty tedious if it had to be done every time the application needed to be rebuilt. The next obvious step is to put these commands into a script so that a single command can perform all of the steps needed to build the application. Listing 6.1 shows the contents of the `buildit` script that might be written to automate the build process.

LISTING 6.1 The `buildit` Script (on the CD-ROM at `./source/ch6/buildit`)

```
1:      #!/bin/sh
2:      # Build the chapter 6 example project
3:
4:      gcc -c -o main.o src/main.c
5:      gcc -c -o app.o src/app.c
6:      gcc -c -o bar.o src/bar.c
7:      gcc -c -o lib.o src/lib.c
8:      gcc -o appexp main.o app.o bar.o lib.o
```

The script collects the commands outlined in the previous section into one place. It allows the developer and user of the source code to build the application with the simple `./buildit` command line. Also, it can be revision controlled and distributed with the source code to ease the understanding needed by those trying to build an application.

One of the disadvantages of the build script is that it rebuilds the entire project every time it is invoked. For the small example in this chapter, this does not cause a significant time increase in the development cycle, but as the number of files increases, rerunning the entire build process turns into a significant burden. One of the major enhancements of the `make` utility over a shell-script solution is its capability to understand the dependencies of a project. Understanding the dependencies allows the `make` utility to rebuild only the parts of the project that need updating because of source file changes.

A Simple Makefile

The make utility uses a developer-created input file to describe the project to be built. GNU make uses the name Makefile as the default name for its input file. Thus, when the make utility is invoked by typing the make command, it looks in the current directory for a file named Makefile to tell it how to build the project. Listing 6.2 illustrates a basic Makefile used to build the sample project.

LISTING 6.2 Simple Makefile (on the CD-ROM at ./source/ch6/Makefile.simple)

```
1:      appexp: main.o app.o bar.o lib.o
2:              gcc -o appexp main.o app.o bar.o lib.o
3:
4:      main.o : src/main.c src/lib.h src/app.h
5:              gcc -c -o main.o src/main.c
6:
7:      app.o : src/app.c src/lib.h src/app.h
8:              gcc -c -o app.o src/app.c
9:
10:     bar.o : src/bar.c src/lib.h
11:             gcc -c -o bar.o src/bar.c
12:
13:     lib.o : src/lib.c src/lib.h
14:             gcc -c -o lib.o src/lib.c
```

Line 1 illustrates the basic construct in Makefile syntax, the *rule*. The portion of the rule before the colon is called the *target,* whereas the portion after the colon is the rule *dependencies*. Rules generally have commands associated with them that turn the prerequisites into the target. In the specific case of line 1, the target of the rule is the appexp application that depends on the existence of main.o, app.o, bar.o, and lib.o before it can be built. Line 2 illustrates the command used to invoke the linker to turn the main.o, app.o, bar.o, and lib.o object files into the appexp executable. It is important to note that one of the idiosyncrasies of Makefile syntax is the need to put a hard tab in front of the commands in a Makefile; this is how the make utility differentiates commands from other Makefile syntax. The make utility uses rules to determine how to build a project. When the make utility is invoked on the command line, it parses the Makefile to find all of the target rules. It then attempts to build a target with the name all; if the all target has not been defined, then make builds the first target it encounters in the Makefile.

For the Makefile in Listing 6.2, the default target is the appexp application because its rule (line 1) occurs first in the Makefile. The neat part about the Makefile rule syntax is that the make utility can chain the rules together to create the whole

build process. When the make utility is invoked, it begins to build the project by trying to build the default rule: rule 1 in the sample Makefile. The rule on line 1 tells the make utility that to build appexp, it must have the files main.o, app.o, bar.o, and lib.o. make checks for the existence of those files to determine if it has everything needed to build the application. If one of the prerequisite files is missing or is newer than the target, then make starts searching the target rules to determine if it has a rule to create the prerequisite.

After an appropriate rule is identified, then the process starts again by ensuring that the new rule's prerequisites exist. Thus the make utility chains rules together into a tree of dependencies that must be satisfied to build the original target. make then starts executing the commands associated with the rules at the leaves of the tree to build the prerequisites needed to move back toward the root of the tree. In the sample Makefile, the process would start with the default rule on line 1. If the object files didn't exist yet, then the make utility would find the rules to make them. First, it would find a rule to create main.o, which occurs on line 4. Next, make would examine the rule on line 4 and realize that the main.c, lib.h, and app.h prerequisites all exist, so the commands to create main.o (line 5) would be executed. The next prerequisite needing to be created would be app.o, so make would move to the rule on line 7 and execute the command on line 8. This process would then continue to create bar.o and lib.o, and finally the command on line 2 would be executed to create the application. If the make command is executed in a clean directory, then you can expect that make would automatically execute the following series of commands:

```
gcc -c -o main.o src/main.c
gcc -c -o app.o src/app.c
gcc -c -o bar.o src/bar.c
gcc -c -o lib.o src/lib.c
gcc -o appexp main.o app.o bar.o lib.o
```

Comparing this to the script in Listing 6.1, you can see that you have reimplemented the simple build script using GNU make. So why use GNU make instead of a build script? After all, it seems like a more complicated way to implement the same steps the build script took care of for you. The answer is the capability of GNU make to build things in an incremental fashion based upon the dependencies that are set up in the Makefile. For example, suppose after building the appexp program you discover an error in the app.c source file. To correct this, you can edit the app.c source file and then rebuild the executable. If you were using the Listing 6.1 script, then all of the source files are rebuilt into objects, and those objects are linked into an executable. On the other hand, with the make utility, the rebuild is accomplished with the following two commands:

```
gcc -c -o app.o src/app.c
gcc -o appexp main.o app.o bar.o lib.o
```

How did GNU `make` eliminate the need for the other commands? Because GNU `make` understands each step that goes into the creation of the executable, it can examine the dates on the files to determine that nothing in the dependency tree for `main.o`, `bar.o`, and `lib.o` changed, and thus these object files don't need to be recreated. Conversely, it examines `app.o` and realizes one of its dependencies, namely `app.c`, did change, so it needs to be rebuilt. If you understand the dependency tree that is represented in the Makefile syntax, then you understand the power of `make` over a simple build script. The Makefile syntax also provides other capabilities beyond those that have been discussed in this section; the rest of this chapter will consider some of the more useful capabilities of GNU `make`.

Makefile Variables

GNU `make` provides support for a form of variable that closely resembles the variables provided in the standard UNIX shells. The variables allow a name to be associated with an arbitrarily long text string. The basic syntax to assign a value to a variable is as follows:

```
MY_VAR = A text string
```

This results in the creation of a variable named `MY_VAR` with a value of `A text string`. The value of a variable can be retrieved in subsequent portions of the Makefile by using the following syntax: `${var-name}`, where `var-name` is replaced with the name of the variable that is being retrieved. Listing 6.3 provides a sample Makefile to illustrate the basic use of variables in a Makefile.

LISTING 6.3 Simple Variable Makefile (on the CD-ROM at `./source/ch6/Makefile.simpvar`)

```
1:
2:      MY_VAR=file1.c file2.c
3:
4:      all:
5:              echo ${MY_VAR}
6:
```

Line 2 assigns the value `file1.c file2.c` to the `MY_VAR` variable. When the `make` utility is invoked, it attempts to build the `all` target defined on line 4. The command on line 5 is executed, and the output is illustrated in Listing 6.4.

LISTING 6.4 Simple Variable Makefile Output

```
1:
2:      $ make
3:      echo file1.c file2.c
4:      file1.c file2.c
5:
```

Line 2 is the invocation of the `make` utility. Line 3 is output by the `make` utility when it runs the command on line 5 of Listing 6.3. Notice that the variable replacement has already occurred by this point in the execution. Line 4 is the actual output from the `echo` command that `make` invokes.

GNU `make` allows the value in a variable to be constructed incrementally by providing the ability to concatenate strings to the existing value of a variable. GNU `make` syntax uses the addition symbol to indicate a concatenation operation. Listing 6.5 illustrates this by modifying the simple Makefile in Listing 6.3 to use the concatenation operation when creating the value in the `MY_VAR` variable.

LISTING 6.5 Variable Concatenation Makefile (on the CD-ROM at `./source/ch6/Makefile.varconcat`)

```
1:
2:      MY_VAR=file1.c
3:      MY_VAR+=file2.c
4:
5:      all:
6:              echo ${MY_VAR}
7:
```

Running `make` against the Makefile in Listing 6.5 should generate the exact same output, illustrated in Listing 6.4, as the Makefile in Listing 6.3.

GNU `make` provides a wide range of built-in functionality to operate on defined variables. Functions are provided for general string-handling operations such as substitution, stripping, and tokenizing. A common use of variables in Makefiles is to store and manipulate filenames and paths that are involved in the `make` process. To facilitate this use of variables, GNU `make` provides specialized functions that operate on the contents of a variable as a path or filename. The contrived Makefile in Listing 6.6 illustrates the use of some of the functions provided by GNU `make`. Listing 6.7 illustrates the expected output from running the `make` utility on the Makefile in Listing 6.6.

LISTING 6.6 Variable Manipulation Makefile (on the CD-ROM at `./source/ch6/Makefile.varmanip`)

```
1:
2:      SRC_VAR=My test string for variable manipulation.
3:
4:      TEST1_VAR=$(subst for,foo,${SRC_VAR})
5:      TEST2_VAR=$(patsubst t%t,T%T, ${SRC_VAR})
6:      TEST3_VAR=$(filter %ing %able, ${SRC_VAR})
7:      TEST4_VAR=$(sort ${SRC_VAR})
8:      TEST5_VAR=$(words ${SRC_VAR})
9:      TEST6_VAR=$(word 2,${SRC_VAR})
10:     TEST7_VAR=$(wordlist 2, 3, ${SRC_VAR})
11:
12:     all:
13:             @echo original str: ${SRC_VAR}
14:             @echo substitution: ${TEST1_VAR}
15:             @echo pattern sub : ${TEST2_VAR}
16:             @echo filter   : ${TEST3_VAR}
17:             @echo sort     : ${TEST4_VAR}
18:             @echo word count : ${TEST5_VAR}
19:             @echo word 2   : ${TEST6_VAR}
20:             @echo word 2 thru 4: ${TEST7_VAR}
```

Line 2 sets up the source string that is used to exercise the GNU `make` string manipulation functions. Lines 4–10 illustrate the use of the Makefile functions: `subst`, `patsubst`, `filter`, `sort`, `words`, `word`, and `wordlist`. Notice that the `make` file syntax for a function call takes the general form of `${func arg1,arg2,...)`. Lines 13–20 output the results when the Makefile is evaluated. Notice the use of the `@` before the `echo` commands in lines 13–20, which tells the `make` utility to suppress the printing of the command line before it executes.

LISTING 6.7 Output from the Makefile in Listing 6.6

```
1:
2:      original str: My test string for variable manipulation.
3:      substitution: My test string foo variable manipulation.
4:      pattern sub : My TesT string for variable manipulation.
5:      filter : string variable
6:      sort : My for manipulation. string test variable
7:      word count : 6
8:      word 2 : test
9:      word 2 thru 4: test string
```

Line 2 outputs the value of the original string that is to be manipulated with the Makefile functions. Line 3 illustrates the output of the `subst` function, substituting the word `for` with the word `foo`. Line 4 illustrates the output of the `patsubst` function; the word `test` has been modified to `TesT`. Notice how the `patsubst` function uses the wildcard `%` to match one or more characters in a pattern. The result of the wildcard match can then be substituted back into the replacement string by using the `%` character. Line 5 illustrates the use of the `filter` function, which again makes use of the `%` wildcard character to match words ending in `ing` or `able`. Line 6 shows the output of the `sort` function that rearranges the input variables into lexical order. Line 7 illustrates the output of the `words` function that performs a word count operation on the input string. Line 8 illustrates the `word` function that allows the subsetting of a string by numerical index. In this case the second word, `test`, is selected. Notice that the indexes start at one rather than zero. Line 8 illustrates the `wordlist` function that allows a substring to be extracted based on word indexes. In the example, words 2 through 3 have been extracted, resulting in the substring `test string`.

The example in Listing 6.7 provides a taste of some of the string-manipulation functions provided by GNU `make`. GNU `make` provides many others that can be found by consulting the GNU `make` documentation.

One of the primary uses of variables in a Makefile is to contain and manipulate the filename and path information associated with the build process. Take note of the use of variables in the next couple of sections, where they are employed in more realistic Makefile examples.

Pattern-Matching Rules

The conversion of one type of file into another in a Makefile often follows a very specific pattern that varies only with the conversion and not the specific file. If this is the case, then GNU `make` provides a special type of rule that allows the target and dependencies to both be specified as patterns. Listing 6.8 contains a Makefile for the sample project introduced at the beginning of this chapter. This example uses a pattern-matching rule in combination with the GNU `make VPATH` capability to simplify the Makefile. Still problematic are the header file dependencies that have been sequestered at the bottom of the new Makefile. The next section introduces one mechanism for automating the processing of header dependencies.

LISTING 6.8 More Realistic Makefile (on the CD-ROM at `./source/ch6/Makefile.realistic`)

```
1:
2:      SRC_FILES=main.c app.c bar.c lib.c
3:      OBJ_FILES=$(patsubst %.c, %.o, ${SRC_FILES})
```

```
4:
5:      VPATH = src
6:
7:      CFLAGS = -c -g
8:      LDFLAGS = -g
9:
10:     appexp: ${OBJ_FILES}
11:             gcc ${LDFLAGS} -o appexp ${OBJ_FILES}
12:
13:     %.o:%.c
14:             gcc ${CFLAGS} -o $@ $
    15:
16:     clean:
17:             rm *.o appexp
18:
19:     MAIN_HDRS=lib.h app.h
20:     LIB_HDRS=lib.h
21:
22:     main.o : $(addprefix src/, ${MAIN_HDRS})
23:     app.o : $(addprefix src/, ${MAIN_HDRS})
24:     bar.o : $(addprefix src/, ${LIB_HDRS})
25:     lib.o : $(addprefix src/, ${LIB_HDRS})
```

This Makefile is quite different from the first one introduced in Listing 6.2, but it accomplishes the same basic task as the original Makefile. First, you should notice the introduction of variables into the Makefile to allow the control parameters (compile flags, file lists, and so on) to be enumerated and set at the top of the Makefile. Lines 2 through 8 set up the variables that control the build process. Line 3 illustrates the use of the pattern substitution function to generate a list of object files on which the final application is dependent. The pattern substitution results in the `OBJ_FILES` variable being assigned the value `main.o app.o bar.o lib.o`. Therefore, the `OBJ_FILES` variable now contains a list of the object files that are needed to link the `appexp` executable. Line 10 is the rule that explicitly states this relationship; it uses the value of the `OBJ_FILES` variable as its dependency list. Lines 16 and 17 illustrate a target that is often found in standard Makefiles. The name of the target is `clean`, and its responsibility is to remove files that are generated by the `make` process. This provides a nice mechanism for a developer to clean up generated files so that a complete rebuild of the entire project can occur. Lines 16 through 25 represent the header file dependencies. It will be left to you to understand how this works because in the next section a different mechanism is introduced to handle include file dependencies automatically.

The most interesting part of Listing 6.8 is the addition on lines 5, 13, and 14. Line 13 introduces a pattern rule to indicate to the make utility how to transform an arbitrary file ending with a .c extension into a corresponding file ending in a .o extension. The transformation is accomplished by executing the commands associated with the pattern rule, in this case the command on line 14. Notice that the command on line 14 uses some special variable references to indicate the files that GCC should operate on. GNU make provides a large number of these special variables to allow the commands in a pattern rule to gain access to information about the matched files. In the specific case of line 14, the $@ variable contains the filename matched for the left side of the rule, and the $ variable contains the filename matched for the right side of the variable. GNU make provides a large number of these special variables for pattern rules that are documented in the GNU make manual. This one pattern rule replaces the four specific rules used in Listing 6.2 to build the object files.

The previous paragraph breezed over an important detail concerning the use of pattern-matching rules. The pattern-matching rules don't really take into account filename and path when performing comparisons. They assume that the left and right sides of the pattern rule both occur in the same directory. In the example provided in this chapter that is not the case, because the source files are one level removed from the location of the Makefile. The source files are all contained in the src directory. To resolve the situation, the VPATH feature of GNU make is used to provide the pattern-matching rules with a list of search directories to use when the right side of a rule isn't found in the current directory. In line 5, the special VPATH variable is set to src so that the pattern rule on line 13 can find the source files it needs when trying to generate the object files.

So what have you gained between Listing 6.2 and Listing 6.8? The Makefile in Listing 6.8 scales much better. It attempts to make a distinction between the operations used to build an application and the files on which those operations are performed. All of the variable aspects of the build process are controlled in the variables at the top of the Makefile, whereas the actions are contained in the rules lower in the file. Thus, to add or delete files from the build, you need only modify the variables at the top of the file. Listing 6.8 still has a problem with the include file tracking, but the next section illustrates how to resolve this situation.

Automatic Dependency Tracking

You encounter problems with including header file dependencies in a Makefile. For example, keeping the Makefile current with the #include directives in the source files becomes problematic as the project grows. Because the preprocessor in the toolchain already has to resolve the #includes, most modern compilers provide a mechanism to output these rules automatically. The generated rules can then be used in

the Makefile to track changes to the `#include` structure and rebuild the project appropriately. Listing 6.9 illustrates the Listing 6.8 Makefile modified to use the automatic dependency tracking mechanism proposed in the GNU `make` manual.

LISTING 6.9 Makefile with Dependency Tracking (on the CD-ROM at `./source/ch6/Makefile.deptrack`)

```
1:
2:      SRC_FILES=main.c app.c bar.c lib.c
3:      OBJ_FILES=$(patsubst %.c, %.o, ${SRC_FILES})
4:      DEP_FILES=$(patsubst %.c, %.dep, ${SRC_FILES})
5:
6:      VPATH = src
7:
8:      CFLAGS = -c -g
9:      LDFLAGS = -g
10:
11:     appexp: ${OBJ_FILES}
12:             gcc ${LDFLAGS} -o appexp ${OBJ_FILES}
13:
14:     %.o:%.c
15:             gcc ${CFLAGS} -o $@ $
16:
17:     clean:
18:             rm *.o appexp
19:
20:     include ${DEP_FILES}
21:
22:     %.dep: %.c
23:             @set -e; rm -f $@; \
24:             gcc -MM $(CFLAGS) $< > $@.$$$$; \
25:             sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@;
\
26:             rm -f $@.$$$$
```

Listing 6.9 is very similar to Listing 6.8 except for line 4 and lines 20 through 26. Line 4 is creating a variable based on the source file list by replacing the `.c` extension with a `.dep` extension. The `.dep` files contain the generated dependency rules for each source file. Line 20 is the most important line in the new Makefile because it establishes a dependency between the main Makefile and all the generated `.dep` files. When `make` goes to evaluate Listing 6.9, it first realizes that to evaluate the Makefile it needs to find all of the `*.dep` files that are included by line 20. `make` must also ensure that all of the `*.dep` files are up to date before it can proceed. So how

do the .dep files get generated? When make is trying to include the *.dep files, it also evaluates the rules in the current Makefile to see if it knows how to create the *.dep files from other files it knows about. Lines 22 through 26 are a pattern-matching rule that tell make how to create .dep files given a .c file. Line 24 invokes the C compiler to generate the basic #include dependency rule. The result is dumped into a temporary file. Line 25 then uses sed to massage the output so that the .dep file itself is dependent on the same dependencies; when any of the dependent files change, the .dep file gets rebuilt as well.

The following is the content of the main.dep file generated by the Makefile in Listing 6.9.

```
main.o main.dep : src/main.c src/lib.h src/app.h
```

This generated rule indicates that main.o and main.dep are dependent on the source files main.c, lib.h, and app.h. Comparing the generated rule against the same handwritten rule in previous listings illustrates that you have automated this process in Listing 6.9.

The method chosen for automatic dependency tracking in this section is the one proposed in the GNU make manual. Numerous other mechanisms have been employed to accomplish the same thing; please consult the resources to find the one that works best for your application.

Summary

A good understanding of make and how it operates is an essential skill in any modern software development environment. This chapter provided a brief introduction to some of the capabilities of the GNU make utility. GNU make has a rich set of capabilities beyond what was discussed in this chapter; a consultation of the resources can help you if you will be using make extensively.

Most of the large projects in Linux software development do not employ make directly but instead employ the GNU automake/autoconf utilities that are layered on top of GNU make. Chapter 8, "Building Packages with automake/autoconf," introduces GNU automake/autoconf, which should be used when starting a new Linux development project. Don't worry; the knowledge of GNU make introduced in this chapter is still invaluable when understanding and debugging problems in the GNU automake/autoconf environment.

This page intentionally left blank

7 Building and Using Libraries

In This Chapter

- Introduction to Libraries
- Building and Using Static Libraries
- Building and Using Shared Libraries
- Building and Using Dynamic Libraries
- GNU/Linux Library Commands

Introduction

This chapter explores the topic of program libraries. First you will investigate static libraries and their creation using the `ar` command. Then you'll look at shared libraries (also called dynamically linked libraries) and some of the advantages (and complexities) they provide. Finally, you'll take a look at some of the utilities that manipulate libraries in GNU/Linux.

What Is a Library?

A library is really nothing more than a collection of object files (a container). When the object files collectively provide related behaviors in a given problem, the objects can be integrated into a library to simplify their access and distribution for application developers.

Static libraries are created using the `ar`, or archive, utility. After the application developer compiles and then links with the library, the needed elements of the library are integrated into the resulting executable image. From the perspective of the application, the external library is no longer relevant because it has been combined with the application image.

One of the most famous libraries is the standard C library that contains the set of ANSI C library functions (string functions, math functions, etc.). One of the most popular is the GNU C Library (`glibc`*), which is the GNU version of the standard C library. Lighter-weight versions of the standard C library exist, such as* `newlib`*,* `uClibc`*, and* `dietlibc` *(which are ideal for embedded applications).*

Shared, or dynamic, libraries are also linked with an application image, but in two separate stages. In the first stage (at the application's build time), the linker verifies that all of the symbols necessary to build the application (functions or variables) are available within either the application or libraries. Rather than pull the elements from the shared library into the application image (as was done with the static library), at the second stage (at runtime) a dynamic loader pulls the necessary shared libraries into memory and then dynamically links the application image with them. These steps result in a smaller image, as the shared library is separate from the application image (see Figure 7.1).

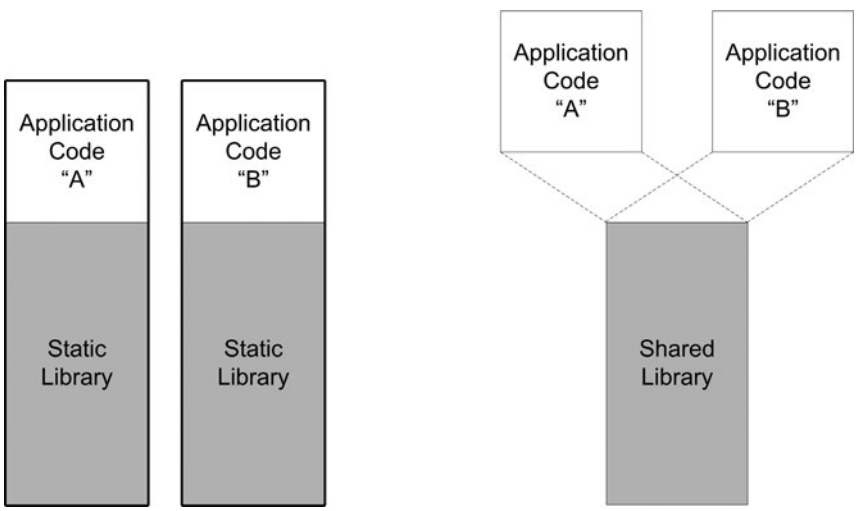

FIGURE 7.1 Memory savings of static versus shared libraries.

The tradeoff to this memory saving for shared libraries is that the libraries must be resolved at runtime. This requires a small amount of time to figure out which libraries are necessary, find them, and then bring them into memory.

The next sections take you through building a couple of libraries using both the static and shared methods to see how they're built and how the program changes to support them.

Building Static Libraries

First, take a look at the simplest type of library development in GNU/Linux. The static library is linked statically with the application image. This means that after the image is built, the external library does not need to be present for the image to execute properly as it is a part of the resulting image.

To demonstrate the construction of libraries, take look at a sample set of source files. This sample builds a simple random number generator wrapper library using the GNU/Linux random functions. First take a look at the API for our library. The header file, `randapi.h`, is shown in Listing 7.1.

LISTING 7.1 Random Number Wrapper API Header (on the CD-ROM at `./source/ch7/statshrd/randapi.h`)

```
1:      /*
2:       * randapi.h
3:       *
4:       * Random Functions API File
5:       *
6:       */
7:
8:
9:      #ifndef __RAND_API_H
10:     #define __RAND_API_H
11:
12:     extern void initRand( void );
13:
14:     extern float getSRand( void );
15:
16:     extern int getRand( int max );
17:
18:     #endif /* __RAND_API_H */
```

This API consists of three functions. The first function, `initRand`, is an initialization function that prepares the wrapper libraries for use. It must be called once prior to calling any of the random functions. Function `getSRand()` returns a random floating-point value between 0.0 and 1.0. Finally, function `getRand(x)` returns a random integer between 0 and (x–1).

While this functionality could be implemented in a single file, this example splits it between two files for the purposes of demonstration. The next file, `initrand.c`, provides the initialization function for the wrapper API (see Listing 7.2). The single function, `initRand()`, simply initializes the random number generator using the current time as a seed.

LISTING 7.2 Random Number Initialization API (on the CD-ROM at `./source/ch7/statshrd/initapi.c`)

```
1:      /*
2:       * Random Init Function API File
3:       *
4:       */
5:
6:      #include <stdlib.h>
7:      #include <time.h>
8:
9:
10:     /*
11:      *  initRand() initializes the random number generator.
12:      *
13:      */
14:
15:     void initRand()
16:     {
17:       time_t seed;
18:
19:       seed = time(NULL);
20:
21:       srand( seed );
22:
23:       return;
24:     }
```

The final API file, `randapi.c`, provides the random number functions (see Listing 7.3). The integer and floating-point random number wrapper functions are provided here.

LISTING 7.3 Random Number Wrapper Functions (on the CD-ROM at `./source/ch7/statshrd/randapi.c`)

```
1:      /*
2:       * randapi.c
3:       *
4:       * Random Functions API File
5:       *
6:       */
7:
8:      #include <stdlib.h>
9:
10:
11:     /*
12:      *  getSRand() returns a number between 0.0 and 1.0.
13:      *
14:      */
15:
16:     float getSRand()
17:     {
18:       float randvalue;
19:
20:       randvalue = ((float)rand() / (float)RAND_MAX);
21:
22:       return randvalue;
23:     }
24:
25:
26:     /*
27:      *  getRand() returns a number between 0 and max-1.
28:      *
29:      */
30:
31:     int getRand( int max )
32:     {
33:       int randvalue;
34:
35:       randvalue = (int)((float)max * rand() / (RAND_MAX+1.0));
36:
37:       return randvalue;
38:     }
```

That's it for the API. Note that both `initapi.c` and `randapi.c` use the single header file `randapi.h` to provide their function prototypes. Now it's time to take a quick look at the test program that utilizes the API and then get back to the task at hand—libraries!

Listing 7.4 provides the test application that uses the wrapper function API. This application provides a quick test of the API by identifying the average value provided, which should represent the average around the middle of the random number range.

LISTING 7.4 Test Application for the Wrapper Function API (on the CD-ROM at `./source/ch7/statshrd/test.c`)

```
1:      #include "randapi.h"
2:
3:      #define ITERATIONS      1000000L
4:
5:      int main()
6:      {
7:        long  i;
8:        long  isum;
9:        float fsum;
10:
11:       /* Initialize the random number API */
12:       initRand();
13:
14:       /* Find the average of getRand(10) returns (0..9) */
15:       isum = 0L;
16:       for (i = 0 ; i < ITERATIONS ; i++) {
17:
18:         isum += getRand(10);
19:
20:       }
21:
22:       printf( "getRand() Average %d\n", (int)(isum /
          ITERATIONS) );
23:
24:
25:       /* Find the average of getSRand() returns */
26:       fsum = 0.0;
27:       for (i = 0 ; i < ITERATIONS ; i++) {
28:
29:         fsum += getSRand();
30:
31:       }
```

```
32:
33:         printf( "getSRand() Average %f\n", (fsum /
            (float)ITERATIONS) );
34:
35:         return;
36:       }
```

If you wanted to build all source files discussed here and integrate them into a single image, you could do the following:

```
$ gcc initapi.c randapi.c test.c -o test
```

This compiles all three files and then links them together into a single image called `test`. This use of `gcc` provides not only compiling of the source files, but also linking to a single image. Upon executing the image, you see the averages for each of the random number functions:

```
$ ./test
getRand() Average 4
getSRand() Average 0.500001
$
```

As expected, the random number that is generated will generate an average value that's in the middle of the random number range.

Now it's time to get back to the subject of libraries, and rather than build the entire source together, you'll build a library for the random number functions. This is achieved using the `ar` utility (archive). Next is the demonstration of the building of the static library along with the construction of the final image.

```
$ gcc -c -Wall initapi.c
$ gcc -c -Wall randapi.c
$ ar -cru libmyrand.a initapi.o randapi.o
$
```

In this example, you first compile the two source files (`initapi.c` and `randapi.c`) using `gcc`. You specify the `-c` option to tell `gcc` to compile only (don't link) and also to turn on all warnings. Next, you use the `ar` command to build the library (`libmyrand.a`). The `cru` options are a standard set of options for creating or adding to an archive. The `c` option specifies to create the static library (unless it already exists, in which case the option is ignored). The `r` option tells `ar` to replace existing objects in the static library (if they already exist). Finally, the `u` option is a safety option to specify that objects are replaced in the archive only if the objects to be inserted are newer than existing objects in the archive (of the same name).

You now have a new file called `libmyrand.a`, which is your static library containing two objects: `initapi.o` and `randapi.o`. Now look at how you can build your application using this static library. Consider the following:

```
$ gcc test.c -L. -lmyrand -o test
$ ./test
getRand() Average 4
getSRand() Average 0.499892
$
```

Here you use `gcc` to first compile the file `test.c` and then link the `test.o` object with `libmyrand.a` to produce the test image file. The `-L.` option tells `gcc` that libraries can be found in the current subdirectory (the period represents the directory). Note that you can also provide a specific subdirectory for the library, such as `-L/usr/mylibs`. The `-L` option identifies the library to use. Note that `myrand` isn't the name of your library, but instead it is `libmyrand.a`. When the `-L` option is used, it automatically surrounds the name specified with `lib` and `.a`. Therefore, if you had specified `-ltest`, `gcc` would look for a library called `libtest.a`.

Now that you see how to create a library and use it to build a simple application, you can return to the `ar` utility to see what other uses it has. You can inspect a static library to see what's contained within it by using the `-t` option:

```
$ ar -t libmyrand.a
initapi.o
randapi.o
$
```

If desired, you can also remove objects from a static library. This is done using the `-d` option, such as:

```
$ ar -d libmyrand.a initapi.o
$ ar -t libmyrand.a
randapi.o
$
```

It's important to note that `ar` won't actually show a failure for a delete. To see an error message if the delete fails, add a `-v` option as shown in the following:

```
$ ar -d libmyrand.a initapi.o
$ ar -dv libmyrand.a initapi.o
No member named ‘initapi.o’
$
```

In the first case, you try to delete the object `initapi.o`, but no error message is generated (even though it doesn't exist in the static library). In the second case, you add the verbose option and the corresponding error message results.

Rather than remove the object from the static library, you can extract it using the `-x` option.

```
$ ar -xv libmyrand.a initapi.o
x - initapi.o
$ ls
initapi.o  libmyrand.a
$ ar -t libmyrand.a
randapi.o
initapi.o
$
```

The extract option is coupled with verbose (`-v`) so that you can see what `ar` is doing. The `ar` utility responds with the file being extracted (`x - initapi.o`), which you can see after doing an `ls` in the subdirectory. Note here that you also list the contents of the static library after extraction, and your `initapi.o` object is still present. The extract option doesn't actually remove the object, it only copies it externally to the archive. The delete (`-d`) option must be used to remove it outright from the static library.

The `ar` utility option list is shown in Table 7.1.

TABLE 7.1 Important Options for the `ar` Utility

Option	Name	Example
`-d`	Delete	`ar -d <archive> <objects>`
`-r`	Replace	`ar -r <archive> <objects>`
`-t`	Table list	`ar -t <archive>`
`-x`	Extract	`ar -x <archive> <objects>`
`-v`	Verbose	`ar -v`
`-c`	Create	`ar -c <archive>`
`-ru`	Update object	`ar -ru <archive> <objects>`

Building Shared Libraries

Now it's time to try the test application again, this time using a shared library instead of a static library. The process is essentially just as simple. First, build a shared library using the `initapi.o` and `randapi.o` objects. One change is necessary when building source for a shared library. Because the library and application are not tied together as they are in a static library, the resulting library can't assume anything about the binding application. For example, in addressing, references must be relative (through the use of a GOT, or Global Offset Table). The loader automatically resolves all GOT addresses as the shared library is loaded. To build source files for position independence, we use the `PIC` option of `gcc`:

```
$ gcc -fPIC -c initapi.c
$ gcc -fPIC -c randapi.c
```

This results in two new object files containing position-independent code. You can create a shared library of these using `gcc` and the `-shared` flag. This flag tells `gcc` that a shared library is to be created:

```
$ gcc -shared initapi.o randapi.o -o libmyrand.so
```

You specify your two object modules with an output file (`-o`) as your shared library. Note that you use the `.so` suffix to identify that the file is a shared library (shared object).

To build your application using the new shared object, you link the elements back together as you did with the static library:

```
$ gcc test.c -L. -lmyrand -o test
```

You can tell that the new image is dependent upon the shared library by using the `ldd` command. The `ldd` command prints shared library dependencies for the given application. For example:

```
$ ldd test
    libmyrand.so => not found
    libc.so.6 => /lib/tls/libc.so.6 (0x42000000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$
```

The `ldd` command identifies the shared libraries to be used by `test`. The standard C library is shown (`libc.so.6`) as is the dynamic linker/loader (`ld-linux.so.2`). Note that the `libmyrand.so` file is shown as not found. It's present in the current sub-

directory with the application, but it must be explicitly specified to GNU/Linux. You do this through the `LD_LIBRARY_PATH` environment variable. After exporting the location of the shared library, you try the `ldd` command again:

```
$ export LD_LIBRARY_PATH=./
$ ldd test
        libmyrand.so => ./libmyrand.so (0x40017000)
        libc.so.6 => /lib/tls/libc.so.6 (0x42000000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$
```

You specify that the shared libraries are found in the current directory (`./`). Then, after performing another `ldd`, the shared library is successfully found.

If you had tried to execute the application without having done this, a reasonable error message would have resulted, telling you that the shared library could not be found:

```
$ ./test
./test: error while loading shared libraries: libmyrand.so:
cannot find shared object file: No such file or directory.
$
```

Dynamically Loaded Libraries

The final type of library that this chapter explores is the dynamically loaded (and linked) library. This library can be loaded at any time during the execution of an application, unlike a shared object that is loaded immediately upon application startup. You'll build the shared object file as you did before, as follows:

```
$ gcc -fPIC -c initapi.c
$ gcc -fPIC -c randapi.c
$ gcc -shared initapi.o randapi.o -o libmyrand.so
$ su -
<provide your root password>
$ cp libmyrand.so /usr/local/lib
$ exit
```

In some distributions, such as Ubuntu, you have no root user. You can gain root access by typing `sudo su -` *and providing your user password, instead of* `su -`.

In this example, you'll move your shared library to a common location (`/usr/local/lib`, which is a standard directory for libraries) rather than relying on

the image and shared library always being in the same location (as was assumed in the previous example). Note that this library is identical to your original shared library. What is different is how the application deals with the library.

To copy the library to `/usr/local/lib`*, you must first gain root privileges. To do so, you use the* `su` *command (or* `sudo su -`*) to create a login shell for the root access.*

Now that you have your shared library re-created, how do you access this in a dynamic fashion from your test application? The answer is that you must modify your test application to change the way that you access the API. First, you can take a look at your updated test app (modified from Listing 7.4). Then you'll investigate how this is built for dynamic loading. The updated test application is shown in Listing 7.5. The next part of the chapter walks through this, identifying what changed from the original application, and then looks at the dynamically loaded (DL) API in more detail.

LISTING 7.5 Updated Test Application for Dynamic Linkage (on the CD-ROM at `./source/ch7/dynamic/test.c`)

```
1:      /*
2:       * Dynamic rand function test.
3:       *
4:       */
5:
6:      #include <stdlib.h>
7:      #include <stdio.h>
8:      #include <dlfcn.h>
9:      #include "randapi.h"
10:
11:     #define ITERATIONS      1000000L
12:
13:
14:     int main()
15:     {
16:       long  i;
17:       long  isum;
18:       float fsum;
19:       void  *handle;
20:       char  *err;
21:
22:       void (*initRand_d)(void);
23:       float (*getSRand_d)(void);
24:       int (*getRand_d)(int);
```

```
25:
26:         /* Open a handle to the dynamic library */
27:         handle = dlopen( "/usr/local/lib/libmyrand.so",
            RTLD_LAZY );
28:         if (handle == (void *)0) {
29:           fputs( dlerror(), stderr );
30:           exit(-1);
31:         }
32:
33:         /* Check access to the initRand() function */
34:         initRand_d = dlsym( handle, "initRand" );
35:         err = dlerror();
36:         if (err != NULL) {
37:           fputs( err, stderr );
38:           exit(-1);
39:         }
40:
41:         /* Check access to the getSRand() function */
42:         getSRand_d = dlsym( handle, "getSRand" );
43:         err = dlerror();
44:         if (err != NULL) {
45:           fputs( err, stderr );
46:           exit(-1);
47:         }
48:
49:         /* Check access to the getRand() function */
50:         getRand_d = dlsym( handle, "getRand" );
51:         err = dlerror();
52:         if (err != NULL) {
53:           fputs( err, stderr );
54:           exit(-1);
55:         }
56:
57:
58:         /* Initialize the random number API */
59:         (*initRand_d)();
60:
61:         /* Find the average of getRand(10) returns (0..9) */
62:         isum = 0L;
63:         for (i = 0 ; i < ITERATIONS ; i++) {
64:
65:           isum += (*getRand_d)(10);
66:
67:         }
```

```
68:
69:         printf( "getRand() Average %d\n", (int)(isum /
            ITERATIONS) );
70:
71:
72:         /* Find the average of getSRand() returns */
73:         fsum = 0.0;
74:         for (i = 0 ; i < ITERATIONS ; i++) {
75:
76:           fsum += (*getSRand_d)();
77:
78:         }
79:
80:         printf("getSRand() Average %f\n",
            (fsum/(float)ITERATIONS));
81:
82:         /* Close our handle to the dynamic library */
83:         dlclose( handle );
84:
85:         return;
86:       }
```

This code might appear a little convoluted given the earlier implementation, but it's actually quite simple after you understand how the DL API works. All that's really going on is that you're opening the shared object file using `dlopen`, and then assigning a local function pointer to the function within the shared object (using `dlsym`). This allows you then to call it from your application. When you're done, you close the shared library using `dlclose`, and the references are removed (freeing any used memory for the interface).

You make the DL API visible to you by including the `dlfcn.h` (DL function) header file. The first step in using a dynamic library is opening it with `dlopen` (line 27). You specify the library you need to use (`/usr/local/lib/libmyrand.so`) and also a single flag. Of the two flags that are possible (`RTLD_LAZY` and `RTLD_NOW`), you specify `RTLD_LAZY` to resolve references as you go, rather than immediately upon loading the library, which would be the case with `RTLD_NOW`. The function `dlopen` returns an opaque handle representing the opened library. Note that if an error occurs, you can use the `dlerror` function to provide an error string suitable for emitting to `stdout` or `stderr`.

While not necessary in this example, if you desired to have an initialization function called when the shared library was opened via `dlopen`, you can create a function called `_init` in the shared library. The `dlopen` function ensures that this `_init` function is called before `dlopen` returns to the caller.

Getting the references for the functions that you need to address is the next step. Take a look at the one that follows (taken from lines 34–39 of Listing 7.5).

```
34:        initRand_d = dlsym( handle, "initRand" );
35:        err = dlerror();
36:        if (err != NULL) {
37:          fputs( err, stderr );
38:          exit(-1);
39:        }
```

The process, as can be seen from this code snippet, is very simple. The API function `dlsym` searches the shared library for the function defined (in this case, the initialization function `"initRand"`). Upon locating it, a `(void *)` pointer is returned and stored in a local function pointer. This can then be called (as shown at line 59) to perform the actual initialization. You automatically check the error status (at line 35), and if an error string is returned, you emit it and exit the application.

That's really it for identifying the functions that you desire to call in the shared library. After you grab the `initRand` function pointer, you grab `getSRand` (lines 42–47) and then `getRand` (lines 49–55).

The test application is now fundamentally the same, except that instead of calling functions directly, you call them indirectly using the pointer-to-function interface. That's a small price to pay for the flexibility that the dynamically loaded interface provides.

The last step in the new test application is to close out the library. This is done with the `dlclose` API function (line 83). If the API finds that no other users of the shared library exist, then it is unloaded. The test image can be built, specifying `-dl` (for linking with `libdl`) as follows:

```
gcc -ldl test.c -o test
```

As was provided with `dlopen`, `dlclose` *provides a mechanism by which the shared object can export a completion routine that is called when the* `dlclose` *API function is called. The developer must simply add a function called* `_fini` *to the shared library, and* `dlclose` *ensures that* `_fini` *is called prior to* `dlclose return`.

And that's it! For the small amount of pain involved in creating an application that utilizes dynamically loaded shared libraries, it provides a very flexible environment that ultimately can save on memory use. Note also that it's not always necessary to make all dynamic functions visible when your application starts. You can instead make only those that are necessary for normal operation and load other dynamic libraries as they become necessary.

The dynamically loaded library API is very simple and is shown here for completeness:

```
void *dlopen( const char *filename, int flag );
const char *dlerror( void );
void *dlsym( void *handle, char *symbol );
int dlclose( void *handle );
```

How a library is made up depends upon what it's representing. The library should contain all functions that are necessary for the given problem domain. Functions that are not specifically associated with the domain should be excluded and potentially included in another library.

Utilities

Now take a look at some of the other utilities that are useful when you are creating static, shared, or dynamic libraries.

file

The `file` utility tests the file argument for the purposes of identifying what it is. This utility is very useful in a number of different scenarios, but in this case it provides you with a small amount of information about the shared object. Take a look at an interactive example:

```
$ file /usr/local/lib/libmyrand.so
/usr/local/lib/libmyrand.so: ELF 32-bit LSB shared object,
Intel 80386, version 1 (SYSV), not stripped
$
```

So, by using `file`, you see that the shared library is a 32-bit ELF object for the Intel 80386 processor family. It has been defined as "not stripped," which simply means that debugging information is present.

size

The `size` command provides you with a very simple way to understand the `text`, `data`, and `bss` section sizes for an object. An example of the `size` command on your shared library is shown here:

```
$ size /usr/local/lib/libmyrand.so
   text    data     bss     dec     hex filename
   2013     264       4    2281     8e9 /usr/local/lib/libmyrand.so
$
```

nm

To dig into the object, you use the `nm` command. This commands permits you to look at the symbols that are available within a given object file. Take a look at a simple example using `grep` to filter your results:

```
$ nm -n /usr/local/lib/libmyrand.so | grep " T "
00000608 T _init
0000074c T initRand
00000784 T getSRand
000007be T getRand
00000844 T _fini
$
```

In this example, you use `nm` to print the symbols within the shared library, but then only emit those with the tag `" T "` to `stdout` (those symbols that are part of the `.text` section, or code segments). You also use the `-n` option to sort the output numerically by address, rather than the default, which is alphabetically by symbol name. This gives you relative address information within the library; if you wanted to know the specific sizes of these `.text` sections, you could use the `-S` option, as follows:

```
$ nm -n -S /usr/local/lib/libmyrand.so | grep " T "
00000608 T _init
0000074c 00000036 T initRand
00000784 0000003a T getSRand
000007be 00000050 T getRand
00000844 T _fini
$
```

From this example, you can see that the `initRand` is located at relative offset 0–74c in the library and its size is 0–36 (decimal 54) bytes. Many other options are available; the `nm` manpage provides more detail on this.

objdump

The `objdump` utility is similar to `nm` in that it provides the ability to dig in and inspect the contents of an object. Take a look at some of the specialized functions of `objdump`.

One of the most interesting features of `objdump` is its ability to disassemble the object into the native instruction set. Here's an excerpt of `objdump` performing this capability:

```
$ objdump -disassemble -S /usr/local/lib/libmyrand.so
...
0000074c <initRand>:
  74c:   55                      push   %ebp
  74d:   89 e5                   mov    %esp,%ebp
  74f:   53                      push   %ebx
  750:   83 ec 04                sub    $0x4,%esp
  753:   e8 00 00 00 00          call   758 <initRand+0xc>
  758:   5b                      pop    %ebx
  759:   81 c3 f8 11 00 00       add    $0x11f8,%ebx
  75f:   83 ec 0c                sub    $0xc,%esp
  762:   6a 00                   push   $0x0
  764:   e8 c7 fe ff ff          call   630 <_init+0x28>
  769:   83 c4 10                add    $0x10,%esp
  76c:   89 45 f8                mov    %eax,0xfffffff8(%ebp)
  76f:   83 ec 0c                sub    $0xc,%esp
  772:   ff 75 f8                pushl  0xfffffff8(%ebp)
  775:   e8 d6 fe ff ff          call   650 <_init+0x48>
  77a:   83 c4 10                add    $0x10,%esp
  77d:   8b 5d fc                mov    0xfffffffc(%ebp),%ebx
  780:   c9                      leave
  781:   c3                      ret
  782:   90                      nop
  783:   90                      nop
...
$
```

In addition to `-disassemble` (to disassemble to the native instruction set), this example also specified `-S` to output interspersed source code. The problem is that this object is compiled to exclude this information. You can easily fix this as follows, by adding `-g` to the compilation process.

```
$ gcc -c -g -fPIC initapi.c
$ gcc -c -g -fPIC randapi.c
$ gcc -shared initapi.o randapi.o -o libmyrand.so
$ objdump -disassemble -S libmyrand.so
...
```

```
00000790 <initRand>:
  *
  */
void initRand()
{
  790:   55                      push   %ebp
  791:   89 e5                   mov    %esp,%ebp
  793:   53                      push   %ebx
  794:   83 ec 04                sub    $0x4,%esp
  797:   e8 00 00 00 00          call   79c <initRand+0xc>
  79c:   5b                      pop    %ebx
  79d:   81 c3 fc 11 00 00       add    $0x11fc,%ebx
  time_t seed;
  seed = time(NULL);
  7a3:   83 ec 0c                sub    $0xc,%esp
  7a6:   6a 00                   push   $0x0
  7a8:   e8 c7 fe ff ff          call   674 <_init+0x28>
  7ad:   83 c4 10                add    $0x10,%esp
  7b0:   89 45 f8                mov    %eax,0xfffffff8(%ebp)
  srand( seed );
  7b3:   83 ec 0c                sub    $0xc,%esp
  7b6:   ff 75 f8                pushl  0xfffffff8(%ebp)
  7b9:   e8 d6 fe ff ff          call   694 <_init+0x48>
  7be:   83 c4 10                add    $0x10,%esp
  return;
}
  7c1:   8b 5d fc                mov    0xfffffffc(%ebp),%ebx
  7c4:   c9                      leave
  7c5:   c3                      ret
  7c6:   90                      nop
  7c7:   90                      nop
...
$
```

Having compiled the source code with `-g`, you now have the ability to understand the C source to machine code mapping.

Numerous other capabilities are provided with `objdump`. The GNU/Linux manpage lists the plethora of other options.

ranlib

The `ranlib` utility is one of the most important utilities when creating static libraries. This utility creates an index of the contents of the library and stores it in the library file itself. When this index is present in the library, the linking stage of

building an image can be sped up considerably. Therefore, the `ranlib` utility should be performed whenever a new static library is created. An example of using `ranlib` is shown here:

```
$ ranlib libmyrand.a
$
```

Note that the same thing can be performed using the `ar` command with the `-s` option, as follows:

```
$ ar -s libmyrand.a
$
```

Printing out the archive can be accomplished in either of these options in `nm`:

```
$ nm –s libmyrand.a
$ nu –print-armap libmyrand.a
```

Summary

This chapter explored the creation and use of program libraries. Traditional static libraries were discussed first, followed by shared libraries, and finally dynamically loaded libraries. The chapter also investigated source code to demonstrate the methods for creating libraries using the `ar` command as well as using libraries with `gcc`. Finally, it discussed a number of library-based utilities, including `ldd`, `objdump`, `nm`, `size`, and `ranlib`.

Dynamic Library APIs

```
#include <dlfcn.h>
void *dlopen( const char *filename, int flag );
const char *dlerror( void );
void *dlsym( void *handle, char *symbol );
int   dlclose( void *handle );
```

8 Building Packages with `automake/autoconf`

by Curtis Nottberg

In This Chapter

- `make` Review
- Introduction to the GNU Autotools
- Converting a Project to Use Autotools
- Quick Introduction to `automake`
- Quick Introduction to `autoconf`

Introduction

The standard GNU `make` utility eases many of the burdens associated with building an executable from multiple source files. It enables incremental building of the source and allows the commands and processes needed to maintain a source package to be collected in a single location. GNU `make` is excellent at implementing the steps needed to build a moderately complex project. However, GNU `make` starts to become cumbersome as projects grow in complexity. Examples of factors that cause Makefile maintenance to become cumbersome are these:

- Projects with a large number of files that have varied build requirements
- Dependencies on external libraries
- A desire to build in a multiplatform environment
- Installing built software in multiple environments
- Distributing a source package

The solution to these complexities is to move up one level and automatically generate the appropriate Makefiles to build the project. This allows GNU `make` to focus on the things it is good at while still allowing the capability to configure for the current build environment. The GNU Autotools are an attempt to provide this next level of build functionality on top of the GNU `make` utility.

The cross-platform GNU Build System (which is made up of the `automake`, `autoconf`, *and* `libtool` *packages) was begun in 1991 by David Mackenzie of the Free Software Foundation.*

A Sample Project

The examples in this chapter show various ways to build a project consisting of four source files. Two of the source files are used to build a library, and the other files build an application that uses the library (see Figure 8.1).

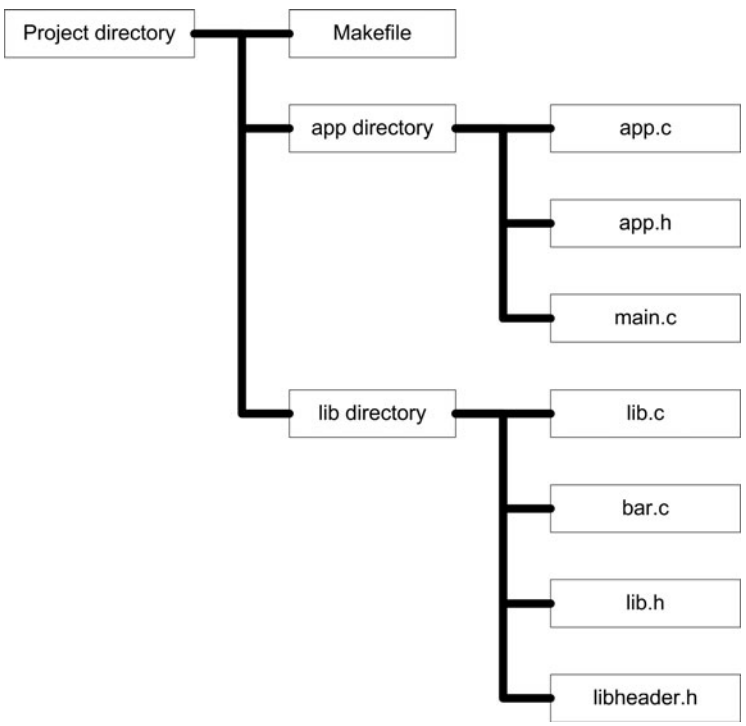

FIGURE 8.1 Directory structure of sample project.

A Makefile Solution

Listing 8.1 shows a simple Makefile that builds the library and an application that uses the library. This Makefile is used to provide a basis for comparing how Autotools would build the same project. Keep in mind that the example considered here is very simple. The application of automake/autoconf to this project might seem like more work than payoff, but as the project grows, the payback from using the Autotools increases.

LISTING 8.1 Simple Makefile to Build Example (on the CD-ROM at ./source/ch8/Makefile.simple)

```
1:    VPATH= lib app
2:
3:    LIBSRC= lib.c bar.c
4:    LIBOBJ= $(LIBSRC:.c=.o)
5:
6:    APPSRC= main.c app.c
7:    APPOBJ= $(APPSRC:.c=.o)
8:
9:    CFLAGS=
10:    INCLUDES= -I ./lib
11:
12:    all: libexp.a appex
13:
14:    %.o:%.c
15:        $(CC) -c $(CFLAGS) $(INCLUDES) -o $@ $
16:
17:    libexp.a: $(LIBOBJ)
18:        $(AR) cru libexp.a $(LIBOBJ)
19:
20:    appex: $(APPOBJ) libexp.a
21:        $(CC) -o appex $(APPOBJ) -lexp -L .
```

Line 1 of the listing sets the VPATH variable. The VPATH variable specifies search paths that are used by the make utility to find the source files for the build rules. The VPATH capability of make allows the use of a single Makefile to reference the source files in the lib and app subdirectories. Lines 3 through 7 create source and object file lists for use by the build rules. Notice that the file list doesn't include paths because this is taken care of by the VPATH search variable. Lines 9 and 10 set up the necessary compiler flags to perform the build. Line 12 sets up the default build target to build both the library and application. Lines 14 and 15 define the rules used to turn C-source files into object files. Lines 17 and 18 describe how to build the library. Finally, lines 20 and 21 describe how to build the application.

The simplified Makefile is missing a few features that would need to be included in a real build system to make it useable: a clean target, dependency tracking on included header files, a method for installing the generated binaries, and so forth. The Autotools implementation provides many of these missing features with only a little bit of additional work when compared to the effort needed to create the simplified Makefile.

A Simple Implementation Using Autotools

The initial implementation using the Autotools requires the creation of five files to replace the simple Makefile described in Listing 8.1. Although this seems like a lot of files to replace a single file, each of the replacement files is generally simpler. Both the simple Makefile and the Autotool files contain roughly the same information, but Autotools chooses to distribute the information differently in the project's directory structure. Figure 8.2 illustrates the directory structure from Figure 8.1 with the addition of the Autotools files.

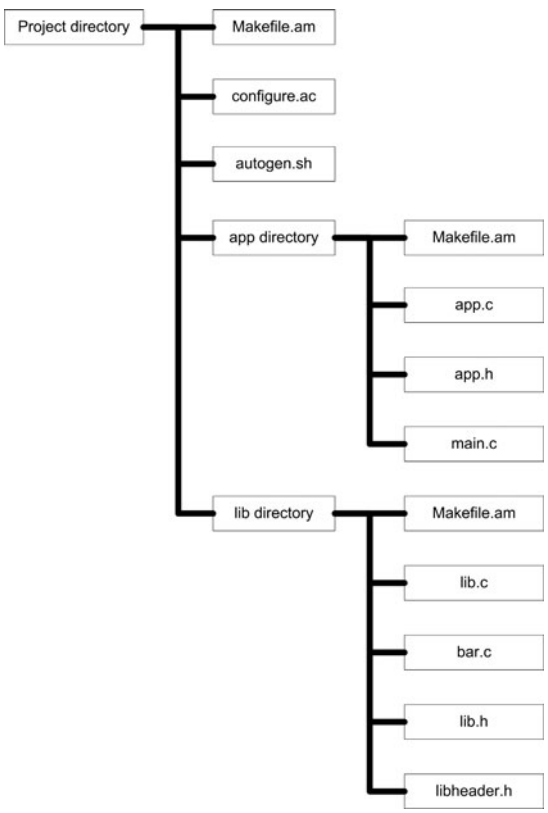

FIGURE 8.2 Directory structure of sample project with Autotool files.

The additional files added to support a simple Autotools project are these:

`autogen.sh`: A shell script to run Autotools to generate the build environment
`configure.ac`: The input file for the `autoconf` tool
`Makefile.am`: The top-level Makefile template
`app/Makefile.am`: The Makefile template for `appexp` executable
`lib/Makefile.am`: The Makefile template for building the `libexp.a` library

These files describe the intended build products and environment to Autotools. Autotools takes this input and generates a build environment template that is further configured on the build system to generate the final set of Makefiles. Assuming that you are developing and building on the same machine, the following commands should configure and build the sample project:

```
# ./autogen.sh
# ./configure
# make
```

Running the `autogen.sh` script executes the Autotool utilities to convert the input files into a build environment template that can be configured on the host system. Executing the `configure` script causes the build environment template to be customized for the build machine. The output of the `configure` script is a set of GNU Makefiles that can be used to build the system. Executing the `make` command in the root directory causes both the library and application to be built.

An examination of `autogen.sh` should be the starting point for understanding how this process works. Listing 8.2 shows a very simple `autogen.sh` script that just executes the Autotools utilities. Generally the `autogen.sh` script in a real project is much more complicated to first check that the Autotools exist and are of the appropriate version. To find an example of a more complex `autogen.sh` script, you should examine this file in the source repositories of your favorite open source project.

LISTING 8.2 Simple `autogen.sh` Script (on the CD-ROM at `./source/ch8/autogen.sh`)

```
1:    #!/bin/sh
2:    # Run this to generate all the initial makefiles, etc.
3:
4:    aclocal
5:    libtoolize –automake
6:    automake -a
7:    autoconf
```

Line 1 indicates the shell to use when running this script. Line 4 runs the `aclocal` utility. The `aclocal` utility creates the local environment needed by the `automake` and `autoconf` tools to work. Specifically `aclocal` makes sure the `m4` macro environment that `automake` and `autoconf` use to implement their functionality is set up appropriately. Line 5 executes the `libtoolize` utility, which enables the `libtool` functionality in `automake`. The `libtool` functionality is discussed in this chapter. Line 6 executes the `automake` utility, which turns the `Makefile.am` files into `Makefile.in` files. This operation is discussed more in the next section. Line 7 executes the `autoconf` utility that takes the `configure.ac` input file and turns it into a pure shell script named `configure`.

automake

The input to the `automake` utility is a series of `Makefile.am` files that describe the targets to be built and the parameters used to build them. The `automake` utility transforms the `Makefile.am` files into `makefile.in` files. The `Makefile.in` file is a GNU `make` format file that acts as a template that the configure script transforms into the final `Makefile`. `automake` has built-in support for building binaries and libraries and with the support of `libtool` can also be used to build shared libraries. The sample project required three separate `automake` files: one in the root directory and one for each subdirectory. Now it's time to take a look at the root `Makefile.am` to see how `automake` handles subdirectories (Listing 8.3).

LISTING 8.3 Listing of the Root `Makefile.am` (on the CD-ROM at `./source/ch8/Makefile.am`)

```
1:  SUBDIRS = lib app
```

The contents of the root `Makefile.am` simply indicate that all the work for this project is done in the subdirectories. Line 1 tells the `automake` utility that it should descend into the subdirectories and process the `Makefile.am` files it finds there. The ordering of directories in the `SUBDIRS` variable is significant; the subdirectories are built in the left to right order specified in the subdirectories list. The sample project uses this to ensure that the `lib` directory is built before the `app` directory, a requirement of the sample project because the application is dependent on the library being built first.

Now it's time to move on to the `lib/Makefile.am` file that shows `automake` how to build the `libexp.a` library (Listing 8.4).

LISTING 8.4 Listing of lib/Makefile.am (on the CD-ROM at ./source/ch8/lib/Makefile.am)

```
1:      lib_LIBRARIES = libexp.a
2:      libexp_a_SOURCES = bar.c lib.c
```

Line 1 is a list of the static libraries to be built in this directory. In this case the only library being built is libexp.a. The syntax of line 1 is more complex than it first appears. The lib_LIBRARIES variable name indicates two pieces of information. The lib portion indicates that when the library is installed it will be put in the lib directory. The LIBRARIES portion indicates that the listed targets should be built as static libraries. Line 2 lists the source files that go into building the libexp.a static library. Again automake uses the format of the variable name to provide the association between both the library that the variable applies to and the content of the variable. The libexp_a portion of the name indicates that this variable's value applies to building libexp.a. The SOURCES portion of the name implies that the value of this variable is a space-separated list of source files. The app/Makefile.am file (Listing 8.5) is very similar to the one in the lib directory but includes a few additional variables to take care of using the libexp.a library that was previously built in the lib directory.

LISTING 8.5 Listing of app/Makefile.am (on the CD-ROM at ./source/ch8/app/Makefile.am)

```
1:      bin_PROGRAMS = appexp
2:      appexp_SOURCES = app.c main.c
3:      appexp_LDADD = $(top_builddir)/lib/libexp.a
4:      appexp_CPPFLAGS = -I $(top_srcdir)/lib
```

Line 1 of Listing 8.5 should look similar to line 1 in Listing 8.4 in that it is a list of things to be built. In this case the bin_PROGRAMS variable name indicates to automake that the result is installed in the bin directory and listed targets should be built as executables. The appexp_ prefix on the variable in lines 2 through 4 indicates that these variables apply to building the appexp executable. Line 2 has the SOURCES variable that lists the source files that are to be compiled into the appexp executable. Line 3 specifies the LDADD variable, which are things that are to be included during linking. In this example, the LDADD variable is used to add the library that was previously built in the lib directory. The $(top_builddir) is set by the configure script when it is run and provides a mechanism for Makefiles to reference the build directories in a relative manner. Line 4 specifies the CPPFLAGS variable that is passed

to the preprocessor when it runs. This variable should contain the `-I include` paths and the `-D` defines that would normally be passed to the preprocessor in a Makefile. In this example it is being used to get access to the library header file contained in the `lib` directory. The `$(top_srcdir)` variable is set by the `configure` script; it provides a mechanism for Makefiles to reference source files in a relative manner.

autoconf

The `autoconf` utility converts the `configure.ac` input file into a shell script named `configure`. The `configure` script is responsible for collecting information about the current build system and using the information to transform the `Makefile.in` template files into the Makefiles used by the GNU `make` utility. The `configure` script performs the transformation by replacing occurrences of configuration variables in the `Makefile.in` template file with values for those variables as determined by the `configure` script. The `configure.ac` input file contains macros that describe the types of configuration checks the `configure` script should perform when it is run. The `configure.ac` for the sample project illustrates the very simple series of checks needed to compile C-files and create static libraries (Listing 8.6).

LISTING 8.6 Listing of `configure.ac` (on the CD-ROM at `./source/ch8/configure.ac`)

```
1:  dnl Process this file with autoconf to produce a configure script
2:  AC_PREREQ(2.53)
3:  AC_INIT(app)
4:  AM_INIT_AUTOMAKE(appexp, 0.1.00)
5:  AC_PROG_CC
6:  AC_PROG_RANLIB
7:  AC_OUTPUT(app/Makefile lib/Makefile Makefile)
```

Line 1 illustrates the comment format used by `autoconf`. Line 2 is a macro defined by `autoconf` to ensure that the version of the `autoconf` utility being used to create the `configure` script is new enough. This macro results in a check to make sure that `autoconf` utility has a version number equal to or greater than 2.53. If the version isn't recent enough, an error is generated, and the `configure` script is not generated. Line 3 is a required `autoconf` macro that must be called before any other macros are invoked; it gives `autoconf` a chance to initialize itself and parse its command-line parameters. The parameter to `AC_INIT` is a name for the project. Line 4 is the initialization macro for `automake`. `autoconf` can be used independently of `automake`, but if they are to be used together, then the `AM_INIT_AUTOMAKE` macro is required in the project's `configure.ac` file. Line 5 and 6 are the first macros that

actually check for tools used during the make process. Line 5 indicates that the project causes the configure script to find and prepare the Makefiles to use the C compiler. Line 6 does the checks to find the tools needed to build static libraries. Line 7 is the other required macro that must exist in a configure.ac file. This macro indicates the output files that should be generated by the configure script. When the configure script is ready to generate its output files, it iterates through the files in the AC_OUTPUT macro and looks for a corresponding file with an .in suffix. It then performs a substitution step on the .in file to generate the output file.

The configure Script

The output of the autoconf utility is a shell script named configure. The sample configure.ac in Listing 8.7 generates a configure script with approximately 4,000 lines when run through the autoconf utility. Executing the configure script collects the required configuration information from the executing system and generates the appropriate Makefiles by performing a substitution step on the Makefile.in files generated by automake. Listing 8.7 examines the output of running the configure script generated by the example.

LISTING 8.7 Output from the Sample configure Script

```
1:     checking for a BSD-compatible install... /usr/bin/install -c
2:     checking whether build environment is sane... yes
3:     checking for gawk... gawk
4:     checking whether make sets $(MAKE)... yes
5:     checking for gcc... gcc
6:     checking for C compiler default output file name... a.exe
7:     checking whether the C compiler works... yes
8:     checking whether we are cross compiling... no
9:     checking for suffix of executables... .exe
10:     checking for suffix of object files... o
11:     checking whether we are using the GNU C compiler... yes
12:     checking whether gcc accepts -g... yes
13:     checking for gcc option to accept ANSI C... none needed
14:     checking for style of include used by make... GNU
15:     checking dependency style of gcc... gcc3
16:     checking for ranlib... ranlib
17:     configure: creating ./config.status
18:     config.status: creating app/Makefile
19:     config.status: creating lib/Makefile
20:     config.status: creating Makefile
21:     config.status: executing depfiles commands
```

Lines 1 through 4 are checks that occur to ensure that the build environment has the appropriate tools to support the `make` process. Lines 5 through 15 are checks generated by the `AC_PROG_CC` macro that locate and ready the compiler toolchain for processing C-source code. Line 16 is a check generated by the `AC_PROG_RANLIB` macro to ensure that the `ranlib` utility exists for generating static libraries. Lines 18 through 20 indicate that the substitution step to turn the `Makefile.in` templates into the actual Makefiles is occurring.

After the `configure` script has completed successfully, then all of the Makefiles needed to build the project should have been successfully created. Typing `make` in the root directory at this point should build the project.

The Generated Makefiles

The generated Makefiles have a number of nice characteristics that were lacking in the simple Makefile of Listing 8.1, such as:

- Automatic dependency tracking. For example, when a header file is modified, only the source files that are affected are rebuilt.
- A clean target that cleans up all the generated output.
- The automated ability to install the generated binaries into the appropriate system directories for execution.
- The automated ability to generate a distribution of the source code as a compressed tar file.

The generated Makefiles have numerous predefined targets that allow the user to invoke these capabilities. The following list examines the common targets used in the `automake`-generated Makefiles:

`make:` The default target. This causes the project binaries to be built.

`make clean:` The clean target. This removes all of the generated build files so that the next call to `make` rebuilds everything.

`make distclean:` This target removes all generated files, including those generated by the `configure` script. After using this target, the `configure` script needs to be run again before another build can take place.

`make install:` This target moves the generated binaries and supporting files into the system directory structure. The location of the installation can be controlled with the `-enable-prefix` parameter that can be passed to the `configure` script when it is run.

`make dist`: This target generates a `.tar.gz` file that can be used to distribute the source code and build setup. Included in the tarball is all of the source files, the `makefile.in` files, and the `configure` script.

Just looking at the default targets provided by the standard `automake` Makefile should indicate some of the power that exists in using the Autotools to generate the Makefiles for your project. The initial setup to use the Autotools can be a bit cumbersome, but after the infrastructure is in place, then future updates to the files are very incremental, and the payback is large compared to implementing the same capabilities by hand in a developer-maintained Makefile.

Summary

This chapter presented the GNU Autotools by illustrating how they can be used to build a simple project. The example makes little use of the advanced features of `automake` and `autoconf`, but it should provide a good illustration of the big-picture concepts needed to understand how the Autotools work. The GNU Autotools provide a wealth of features that are quite useful to larger software projects, and the effort of integrating them into your project should be expended early on. The downside of the tools is that they can be somewhat difficult to employ properly, and the documentation for them is a bit arcane. On balance, the uses of the Autotools are well worth the effort, but expect to put a little bit of time into getting things working correctly. One of the best ways to learn about the more advanced usage of `automake` and `autoconf` is to look at the existing implementations used in current open source projects. The simple example presented in this chapter should provide the basis needed to examine and learn from the more complex use of the GNU Autotools found in the larger open source projects.

This page intentionally left blank

9 Source Control in GNU/Linux

In This Chapter

- Source Control Models
- CVS Overview
- Subversion Overview
- GIT Overview

Introduction

Software is typically developed as an evolutionary process, and when it's developed by a group of programmers, some method is necessary to control access, track changes, and establish relationships between the elements that make up the software. This chapter explores source control systems (also known as software configuration management systems or revision control systems). Numerous options exist for source control that offer different capabilities while providing the core functionality that's needed. This chapter explores these models and a number of the GNU/Linux approaches that are available.

Defining Source Control

Source, or revision, control is the art of managing software through its process of development. Source control systems come in a variety of forms, but each provides a set of basic operations. These operations are as follows:

- The ability to share source code and other files with one or more developers for collaboration
- The ability to track changes in files
- The ability to merge changes between files
- The ability to group files together

These are the most basic operations of a source control system, and they are the core of the abilities. Other capabilities include showing differences in files. As you'll soon learn, source control systems implement this functionality in different ways. Some of these differences are explored in terms of the architectures of the notable systems.

Source Control Paradigms

Source control systems can be categorized in a number of ways, but the two most useful are the repository architecture and the history model. This section introduces these characteristics and some of the differences in the approaches.

Repository Models

The repository is where source is managed, and it also defines the way that users interact with the repository (the protocol). In general you have two fundamentally different architectures, the centralized repository and the distributed repository. This section reviews both, and the hybrid that combines the strengths of both.

Centralized Architecture

The centralized architecture, as the name implies, is one where a central repository is used in a star architecture with sandboxes feeding it from the periphery. Each developer (client) creates a sandbox for development and then feeds these changes upstream to the central repository (server). This is shown in Figure 9.1.

Upstream in this context means that developers each have sandboxes and coordinate their changes through a single repository. Centralized repositories have the concept of branches to allow multiple developers to coordinate on shared changes off the tip (head of the repository).

Distributed and Hybrid Architecture

The distributed architecture has not just a single repository but multiple repositories. Developers can take from a central repository, but then operate independently. But what differs in this from the centralized architecture is that developers' sandboxes can become repositories in their own right, merging changes from others. In

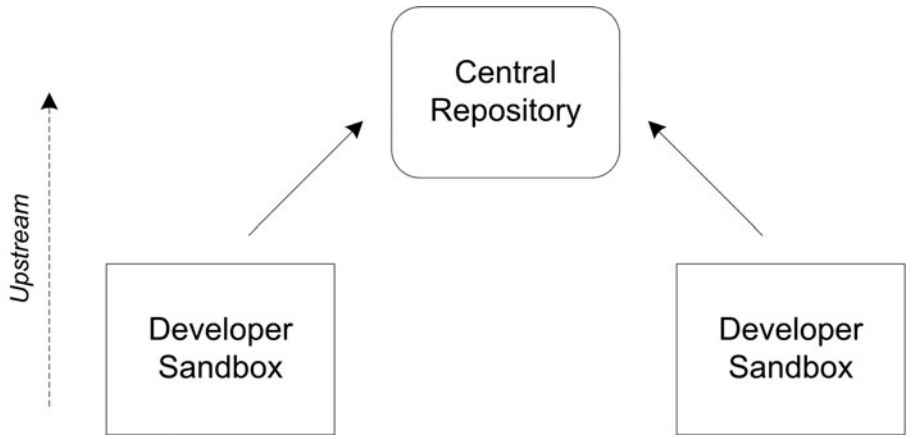

FIGURE 9.1 The centralized architecture for source management.

this way, you have a hierarchy of development sandboxes that all merge upstream toward one or more repositories (see Figure 9.2).

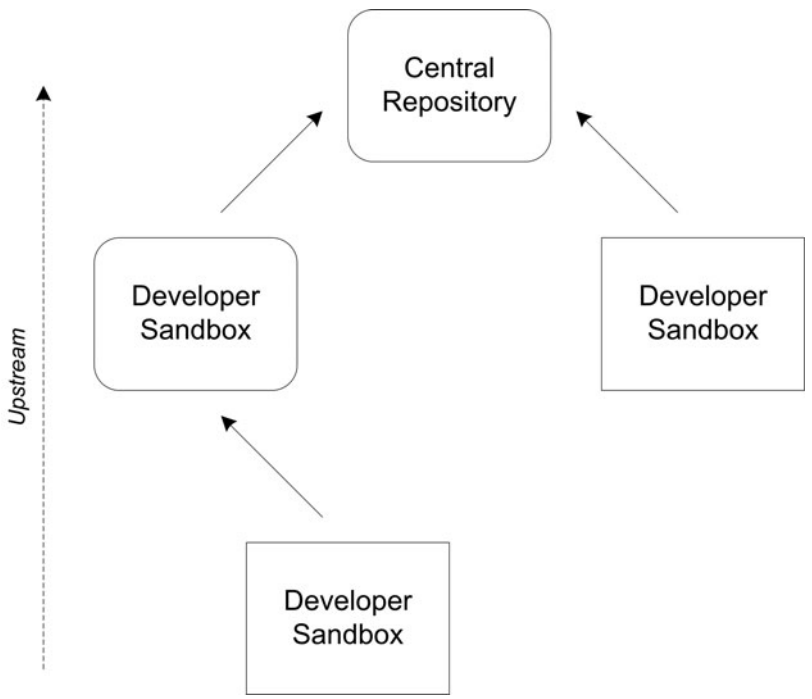

FIGURE 9.2 The distributed architecture for source management.

Note that this hierarchy allows many developers to work on different aspects of the system from their own repository, merging changes upstream (or downstream) when needed.

The hybrid architecture mixes the two, allowing a central repository for all changes, but with many developer repositories that feed into it.

Revision Model

The revision or history architecture defines the way that changes are stored and managed within the repository, including the way the files change over time. Two specific architectures are explored here, the snapshot architecture and the change-set architecture.

Snapshot Architecture

The snapshot architecture is the simplest because no history is required to be maintained. In this model, complete changes are stored to files rather than any differences between them. The snapshot model stores all files in their entirety, so space is not utilized well, but the model is very fast because little processing is required to retrieve a particular version.

Change-Set Architecture

The change-set (or delta) architecture stores files based upon their differences. Instead of storing a complete file for each version, only the differences of the files are stored (a revision specifies deltas instead of the entire file). This approach is much more efficient for space, but is less efficient for performance because processing can be required to identify a specific version of a file (applying deltas, etc.).

Useful Source Control Tools

In the open source domain, you find many different options for source management. This chapter covers three of the most important: CVS, Subversion, and the newcomer Git.

CVS

The Concurrent Versions System (CVS) is one of the older source control systems. Its name came from the goal to develop and collaborate on software concurrently without worrying about one developer overwriting changes of another. CVS was created in the mid-1980s by Dick Grune and remains one of the most popular open source version control systems available. Its heritage comes from the older Revision Control System (RCS).

This section explores CVS and how to use it to manage a simple project. The intent is to provide a hands-on introduction to the basic capabilities of CVS and explore its important capabilities.

From a model perspective, CVS uses a centralized repository and a snapshot approach to maintain history. CVS is implemented in the C language and is released under the GPL.

Setting Up a New CVS Repository

Creating a new CVS repository is done very simply with the `init` command of CVS. You first define the root of the CVS repository by setting it as an environment variable (`CVSROOT`). This could also be done on the command line with the `–d` option, but because you will use the repository often, you use an environment variable. In this case, I use my home directory for the repository, but for sharing, you might want to put this in a shared directory.

```
$ export CVSROOT=/home/mtj/cvs_repo
$ cvs init
```

When this command is complete, a new subdirectory is created that contains another directory (`./cvs_repo/CVSROOT`). This contains information about the repository as well as working data (users, notification lists, etc.).

Adding to the Repository

Now that you have your repository, it's time to add to it. You create a working directory (`/home/mtj/work`) and then add a file.

```
$ mkdir work
$ cd work
```

Next you create your file (shown in Listing 9.1) with your favorite editor.

LISTING 9.1 File `./work/test.c` Ready for CVS Checkin

```
1:  #include <stdio.h>
2:
3:  int main()
4:  {
5:    print("test string.\n");
6:
7:    return 0;
8:  }
```

Checking the file into your repository is done first with the CVS `import` command. With this command you enter your working directory (`./work`) and import as follows:

```
$ cvs import –m "my import" my_files mtj start
N my_files/test.c

No conflicts created by this import

$
```

With the `import` command, you specify a comment for the `import` (specified with the `–m` option) and then three additional arguments. The first argument is the module that you want to define (where the files are to be placed). The second argument is the vendor tag, and the last is the release tag. You can define the last two tags however you like.

Manipulating Files in the Repository

At this stage, you have a repository created and a new module (`my_files`) that contains a single file. Now it's time to check out this repository in another subdirectory (`./work2`). We use the `checkout` CVS command to get the `my_files` module.

```
$ mkdir work2 ; cd work2
$ cvs checkout my_files
cvs checkout: Updating my_files
U my_files/test.c
$ ls my_files
CVS  test.c
$
```

Note that from Listing 9.1, if you tried to compile the file, you would end up with an error (undefined reference) because of the misspelling of `printf`. You can correct this mistake and can then view the changes to the repository using the CVS `diff` command.

```
$ cvs diff test.c
Index: test.c
===================================================================
RCS file: /home/mtj/cvs_repo/my_files/test.c,v
retrieving revision 1.1.1.1
diff -r1.1.1.1 test.c
5c5
```

```
<   print("test string.\n");
—-
>   printf("test string.\n");
$
```

At this point, the local sandbox contains the change. The next step is to commit this change to the repository. This is done using the `commit` command. Note that you can specify no options for `commit`, which would affect all files in the directory (and subdirectory). In the following example, you are currently located in the `./my_files` subdirectory.

```
$ cvs commit
/home/mtj/cvs_repo/my_files/test.c,v  <—  test.c
new revision: 1.2; previous revision 1.1
$
```

Note the difference in the revision numbers. Each time a new file is committed, it receives a new revision number (in this case from 1.1 to 1.2). These revision tags can be used to identify specific revisions of files; for example, you can check out the specific revision of a file as follows:

```
$ cvs checkout —r 1.1 my_files/test.c
U my_files/test.c
$
```

Merging Changes from the Repository

If you want to bring changes from the repository into your sandbox (for example, those committed by another developer), you use the CVS `update` command. This very simply is performed as follows:

```
$ cvs update my_files
cvs update: Updating my_files
$
```

Other Useful Commands

You can also perform `diffs` on revision of files. In the following example, two revisions of a file are defined, along with the `diff` command to provide the differences. Note the greater than and less than symbols in the example that follows. This example shows < for revision 1.1 and > for revision 1.2 (how the lines that differ appear in each revision).

```
$ cvs diff –r 1.1 –r 1.2 my_files/test.c
Index: my_files/test.c
===================================================================
RCS file: /home/mtj/cvs_repo/my_files/test.c,v
retrieving revision 1.1
retrieving revision 1.2
diff -r1.1 -r1.2
5c5
<   print("test string.\n");
—-
>   printf("test string.\n");
$
```

The `log` command can be used to show the history of a file in the repository. Given the changes so far, you can view the file's log as follows:

```
mtj@camus:~$ cvs log my_files/test.c
RCS file: /home/mtj/cvs_repo/my_files/test.c,v
Working file: my_files/test.c
head: 1.2
branch:
locks: strict
access list:
symbolic names:
        start: 1.1.1.1
        mtj: 1.1.1
keyword substitution: kv
total revisions: 3;     selected revisions: 3
description:
————————————
revision 1.2
date: 2007-12-30 05:23:16 +0000;  author: mtj;  state: Exp;  lines: +1 -1
Correct printf.
————————————
revision 1.1
date: 2007-12-30 04:35:59 +0000;  author: mtj;  state: Exp;
branches:  1.1.1;
Initial revision
————————————
revision 1.1.1.1
date: 2007-12-30 04:35:59 +0000;  author: mtj;  state: Exp;  lines: +0 -0
my import
=============================================================================
$
```

Repository Tagging

Tagging is an important part of source control because it allows you to freeze the repository with a given name for all time. This allows you to create a version of your software and later extract that particular set of files to rebuild. You can tag your current repository with the following command:

```
$ cvs rtag –b release_1 my_files
cvs rtag: Tagging my_files
$
```

This tags the files in the module and does not require a `commit` to be performed. The `–b` option specifies that you are creating a branch (an epoch of the files at the current time in the repository). Another tagging command (`cvs tag`) tags the local repository, and requires a `commit` of the files to take effect. Later, if you wanted to retrieve that particular revision of files, you can issue the following command:

```
$ cvs checkout –r release_1 my_files
U my_files/test.c
$
```

Note that any changes made to the files at this time would be committed back to the branch, and not to the head of the repository. This allows a branch to be created at any point in time and for changes to be applied only to it. You can see the current revision of the files using the `cvs status` command:

```
$ cvs status my_files/test.c
===================================================================
File: test.c            Status: Up-to-date

   Working revision:    1.2     Sun Dec 30 05:23:16 2007
   Repository revision: 1.2     /home/mtj/cvs_repo/my_files/test.c,v
   Sticky Tag:          release_1 (branch: 1.2.2)
   Sticky Date:         (none)
   Sticky Options:      (none)

$
```

Note the sticky tag for the file that indicates the branch name (`release_1`). The `status` command is a useful way to know which version of files is currently checked out.

Removing Files from the Repository

If desired, files can be removed from the repository (though not completely). When a file (or directory) is removed in CVS, it is placed into the *attic*, which is a special directory containing deleted files (they're never deleted completely). The `cvs remove` command is used to remove a file.

```
$ cvs remove my_files/test.c
$ cvs commit
/home/mtj/cvs_repo/my_files/test.c,v  <--  test.c
new revision: delete; previous revision: 1.2
$
```

You need to understand the context in which files are removed. Note in the preceding example that you have removed a file from your branch (1.2). This removes the 1.2 file, but all other revisions (the original `commit` and `update` for the misspelled `printf`) are still present.

CVS Summary

CVS remains one of the most stable and popular choices for source control. It's a standard part of most Linux distributions, so you typically find it without any additional installs. Because CVS is ubiquitous, it's something you probably find in a project near you, so it's worth your time to learn and master.

Subversion

Subversion (or SVN) is a new version control system that was designed to be an alternative to CVS. In fact, many high-profile open source projects have already migrated from CVS to SVN (such as the Apache Web server project). In 2007, Subversion was defined as the leader in standalone software configuration management tools [Wikipedia:SVN].

As with CVS, this chapter explores SVN for the purposes of creating a repository and manipulating files within it.

From a model perspective, SVN uses a centralized repository and a snapshot and change-set approach to maintain history. SVN is implemented in the C language and is released under the Apache software license.

Setting Up a New SVN Repository

To create a new SVN repository, you use the `svnadmin` command. This is different from the typical subversion command `svn`, which you can use for the more common operations.

Unlike CVS, you need to start the SVN server, which is done very simply as follows:

```
$ svnserve –d –r=/home/mtj
```

In this case, you specify to run as a daemon (in the background) with the `–d` option and specify the root level for your repository as a home directory. You'll probably want a more suitable place for your repository.

As with CVS, you export an environment variable to represent the location of the repository.

```
$ export SVNROOT=/home/mtj/svn_repo
$ svnadmin create ${SVNROOT}
```

The `svnadmin` utility is used to create your new repository with the `create` command. This utility is special in that it can be used to create repositories, verify them, build hot copies of the repository, recover from internal issues, dump the contents of the repository, etc. You can learn about all of the capabilities of the `svnadmin` utility by using the `help` command:

```
$ svnadmin help
general usage: svnadmin SUBCOMMAND REPOS_PATH  [ARGS & OPTIONS ...]
Type 'svnadmin help <subcommand>' for help on a specific subcommand.

Available subcommands:
   crashtest
   create
   deltify
   dump
   help (?, h)
   hotcopy
   list-dblogs
   list-unused-dblogs
   load
   lslocks
   lstxns
   recover
   rmlocks
   rmtxns
   setlog
   verify

$
```

When this command is complete, a new subdirectory is created that contains a variety of other control directories (`./svn_repo/*`). This contains information about the repository as well as working data (users, locks, etc.).

Adding to the Repository

As with CVS, you can use the `import` command to `svn` to import existing source into the newly created repository. But here the simple approach is used where you just add the file to the repository using the `add` command to `svn`.

You begin by grabbing the current repository by checking it out. It is empty at first, but this allows you to add to the repository without using `import`.

```
$ svn checkout file:///home/mtj/svn_repo svnwork
Checked out revision 0.
$
```

In the sandbox (working directory, `/home/mtj/svnwork`), you add the file `test.c`, which is shown in Listing 9.1. After it's created, we add to the repository and then commit this change from the sandbox to the central repository:

```
$ svn add test.c
A         test.c
$ svn commit
Adding         test.c
Transmitting file data .
Committed revision 1.
$
```

Manipulating Files in the Repository

At this point, you have a repository created. Now you can correct the mistake in the source file (the missing `f` in `printf`). First, you need to check out the repository into a new directory and update the source file. After it's updated, you check in your files with a new `commit`.

```
$ svn checkout file:///home/mtj/svn_repo svntest
A   svntest/test.c
$ cd svntest
$ vi test.c
...
$ svn commit
Sending        test.c
Transmitting file data .
Committed revision 2.
$
```

Similar to CVS, you can use `svn diff test.c` to see the changes to the file before you check in. You can also use `diff` with the versions to see our changes. SVN allows a range of revisions as `rev_start:rev_end` as demonstrated here:

```
$ svn diff -r 1:2 test.c
Index: test.c
===================================================================
—- test.c      (revision 1)
+++ test.c      (revision 2)
@@ -2,7 +2,7 @@

 int main()
 {
-  print("test string.\n");
+  printf("test string.\n");

   return 0;
 }
$
```

In addition to specifying revision numbers, you can also provide symbolic arguments to specify other revisions. For example, rather than a version number you can provide a date, `HEAD` (which is the tip of the repository), `COMMITTED` (which is the last commit), and `PREV` (which is the version prior to the last committed). For example, to perform the same action as the last example, you can use `COMMITTED` and `PREV` as follows:

```
$ svn diff —r COMMITTED:PREV test.c
```

Merging Changes from the Repository

Merging changes from the repository into your local sandbox is performed with the `update` option to the `svn` command. Any changes that have been committed to the central repository are merged into your local sandbox. Each file is shown with its current revision. As shown here, no updates are found, and the file is indicated as revision 2.

```
$ svn update
At revision 2.
$
```

Other Useful Commands

One interesting command in SVN is `revert`, which allows you to remove your local copy and get the last version from the repository. This command is performed simply as:

```
$ svn revert test.c
```

In SVN, you can lock one or more files to guarantee exclusive access. While a file is locked, no other user can commit (check in) changes. This can be useful in situations where numerous changes are planned and you want to avoid manual merging. Locking is performed with the `lock` option and unlocking with the `unlock` option to the `svn` command.

```
$ svn lock test.c
'test.c' locked by user 'mtj'.
$ svn unlock test.c
'test.c' unlocked.
$
```

In some cases, developers can tag a repository (discussed shortly) to create a branch for development. This allows development to occur on a different set of files (for experimental code or code changes for a specific customer). While on a branch, the files are tagged with a separate revision number so that they can be uniquely identified. Branches are very useful for independent development, where a branch is used to contain experimental or working code. When the branched code is deemed suitable, the branch can be joined to the tip (or head of the repository). This is where the `merge` option comes to play. The `merge` option merges changes from two different revisions of source files (similar to the `join` operation in CVS). For example:

```
$ svn merge  -r 199:HEAD file:///home/mtj/svn_repo
```

merges changes to all files tagged with revision `199` with the tip. Note that this operation affects the repository (merges take place in the central repository).

One other difference between CVS and Subversion is that in CVS, revision numbers are specific to files, where in Subversion, revision numbers are applied to a complete change-set.

Another useful command you can use to review which user is responsible for which set of changes is the `blame` option. This oddly named option to SVN shows

which users last touched each line of a file. The option also shows what version number is associated with the file (for example, which changes were introduced by the last revision). Given the changes to `test.c` so far, you can view the file as:

```
$ svn blame test.c
     1        mtj #include <stdio.h>
     1        mtj
     1        mtj int main()
     1        mtj {
     2        mtj   printf("test string.\n");
     1        mtj
     1        mtj   return 0;
     1        mtj }
$
```

Note here the line updated is shown as revision 2 while the remainder of the file remains with revision 1.

Repository Tagging

One of the key differences between CVS and Subversion is the way that branches and tags are handled. Subversion has no `tag` command, but instead the `copy` command (which correlates all files together)

Subversion permits a variety of ways to create branches, but the simplest is using the `copy` command on the repository as a URL. This is demonstrated as follows:

```
$ svn copy –r HEAD file://localhost/home/mtj/svn_repo \
                        file://localhost/home/mtj/svn_repo/br_1
$
```

With the branch created, you can check it out by specifying the branch as follows:

```
$ svn checkout file://localhost/home/mtj/svn_repo/br_1 svnwork
A    svnwork/test.c
Checked out revision 3.
$
```

You can now check the status of the branch with the `log` option. With this option you can see the history of the file, showing (from most recent to least recent) your branch (`A`), edit (`M`), and initial introduction (`A`) into the repository.

```
$ svn log -v svnwork
------------------------------------------------------------------------
r3 | mtj | 2008-01-06 00:18:24 -0700 (Sun, 06 Jan 2008) | 1 line
Changed paths:
   A /br_1 (from /:2)

Branched.

------------------------------------------------------------------------
r2 | mtj | 2008-01-05 16:49:52 -0700 (Sat, 05 Jan 2008) | 2 lines
Changed paths:
   M /test.c

Updated.

------------------------------------------------------------------------
r1 | mtj | 2008-01-05 16:45:57 -0700 (Sat, 05 Jan 2008) | 2 lines
Changed paths:
   A /test.c

Added.

------------------------------------------------------------------------
$
```

Note that any changes made to this sandbox are committed to the branch and not to the tip (`HEAD`). This is important because the branch acts as an independent sandbox allowing you to make your changes without affecting the tip (which should be stable).

Removing Files from the Repository

Files or directories (including tags and branches) can be removed from the repository with the `delete` option. This removes assets, but maintains logs to track the changes. For example, say you want to delete your newly created branch file.

```
$ svn delete test.c
$ svn commit
```

If you now remove the `svnwork` directory and check it out anew, you can see from the log that the branch version was removed. You can also remove the branch itself like so:

```
$ svn checkout file://localhost/home/mtj/svn_repo svnwork
$ svn delete svnwork/br_1
$ svn commit
```

Now if you use the `svn log` command on the `svnwork/test.c`, you find that only two revisions are known (because the tip and branch are unrelated):

```
$ svn log -v test.c
------------------------------------------------------------------------
r2 | mtj | 2008-01-05 16:49:52 -0700 (Sat, 05 Jan 2008) | 2 lines
Changed paths:
   M /test.c

Updated.

------------------------------------------------------------------------
r1 | mtj | 2008-01-05 16:45:57 -0700 (Sat, 05 Jan 2008) | 2 lines
Changed paths:
   A /test.c

Added.

------------------------------------------------------------------------
$
```

SVN Summary

If you're starting a new project, SVN should be on the top of your list for source management systems. It improves upon CVS while retaining the look and feel of CVS commands and behaviors.

Git

The Git source control system is one of the newest available, coming right out of Linux kernel development. Its origin is somewhat controversial, but it was created to replace the Bitkeeper source control system that was traditionally used to maintain Linux kernel source.

Git is interesting because it uses a different architecture than CVS and Subversion. Git uses a decentralized model permitting multiple distributed repositories (not a single centralized repository with external sandboxes). Git is also change-set based and uses a file-group model instead of simply tracking individual files.

This section explores Git for simple project management and gives you the basics for operating your own Git repository.

Installing Git

Because Git is so new, it's unlikely that you have the binaries on your system. Here's how you can get the latest. If you have APT, then that's your best bet. The package name for APT is cogito. If you don't have APT, then follow the instructions here to get the Git SCM source package, build, and install.

```
$ wget http://www.kernel.org/pub/software/scm/git/git-1.5.3.7.tar.gz
$ tar xvfz git-1.5.3.7.tar.gz
$ cd git-1.5.3.7
$ ./configure
$ make
$ make install
```

When these steps are successfully completed, the Git tools are installed. Most Git commands are available through the utility `git`.

Setting Up a New Git Repository

The `init` option to the `git` command is used to create a new repository. The first step is to define where the Git repository should be stored. The default is the current working directory in a new directory named `.git`. Instead, this example defines the Git repository as `git_repo`. Git expects that the root directory is defined in the `GIT_DIR` environment variable.

```
$ export GIT_DIR=/home/mtj/git_repo
$ git init
Initialized empty Git repository in /home/mtj/git_repo/
$
```

When this command is complete, the new subdirectory is created (`git_repo`) that contains a variety of other directories. This contains information about the repository as well as working data (tags, etc.).

Adding to the Repository

Now you can add to the repository by creating a work tree. The work tree is created with the `clone` option to the `git` command.

```
$ git clone -l -s git_repo git_work
Initialized empty Git repository in /home/mtj/git_work/.git/
$
```

Next, you define another environment variable to let Git know where the work tree resides.

```
$ export GIT_WORK_TREE=/home/mtj/git_work
```

To add to the tree, you descend into the work tree subdirectory and add the file (again, from Listing 9.1).

```
$ cd git_work
```

With the new file in the work tree, you add and commit the file to the upstream repository.

```
$ git add .
$ git commit
Created initial commit f89844f: Added.
 1 files changed, 8 insertions(+), 0 deletions(-)
 create mode 100644 test.c
$
```

Manipulating Files in the Repository

Now you can correct the source file and check it back into the repository. With the change made, you can check to see how it differs from the tip (`HEAD`) using the `diff` option to the `git` command. This is issued from the work tree (`./git_work`). This lists the entire source file and shows which file represents the repository (`a`) and which represents the work tree (`b`) and then the typical `diff` format of changed lines.

```
$ git diff
diff –git a/test.c b/test.c
index 16e6e96..3c04533 100644
–- a/test.c
+++ b/test.c
@@ -2,7 +2,7 @@

 int main()
 {
-  print("test string.\n");
+  printf("test string.\n");

   return 0;
 }
$
```

To commit the changes, you use the `commit` option with –a. The –a argument specifies that all updates should be committed (not just new files, but also updated and removed files).

```
$ git commit –a
Created commit 0ca384b: Updated.
 1 files changed, 1 insertions(+), 1 deletions(-)
$
```

You can also review changes with the `show` command. This shows the files that were changed along with change information (commit SHA1 hash, author, log, changed lines, etc.).

```
$ git show
commit 0ca384bf99436306b43d71646b73e6f3f324f3a4
Author: M. Tim Jones <mtj@camus.tim-jones.localdomain>
Date:   Sun Jan 6 16:14:22 2008 -0700

    Updated.

diff –git a/test.c b/test.c
index 16e6e96..3c04533 100644
–- a/test.c
+++ b/test.c
@@ -2,7 +2,7 @@

 int main()
 {
-  print("test string.\n");
+  printf("test string.\n");

   return 0;
 }
$
```

Merging Changes from the Repository

To merge any changes from another branch, or the tip, you use the `merge` option of the `git` command. This takes changes from that branch and merges them into our local work tree. The command is issued with a branch name; in the following example you merge from the `HEAD` (tip), and no changes are found.

```
$ git merge HEAD
Already up-to-date.
$
```

Other Useful Commands

Git has so many other useful commands that it's difficult to do it justice here. But to give you a taste, here are a couple of commands that illustrate the strength of Git as a source management solution.

First, recall the `show` command, which shows a variety of information about a file or files in the repository. You can symbolically address other revisions in the repository with modifiers. For example, if you issued `git show HEAD`, you would see the current `HEAD` and changes made in the prior revision. You could also modify `HEAD` to show the previous set of changes (`HEAD`'s parent) as follows:

```
$ git show HEAD^
commit f89844f4238827a25468c5d6d5f2fd27a18d2fb5
Author: M. Tim Jones <mtj@camus.tim-jones.localdomain>
Date:   Sun Jan 6 15:46:34 2008 -0700

    Added.

diff –git a/test.c b/test.c
new file mode 100644
index 0000000..16e6e96
–- /dev/null
+++ b/test.c
@@ -0,0 +1,8 @@
+#include <stdio.h>
+
+int main()
+{
+  print("test string.\n");
+
+  return 0;
+}
$
```

This shows the change of the prior revision of `HEAD` (which happens to be the initial checkin). You could modify this for `HEAD^^`, which represents the parent of `HEAD^`, but in this case because no further lineage exists, the result would be unknown.

Git can quickly summarize the changes that exist in your work tree with the `status` option. This option tells you which files were added, removed, or changed as compared to the parent of your work tree. As shown in the following, no changes are outstanding.

```
$ git status
# on branch master
nothing to commit (working directory clean)
```

Repository Tagging

One of Git's strengths is its tagging capabilities. A tag is created very simply with the `tag` option of the `git` command. Here you tag with the name `release_1`.

```
$ git tag release_1
$
```

This isn't entirely encouraging, but if you issue the `tag` command again without any options, you can see the list of tags that currently exist.

```
$ git tag
release_1
$
```

The `show-branch` option can also be used to check the status of the branch. Note here that you specify the branch that you are interested in.

```
$ git show-branch release_1
[release_1] Updated.
$
```

Git also allows tags to be deleted, which is done with the `–d` option:

```
$ git tag –d release_1
Deleted tag 'release_1'
$
```

Branching is done similarly in Git with the `branch` option. The following commands create a branch and then show it. Note that the `*` indicates the current branch (`master`), which you change by checking out the branch.

```
$ git branch my_branch
$ git branch
* master
my_branch
$ git checkout my_branch
Switched to branch "my_branch"
$ git branch
  master
* my_branch
```

You can see from these simple examples that tagging and branching in Git is a simple operation. That's the tip of the iceberg; more information is available in the Git documentation and user's guide.

Removing Files from the Repository

Removing files or directories from a repository is done with the `rm` option. This is done simply as follows:

```
$ git rm test.c
rm 'test.c'
$ git commit
Created commit 351c4f6: Deleted.
 1 files changed, 0 insertions(+), 8 deletions(-)
delete node 10064 test.c
$
```

Note that deletions (like updates) do not affect the upstream repository without a commit.

Git Summary

Git is a great new distributed source control system that is seeing adoption outside of the Linux kernel (for which it was originally designed). The distributed repository model is ideal for situations where hierarchical project teams work toward a master repository. It also offers a great set of options to monitor checkins, which are hashed for security and traceability. Git is a great choice for new projects that require greater control and project visibility.

Summary

As with many other technologies, you find no one-size-fits-all in source and revision management. Centralized repositories (still the most popular) are simple and efficient to manage, but distributed repositories provide the means to distribute development. It's no surprise that distributed models of source management arise now because the evolution of open source development has guided them. When you consider projects such as the Linux kernel, you find the distributed model works very well. Regardless of the model that you choose, you can no doubt find a source management system that fits your needs. This chapter presented CVS, Subversion, and Git, but many others exist (such as Arch) and are worth your time to explore.

References

[Wikipedia:SVN] "Subversion (software)," last retrieved on 12/30/2007 at http://en.wikipedia.org/wiki/Subversion_(software).

Resources

CVS—Concurrent Versions System, last retrieved on 01/06/2008 at http://www.nongnu.org/cvs/.

Collabnet Subversion Home Site, last retrieved on 01/06/2008 at http://subversion.tigris.org/project_packages.html.

"Git—Fast Version Control System," last retrieved on 01/06/2008 at http://git.or.cz.

10 Data Visualization with Gnuplot

In This Chapter

- Gnuplot Installation
- An Introduction to the Gnuplot User Interface
- Plotting Functions and Data from Files
- 2D and 3D Plotting
- Building Multiplot Graphs

Introduction

GNU/Linux offers a multitude of open-source solutions to visualize data. These not only transform your data into graphs, plots, or specialized images, but also include the capabilities to filter and reduce your data to make it more useful. This chapter explores one of the most popular solutions, Gnuplot.

Gnuplot

Gnuplot is one of the older visualization programs, but it remains one of the best and most popular. It uses a shell interface and runs on practically any system you can think of (such as Linux, VMS, and even the Atari). It's interactive through a command-line shell, so you can plot functions or your data in 2D or 3D with a plethora of different options. Finally, you can find language bindings for a number of popular languages such as Ruby, Python, and Smalltalk. These bindings allow you to generate plots from within these languages.

Gnuplot was first released in 1986, and while the name implies that it's part of the GNU project, it's actually released under its own license.

In this chapter, we'll review the Gnuplot solution and explore some of its capabilities.

Installing Gnuplot

Gnuplot can be packaged as a standard utility, but if your distribution doesn't have Gnuplot (as indicated by no response to the command `which gnuplot`), then you can install it easily. If your distribution supports APT, then Gnuplot can be installed with the following:

```
$ sudo apt-get install gnuplot
```

Otherwise, you can install it from source as shown in Listing 10.1.

LISTING 10.1 Installing the Gnuplot Tool from Source

```
$ sudo su –
$ cd /usr/local/src
$ wget http://internap.dl.sourceforge.net/sourceforge/gnuplot/
gnuplot-4.2.2.tar.gz
$ tar xvfz gnuplot-4.2.2.tar.gz
$ cd gnuplot-4.2.2
$ ./configure
$ make
$ make install
$ which gnuplot
/usr/local/bin/gnuplot
$
```

User Interface

Now it's time to get started working with Gnuplot. First, you need to get acquainted with the Gnuplot user interface shell. When you start Gnuplot, you see the code shown in Listing 10.2. The prompt `gnuplot>` indicates that Gnuplot is ready for user input. You should also note that the terminal type is set to `x11`, which means that any plots that you generate are emitted graphically to the screen. You can also redirect plots to files, which is explored later in the chapter.

LISTING 10.2 The Startup Screen for Gnuplot

```
$ gnuplot

        G N U P L O T
        Version 4.2 patchlevel 0
        last modified March 2007

        ...

Terminal type set to `x11`
gnuplot>
```

In addition to the plethora of commands available in Gnuplot, you also have a large number of variables that you can use to tailor Gnuplot's operation and specify plot options. Variables are displayed with the `show` command and set with the `set` command. In the following example, you use the `show` command with the `terminal` variable to find that the terminal is currently set to `x11` (the default). You then set the `terminal` variable to `png` (output format) using the `set` command (see Listing 10.3).

LISTING 10.3 Setting and Viewing Variables

```
gnuplot> show terminal

   terminal type is x11 0

gnuplot> set terminal png
Terminal type set to 'png'
Options are 'nocrop medium'
gnuplot> set terminal x11
Terminal type set to 'x11'
Options are '0'
gnuplot>
```

Gnuplot supports a very large number of variables, some of which are explored in this chapter. You can see all of the available variables using the command `show all`. You can remove a variable using the `unset` command.

From the Gnuplot shell you can get help on any command or option by typing `help`. *If you know the command that you're interested in, you can more quickly find the information you need by typing* `help <cmd>`.

Simple Plots

Now it's time to take a look at how to generate some simple plots with Gnuplot, and explore some of the plotting options along the way. You can create a simple plot by plotting a function within Gnuplot.

```
gnuplot> plot sin(x)
```

This plot is shown in Figure 10.1. Note here that the scale was automatically defined (the X range was defaulted to -10.0 to 10.0, whereas the Y range was autoscaled to the range of the data, -1.0 to 1.0).

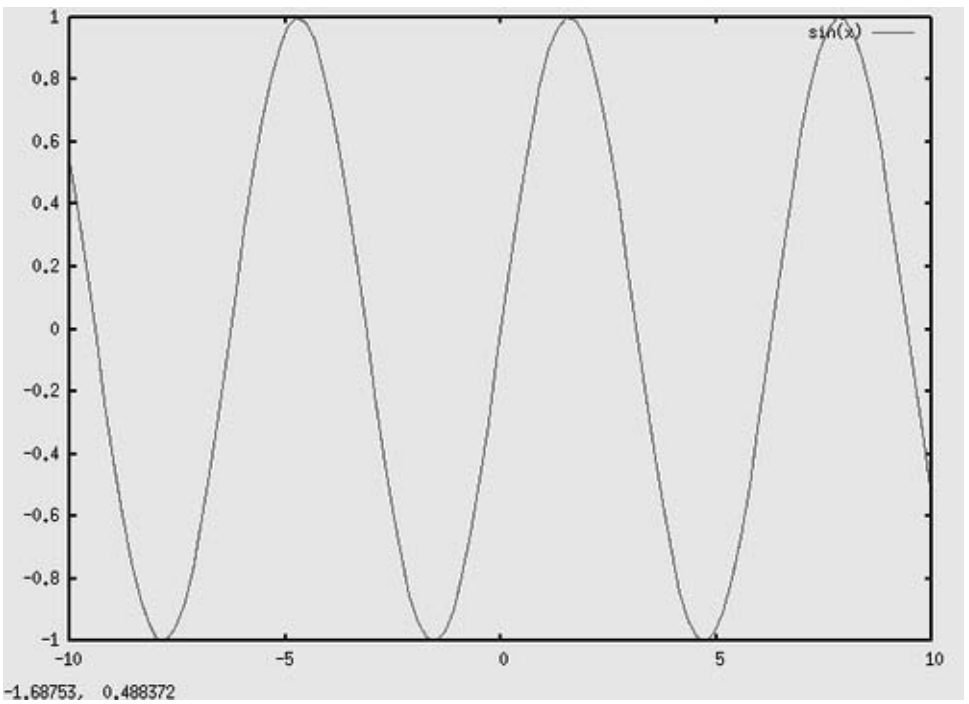

FIGURE 10.1 A simple function plot in 2D.

The plot lacks some of the basic elements that make it useful, such as a title, labels on the axis, etc. You can very easily add these to the plot by setting the variables that represent those plot elements. Listing 10.4 illustrates some of the variables that modify the plot for the title, ranges, axis labels, and point labels within the plot (see Listing 10.4).

LISTING 10.4 Increasing the Readability of the Plot

```
gnuplot> set title "Simple Function Plot"
gnuplot> set xrange [-3.14159:3.14159]
gnuplot> set yrange [-1.1:1.1]
gnuplot> set xlabel "theta"
gnuplot> set ylabel "sin(theta)"
gnuplot> set label "sin(0.0)" at 0.0, 0.0
gunplot> plot sin(x)
```

The result of Listing 10.4 is the plot shown in Figure 10.2.

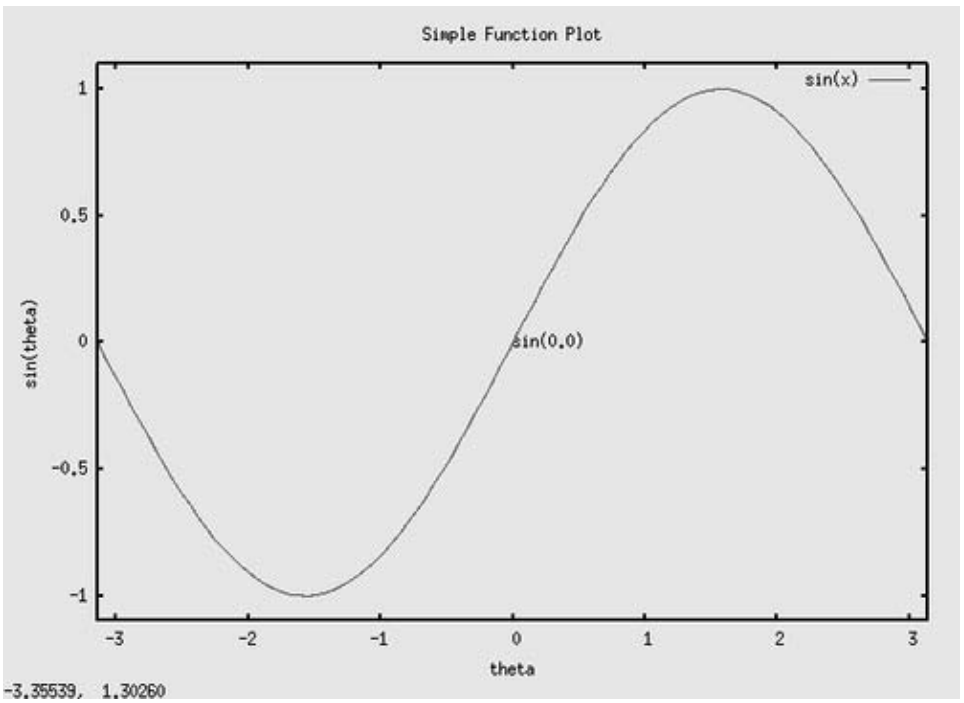

FIGURE 10.2 Updated 2D plot from Listing 10.4.

Retyping this can be tiresome, so you can script Gnuplot by including this in a text file and then loading it into Gnuplot. Gnuplot scripts are usually suffixed with `.p`, so invoking our script from the file `script.p` is done as follows:

```
gnuplot> load 'script.p'
```

This causes the plot shown in Figure 10.2 to be rendered.

Plotting Data from a File

In most cases, you want to plot your own data from outside of Gnuplot. This is simple as well; instead of plotting a function, you plot a filename. For example, you can start with data collection.

Listing 10.5 provides a simple script that collects data from the `/proc` filesystem. This is real-time data that represents kernel activity. In this case, you capture interrupt counts from the `/proc/stat` file.

LISTING 10.5 Simple Script to Capture Interrupt Counts

```
#!/bin/bash
rm –f procstat.txt
for ((i=0;i<30;i+=1)); do
cat /proc/stat | grep intr >> procstat.txt
sleep 1
done
exit
```

The `stat` file in `/proc` provides a number of lines of data, so you filter out everything but the line that contains `intr`. The result is 30 lines of interrupt counts. The data in `procstat.txt` appears as follows:

```
intr 190574 122 6680 0 1 1 0 5 0 3 0 0 0 5221 0 0 128153 ...
intr 190575 122 6681 0 1 1 0 5 0 3 0 0 0 5221 0 0 128153 ...
intr 190590 122 6681 0 1 1 0 5 0 3 0 0 0 5221 0 0 128168 ...
...
```

The interrupt data contains interrupt counts for each of the interrupts in the system. The first count (after the `intr` header) is the total number of interrupts that have occurred. Each line thereafter is an individual interrupt. To plot the relationship of one interrupt to the total, you can use the `plot` command that follows. Note here that instead of a function, you specify the filename of your data in single quotes. You then specify the data points of interest (second and eighteenth). You then customize the plot by specifying that it should be linespoints (lines with points at the data elements) and a smoothed curve. Note also the definition of the plot element title with the title option.

```
gnuplot> plot 'procstat.txt' using 2:18 with linespoints \
        smooth Bezier title "Interrupt Relationship"
```

The result is the plot in Figure 10.3.

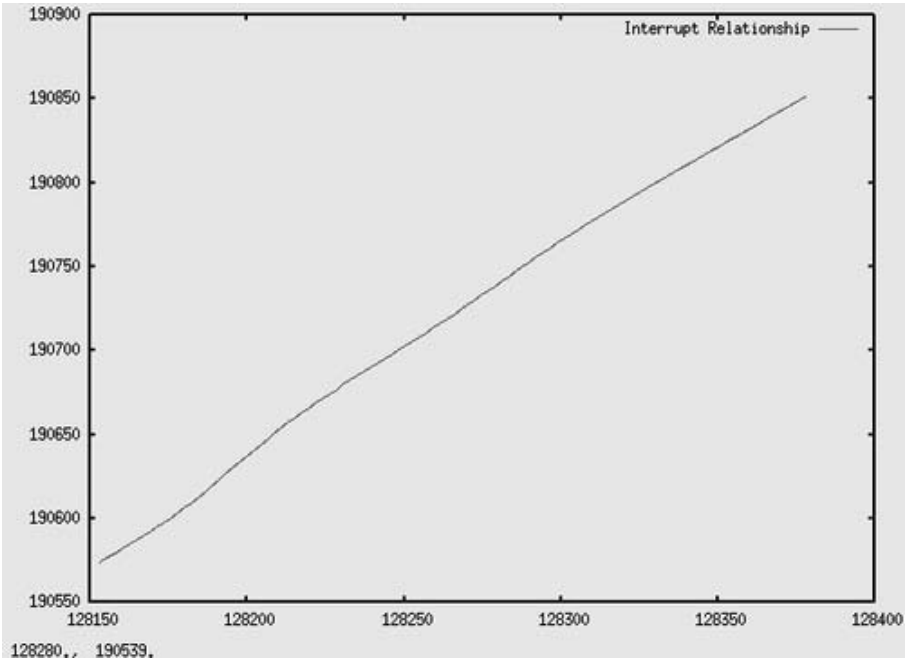

FIGURE 10.3 Sample plot of interrupt data from a file.

In addition to using the `Bezier` *option with* `smooth`*, you can also provide* `csplines`*,* `acsplines`*, and* `sbezier`*. These options, in addition to* `Bezier`*, create a continuous curve between the provided data points.*

Plotting Functions in 3D

The `splot` command is used in Gnuplot to generate three-dimensional plots. Because a 3D plot is made up of three dimensions (a 2D projection), you provide three values representing the X, Y, and Z coordinates. Gnuplot supports a variety of formats of 3D plot data (such as the matrix format), but this section explores the standard case first.

To plot a function with `splot`, simply provide the X and Y ranges and then the Z component. As shown in the following, you constrain the range of the X and Y components to the range -2 to 2. Finally, the Z component is provided as the function shown.

```
gnuplot> splot [x=-2:2] [y=-2:2] sin(x) * cos(y)
```

Gnuplot then does the rest, resulting in the function plot shown in Figure 10.4

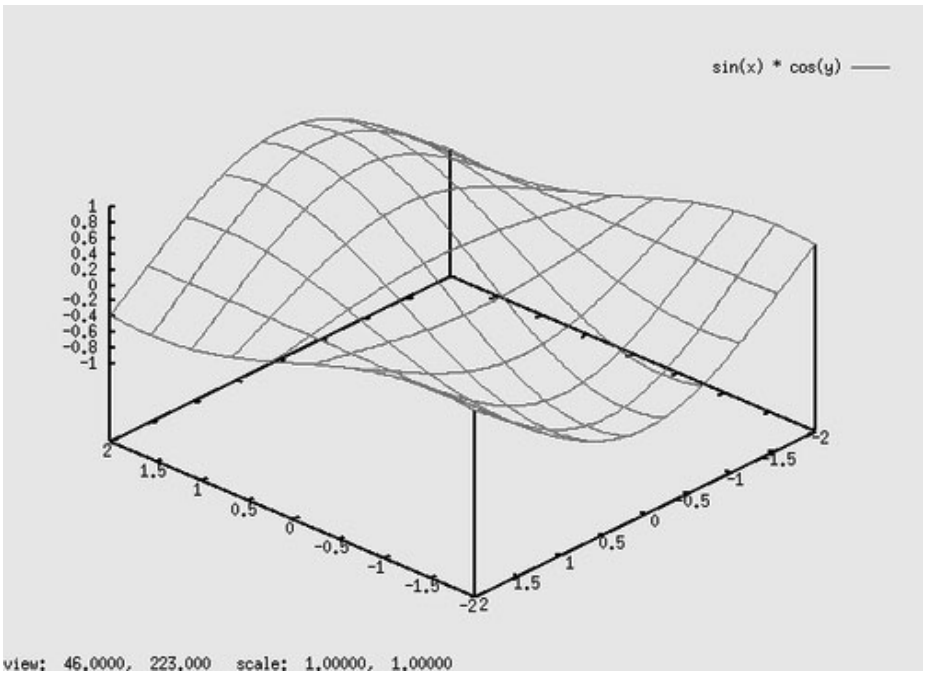

FIGURE 10.4 A simple 3D function plot.

Creating 3D plots from a file is also simple once you know the format of the data. In the simplest form, you can specify data in terms of the X, Y, and Z components, as shown in Listing 10.6. Note also here that some lines contain a comment, or blank space. Gnuplot automatically ignores these to produce a correct plot (but it makes the format more readable).

LISTING 10.6 Gnuplot Three-Dimensional Data Format

```
#  X  Y   Z
   0  0   1
   0  1   2
   0  2   4
   0  3   8

   1  0   1
   1  1   3
   1  2   9
   1  3  27
```

```
2  0   1
2  1   4
2  2  16
2  3  64
```

To plot the data in Listing 10.6 (shown in Figure 10.5), you can use the `splot` command shown as follows. Note that you tell Gnuplot about the format of the data with the `using` option (which establishes the fields of the data to be used for the plot).

```
gnuplot> splot 'data.txt' using 1:2:3 with linespoints title "3D plot
data"
```

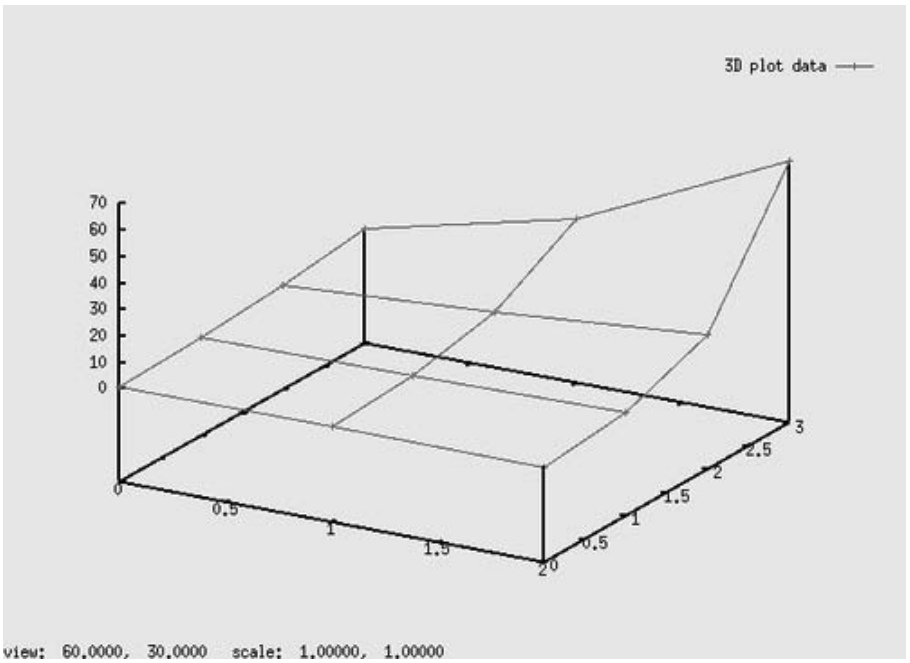

FIGURE 10.5 A simple 3D plot from data (from Listing 10.6).

Note that the data in Listing 10.6 does not need to be in any particular order. It can be scattered in an arbitrary order.

Finally, a simplified format for the data in Listing 10.6 is shown in Listing 10.7. This is the matrix format.

LISTING 10.7 The Matrix Format for 3D Plots (`data.txt`)

```
1  2  4  8
1  3  9 27
1  4 16 64
```

To plot the data in this format, you simply need to tell Gnuplot that the data is in this special format. This is done as follows for the plot shown in Figure 10.6.

```
gnuplot> splot 'data.txt' matrix with linespoints
```

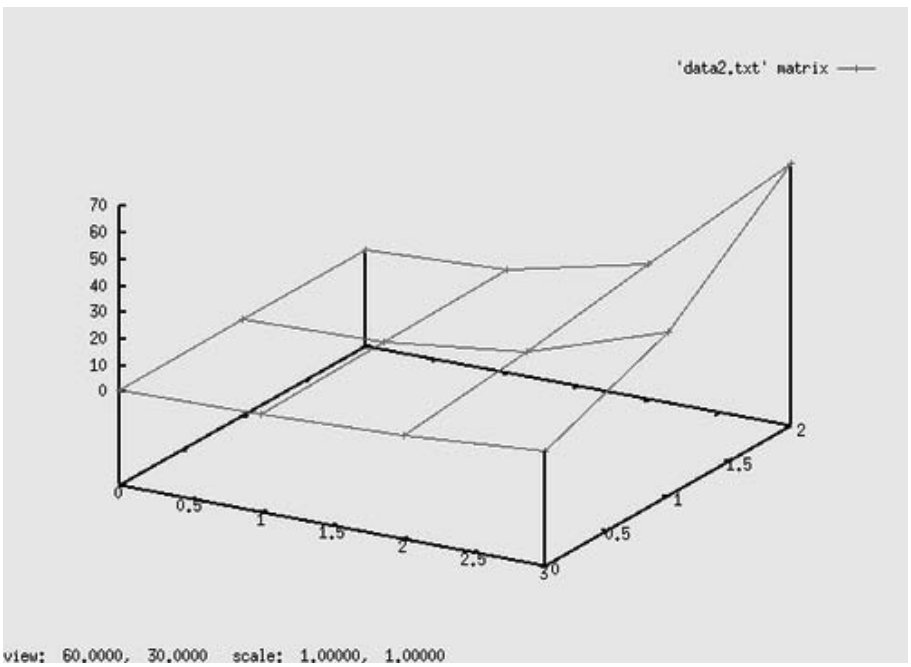

FIGURE 10.6 A simple matrix plot.

3D Plots with Contours

In addition to 3D plots, you can also apply contours to the plots. A contour plot is a two-dimensional representation of the Z component of a 3D plot onto a 2D surface. To enable a contour plot you simply enable it with the command `set contour`. You also customize the plot to increase the number of contour lines and demonstrate the `isosamples` variable. `isosamples` allows you to control the density of the grid lines on the 3D plot.

```
gnuplot> set contour
gnuplot> set isosamples 40
gnuplot> set cntrparam levels 10
gnuplot> splot [x=-4:4] [y=-4:4] sin(x)*cos(y)
```

The result is shown in Figure 10.7. This can be helpful to see the contour of the plot, especially if some of the data is hidden or obfuscated.

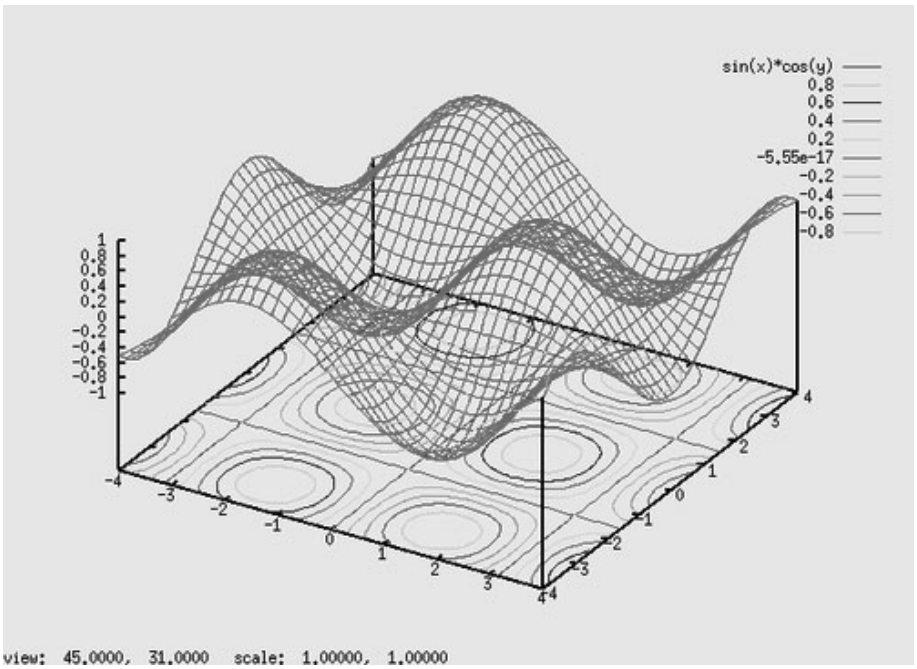

FIGURE 10.7 Incorporating a contour plot onto a 3D plot.

Hidden Line Removal

Hidden line removal can also be applied to plots to make them appear more realistic. Removing hidden lines simply means that lines that are covered by other surfaces are not displayed. Enabling hidden line removal is done with the following command:

```
gnuplot> set hidden
gnuplot> splot [x=-4:4] [y=-4:4] sin(x)*cos(y)
```

When this is executed, and the previous plot shown in Figure 10.7 is re-plotted, Figure 10.8 results.

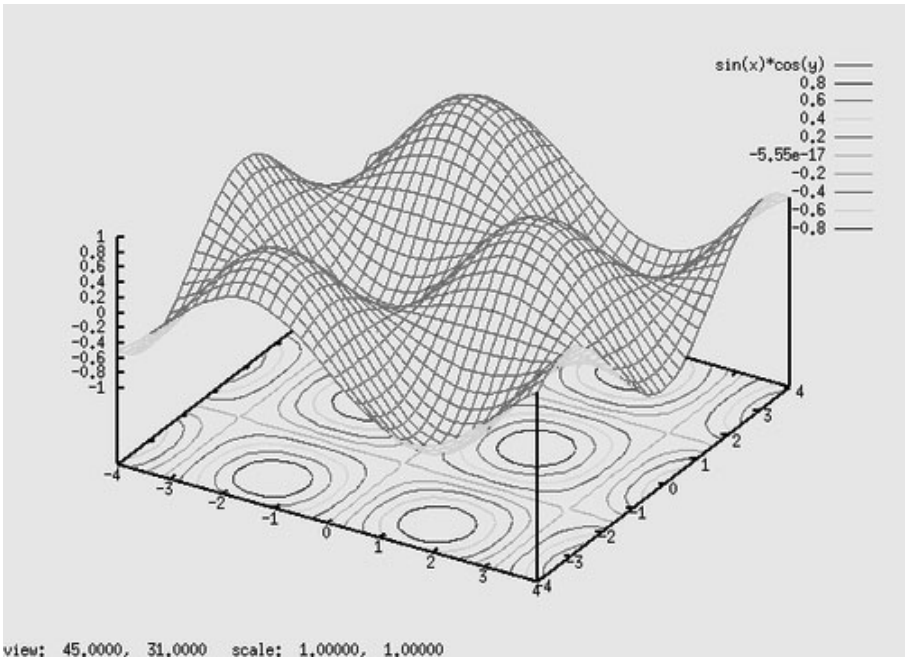

FIGURE 10.8 Hidden line removal applied to Figure 10.7.

Storing Plots to a File

Because you typically generate plots to communicate information with others, the ability to generate plots into files is necessary. To store a plot to a file you need to specify the format of the output file and the name of the output file. The output format is defined using the `terminal` variable and the output file defined with the `output` variable. The example shown in the following code lines emits a graphics file named `plot.png` containing the plot in Portable Network Graphics (PNG) format.

```
gnuplot> set terminal png
Terminal type set to 'png'
Options are 'nocrop medium'
gnuplot> set output "plot.png"
gnuplot> splot [x=-4:4] [y=-4:4] sin(x)*cos(y)
```

To reset the terminal back to the previous settings, you can issue the command `pop`.

```
gnuplot> set terminal pop
```

Gnuplot supports a large number of terminal options, including `gif`, `corel`, `jpeg`, `postscript`, `svg`, and many others.

MULTIPLOTS

Gnuplot also supports the feature of multiplots, which is the capability of incorporating numerous plots onto a single plot image. You have a number of ways to create multiplots, but the easiest is using the `layout` directive. This enables you to specify the number of plots on the page (rows and columns) and how the plots should automatically be placed (rows first or columns first). A title can also be applied for the entire plot here (using the `title` option).

After the layout is defined, each of the plots can be specified as shown in Listing 10.8. After the `unset multiplot` command is encountered, the plot is displayed. The resulting plot is shown in Figure 10.9.

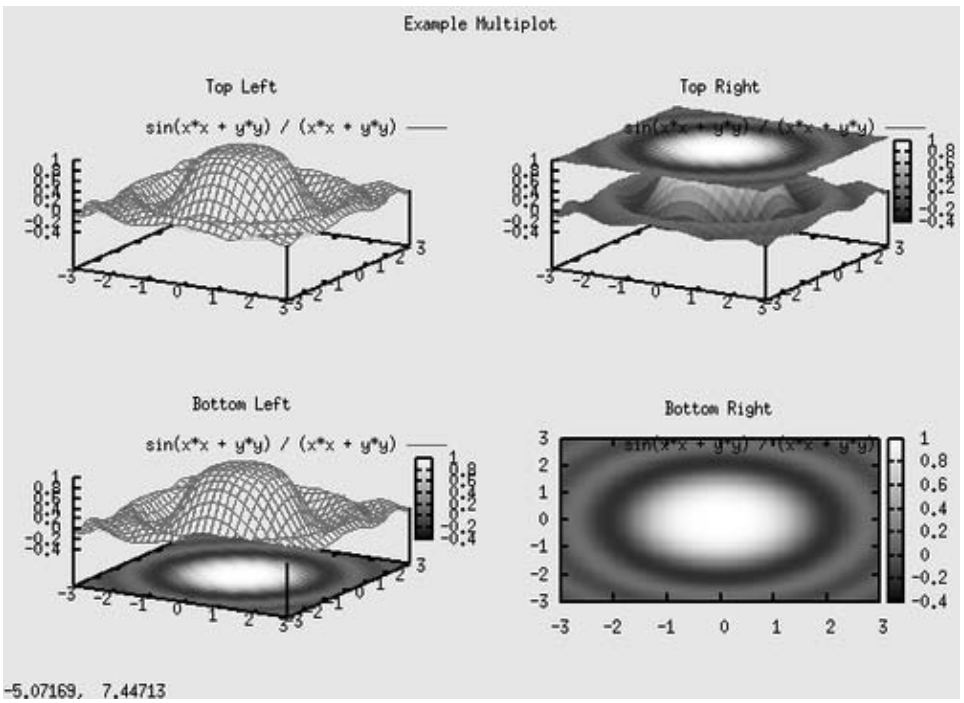

FIGURE 10.9 Sample four quadrant multiplot (using Listing 10.8).

LISTING 10.8 Generating a Simple Four Quadrant Multiplot (`mplot.p`)

```
set multiplot layout 2,2 rowsfirst title "Example Multiplot"

set title "Top Left"
set hidden
set isosamples 30
set xrange [-3:3]
set yrange [-3:3]
splot sin(x*x + y*y) / (x*x + y*y)


set title "Top Right"
set hidden
set isosamples 30
set xrange [-3:3]
set yrange [-3:3]
set pm3d at st
splot sin(x*x + y*y) / (x*x + y*y)


set title "Bottom Left"
set hidden
set isosamples 30
set xrange [-3:3]
set yrange [-3:3]
set pm3d at b
splot sin(x*x + y*y) / (x*x + y*y)


set title "Bottom Right"
set samples 100
set isosamples 100
set xrange [-3:3]
set yrange [-3:3]
set pm3d map
set palette gray
splot sin(x*x + y*y) / (x*x + y*y)

unset multiplot
```

Tools That Use Gnuplot

Gnuplot is so popular that you can find it used as the means for graph visualization in a number of applications. For example, you can find Gnuplot in the GNU Octave package (a Matlab-like program for scientific computing) and also in Maxima (a computer algebra system written in Common Lisp). It's also used in a number of other applications because of its flexibility.

Summary

Gnuplot is one of the most popular open-source data visualization applications and is also one of the oldest. It supports interactive scripting to plot functions and data in a number of ways as well as a scripted mode that allows you to write scripts and generate plots either online or into files of various formats. Gnuplot is actively developed and a major new revision appeared in 2007.

Resources

Gnuplot Demo Plots (useful site demonstrating how to achieve various plotting results with Gnuplot) at http://gnuplot.sourceforge.net/demo_4.2.

The Gnuplot Home Page at http://www.gnuplot.info.

This page intentionally left blank

Part III Application Development Topics

This part of the book reviews a number of topics important to application development. This includes using the most important elements of GNU/Linux including various IPC mechanisms, Sockets, and multiprocess and multithreaded programming.

Chapter 11: File Handling in GNU/Linux

The file handling APIs are important in GNU/Linux because they contain patterns for many other types of I/O, such as sockets and pipes. This chapter demonstrates the proper use of the file handling APIs using binary, character, and string interfaces. Numerous examples illustrate the APIs in their different modes.

Chapter 12: Programming with Pipes

The pipe model of communication is an older aspect of UNIX, but it is still an important one considering its wide use in shell programming. The pipe model is first reviewed, with discussion of anonymous and named pipes. The API to create pipes is discussed along with examples of using pipes for multiprocess communication. Shell-level creation and use of pipes completes this chapter.

Chapter 13: Introduction to Sockets Programming

Network programming using the standard Sockets API is the focus of this chapter. Each of the API functions is detailed illustrating their use in both client and server systems. After a discussion of the Sockets programming paradigm and each of the API functions, other elements of Sockets programming are discussed including multilanguage aspects.

Chapter 14: GNU/Linux Process Model

The GNU/Linux process model refers to the standard multiprocessing environment. This chapter discusses the `fork` function (to create child processes) and the other process-related API functions (such as `wait`). The topic of signals is also discussed including the range of signals and their uses. Finally, the GNU/Linux process commands (such as `ps`) are detailed.

Chapter 15: POSIX Threads (Pthreads) Programming

Programming with threads using the pthreads library is the topic of this chapter. The functions in the pthreads library are discussed including thread creation and destruction, synchronization (with mutexes and condition variables), communication, and other thread-related topics. Problems in multithreaded applications are also discussed, such as re-entrancy.

Chapter 16: IPC with Message Queues

Message queues are a very important paradigm for communication in multiprocess applications. The model permits one-to-many and many-to-one communication and a very simple and intuitive API. This chapter details the message queue APIs for creating, configuring, and then sending and receiving messages. Advanced topics such as conditional message receipt are also discussed along with user-layer utilities for message queue inspection.

Chapter 17: Synchronization with Semaphores

Semaphores in GNU/Linux and the ability to create critical sections are the topics of this chapter. After a discussion of the problems that semaphores solve, the API for semaphores is detailed including creation, acquisition, and release and removal. The advanced features provided by GNU/Linux such as semaphore arrays are discussed including user-level commands to inspect and remove semaphores.

Chapter 18: Shared Memory Programming

One of the most important process communication mechanisms available in GNU/Linux is shared memory. The shared memory APIs allow segments of memory to be created and then shared between two or more processes. This chapter details the shared memory APIs for creating, attaching, detaching, and locking and unlocking shared memory segments.

Chapter 19: Advanced File Handling

In Chapter 11, "File Handling," the basics of file handling in GNU/Linux were covered. In this chapter, the more advanced topics are explored, including file typing, traversing directories and filesystems (using a variety of mechanisms), and also filesystem event notification. For event notification, the inotify approach is explored, showing how to monitor many types of filesystem events from a user-space application.

Chapter 20: Other Application Development Topics

This final chapter of Part III explores some of the important application development topics that were not covered in the preceding chapters. The topics explored here include command-line parsing with the `getopt` and `getopt_long` APIs, time conversion functions, `sysinfo`, memory mapping with `mmap`, and locking and unlocking memory pages for performance.

This page intentionally left blank

11 File Handling in GNU/Linux

In This Chapter

- Understand File Handling APIs in GNU/Linux
- Explore the Character Access Mechanisms
- Explore the String Access Mechanisms
- Investigate Both Sequential and Nonsequential (Random Access) Methods
- Review Alternate APIs and Methods for File Access

Introduction

This chapter looks at the file handling APIs of GNU/Linux and explores a number of applications to demonstrate the proper use of the file handling APIs. It looks at a number of different file handling functions, including character interfaces, string interfaces, and ASCII-mode and binary interfaces. The emphasis on this chapter is to discuss the APIs and then employ them in applications to illustrate their use.

File Handling with GNU/Linux

File handling within GNU/Linux is accomplished through the standard C library. You can create and manipulate ASCII text or binary files with the same API. You can append to files or seek within them.

This chapter takes a look at the `fopen` call (to open or create a file), the `fwrite` and `fread` functions (to write to or read from a file), `fseek` (to position yourself at a given position in an existing file), the `feof` call (to test whether you are at the end of a file while reading), and some other lower level calls (such as `open`, `write`, and `read`).

The file system in Linux is based on the original Unix File System, Version 7 Unix, which was released in 1979 from Bell Labs.

File Handling API Exploration

Now it's time to get your hands dirty by working through some examples of GNU/Linux stream file I/O programming.

Creating a File Handle

To write an application that performs file handling, you first need to make visible the file I/O APIs (function prototypes). This is done by simply including the `stdio.h` header file, as follows:

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

Not doing so results in compiler errors (undeclared symbols). The next step is to declare the handle to be used in file I/O operations. This is often called a *file pointer* and is a transparent structure that should not be accessed by the developer.

```
FILE *my_fp;
```

The next sections build on this to illustrate ASCII and binary applications.

Opening a File

Now it's time to open a file and illustrate the variety of modes that can be used. Recall that opening a file can also be the mechanism to create a file. This example investigates this first.

The `fopen` function is very simple and provides the following API:

```
FILE *fopen( const char *filename, const char *mode );
```

You specify the filename that you want to access (or create) through the first argument (`filename`) and then the mode you want to use (`mode`). The result of the `fopen` operation is a `FILE` pointer, which could be `NULL`, indicating that the operation failed.

The key to the `fopen` call is the mode that is provided. Table 11.1 provides an initial list of access modes.

TABLE 11.1 Simple File Access Modes

Mode	Description
`r`	Open an existing file for read
`w`	Open a file for write (create new if a file doesn't exist)
`a`	Open a file for append (create new if a file doesn't exist)
`rw`	Open a file for read and write (create new if a file doesn't exist)

The mode is simply a string that the `fopen` call uses to determine how to open (or create) the file. If you wanted to create a new file, you could simply use the `fopen` call as follows:

```
my_fp = fopen( "myfile.txt", "w" );
```

The result would be the creation of a new file (or the destruction of the existing file) in preparation for `write` operations. If instead you wanted to read from an existing file, you'd open it as follows:

```
my_fp = fopen( "myfile.txt", "r" );
```

Note that you are simply using a different mode here. The `read` mode assumes that the file exists, and if not, a `NULL` is returned.

In both cases, it is assumed that the file `myfile.txt` either exists or is created in the current working directory. The current directory is the directory from which you invoked your application.

It's very important that the results of all file I/O operations be checked for success. For the `fopen` call, you simply test the response for `NULL`. What happens upon error is ultimately application dependent (you decide). An example of one mechanism is provided in Listing 11.1.

LISTING 11.1 Catching an Error in an `fopen` Call (on the CD-ROM at `./source/ch11/test.c`)

```
1:        #include <stdio.h>
2:        #include <errno.h>
3:        #include <string.h>
4:
5:        #define MYFILE    "missing.txt"
6:
7:        main()
8:        {
9:
10:         FILE *fin;
11:
12:         /* Try to open the file for read */
13:         fin = fopen( MYFILE, "r" );
14:
15:         /* Check for failure to open */
16:         if (fin == (FILE *)NULL) {
17:
18:           /* Emit an error message and exit */
19:           printf("%s: %s\n", MYFILE, strerror( errno ) );
20:           exit(-1);
21:
22:         }
23:
24:         /* All was well, close the file */
25:         fclose( fin );
26:
27:       }
```

In Listing 11.1, you use a couple of new calls not yet discussed. After trying to open the file at line 13, you check to see if the new file handle is `NULL` (zero). If it is, then you know that either the file is not present or you are not able to access it (you don't have proper access to the file). In this case, you emit an error message that consists of the file that you attempted to open for `read` and then the error message that resulted. You capture the error number (integer) with the `errno` variable. This is a special variable that is set by system calls to indicate the last error that occurred. You pass this value to the `strerror` function, which turns the integer error number into a string suitable for printing to standard-out. Executing the sample application results in the following:

```
$ ./app
missing.txt: No such file or directory
$
```

Now it's time to move on to writing and then reading data from a file.

The `errno` *variable is set to* `ENOENT` *if the file does not exist or* `EACCES` *if access to the file was denied because of lack of permissions.*

Reading and Writing Data

A number of methods exist for both reading and writing data to a file. More options can be a blessing, but it's also important to know where to use which mechanism. For example, you can read or write on a character basis or on a string basis (for ASCII text only). You can also use a more general API that permits reading and writing records, which supports both ASCII and binary representations. This chapter looks at each here, but focuses primarily on the latter mechanism.

The standard I/O library presents a buffered interface. This has two very important properties. First, system reads and writes are in blocks (typically 8KB in size). Character I/O is simply written to the `FILE` buffer, where the buffer is written to the media automatically when it's full. Second, `fflush` is necessary, or non-buffered I/O must be set if the data is being sent to an interactive device such as the console terminal.

Character Interfaces

The character interfaces are demonstrated in Listings 11.2 and 11.3. Listing 11.2 illustrates character output using `fputc` and Listing 11.3 illustrates character input using `fgetc`. These functions have the following prototypes:

```
int fputc( int c, FILE *stream );
int fgetc( FILE *stream );
```

This example generates an output file using `fputc` and then uses this file as the input to `fgetc`. In Listing 11.2, you open the output file at line 11 and then work your way through your sample string. The simple loop walks through the entire string until a `NULL` is detected, at which point you exit and close the file (line 21). At line 16, you use `fputc` to emit the character (as an `int`, per the `fputc` prototype) as well as specify your output stream (`fout`).

LISTING 11.2 The `fputc` Character Interface Example (on the CD-ROM at `./source/ch11/charout.c`)

```
1:      #include <stdio.h>
2:
3:      int main()
4:      {
5:        int i;
6:        FILE *fout;
7:        const char string[]={"This\r\nis a test\r\nfile.\r\n\0"};
8:
9:        fout = fopen("inpfile.txt", "w");
10:
11:       if (fout == (FILE *)NULL) exit(-1);
12:
13:       i = 0;
14:       while (string[i] != NULL) {
15:
16:         fputc( (int)string[i], fout );
17:         i++;
18:
19:       }
20:
21:       fclose( fout );
22:
23:       return 0;
24:     }
```

The function to read this file using the character interface is shown in Listing 11.3. This function is very similar to the file creation example. You open the file for `read` at line 8 and follow with a test at line 10. You then enter a loop to get the characters from the file (lines 12–22). The loop simply reads characters from the file using `fgetc` and stops when the special `EOF` symbol is encountered. This is the indication that you've reached the end of the file. For all characters that are not `EOF` (line 16), you emit the character to standard-out using the `printf` function. Upon reaching the end of the file, you close it using `fclose` at line 24.

LISTING 11.3 The `fgetc` Character Interface Example (on the CD-ROM at `./source/ch11/charin.c`)

```
1:      #include <stdio.h>
2:
3:      int main()
```

```
4:        {
5:          int c;
6:          FILE *fin;
7:
8:          fin = fopen("inpfile.txt", "r");
9:
10:         if (fin == (FILE *)0) exit(-1);
11:
12:         do {
13:
14:           c = fgetc( fin );
15:
16:           if (c != EOF) {
17:
18:             printf("%c", (char)c);
19:
20:           }
21:
22:         } while (c != EOF);
23:
24:         fclose( fin );
25:
26:         return 0;
27:       }
```

Executing the applications is illustrated as follows:

```
$ ./charout
$ ./charin
This
is a test
file.
$
```

The character interfaces are obviously simple, but they are also inefficient and should be used only if a string-based method cannot be used. We'll look at this interface next.

String Interfaces

This section takes a look at four library functions that provide the means to read and write strings. The first two (`fputs` and `fgets`) are simple string interfaces, and the second two (`fprintf` and `fscanf`) are more complex and provide additional capabilities.

The `fputs` and `fgets` interfaces mirror the previously discussed `fputc` and `fgetc` functions. They provide the means to write and read variable-length strings to files in a very simple way. Prototypes for the `fputs` and `fgets` are defined as:

```
int fputs( int c, FILE *stream );
char *fgets( char *s, int size, FILE *stream );
```

First take a look at a sample application that accepts strings from the user (via standard-input) and then writes them to a file (see Listing 11.4). This sample then halts the input process after a blank line has been received.

LISTING 11.4 Writing Variable Length Strings to a File (on the CD-ROM at `./source/ch11/strout.c`)

```
1:      #include <stdio.h>
2:
3:      #define LEN    80
4:
5:      int main()
6:      {
7:        char line[LEN+1];
8:        FILE *fout, *fin;
9:
10:       fout = fopen( "testfile.txt", "w" );
11:       if ( fout == (FILE *)0 ) exit(-1);
12:
13:       fin = fdopen( 0, "r" );
14:
15:       while ( (fgets( line, LEN, fin )) != NULL ) {
16:
17:         fputs( line, fout );
18:
19:       }
20:
21:       fclose( fout );
22:       fclose( fin );
23:
24:       return 0;
25:     }
```

The application shown in Listing 11.4 gets a little trickier, so the next paragraphs walk through this one line by line to cover all of the points. You declare the line string (used to read user input) at line 7, called oddly enough, `line`. Next, you declare two `FILE` pointers, one for input (called `fin`) and one for output (called `fout`).

At line 10, you open the output file using `fopen` to a new file called `testfile.txt`. You check the error status of this line at line 11, exiting if a failure occurs. At line 13, you use a special function `fdopen` to associate an existing file descriptor with a stream. In this case, you associate in the standard-input descriptor with a new stream called `fin` (returned by `fdopen`). Whatever you now type in (standard-in) is routed to this file stream. Next, you enter a loop that attempts to read from the `fin` stream (standard-in) and write this out to the output stream (`fout`). At line 15, you read using `fgets` and check the return with `NULL`. The `NULL` appears when you close the descriptor (which is achieved through pressing Ctrl+D at the keyboard). The `line` read is then emitted to the output stream using `fputs`. Finally, when the input stream has closed, you exit the loop and close the two streams at lines 21 and 22.

Now take a look at another example of the `read` side, `fgets`. In this example (Listing 11.5), you read the contents of the test file using `fgets` and then `printf` it to standard-out.

LISTING 11.5 Reading Variable Length Strings from a File (on the CD-ROM at `./source/ch11/strin.c`)

```
1:      #include <stdio.h>
2:
3:      #define LEN    80
4:
5:      int main()
6:      {
7:        char line[LEN+1];
8:        FILE *fin;
9:
10:       fin = fopen( "testfile.txt", "r" );
11:       if ( fin == (FILE *)0 ) exit(-1);
12:
13:       while ( (fgets( line, LEN, fin )) != NULL ) {
14:
15:         printf( "%s", line );
16:
17:       }
18:
19:       fclose( fin );
20:
21:       return 0;
22:     }
```

In this example, you open the input file and create a new input stream handle called `fin`. You use this at line 13 to read variable-length strings from the file, and when one is read, you emit it to standard-out via `printf` at line 15.

This demonstrates writing and reading strings to and from a file, but what if your data is more structured than simple strings? If your strings are actually made up of lower level structures (such as integers, floating-point values, or other types), you can use another method to more easily deal with them. This is the next topic of discussion.

Consider the problem of reading and writing data that takes a regular form but consists of various data types (such as C structures). Say that you want to store an integer item (an `id`), two floating-point values (2D coordinates), and a string (an object name). Look first at the application that creates this file (see Listing 11.6). Note that in this example you ultimately deal with strings, but using the API functions, the ability to translate to the native data types is provided.

LISTING 11.6 Writing Structured Data in ASCII Format (on the CD-ROM at `./source/ch11/strucout.c`)

```
1:      #include <stdio.h>
2:
3:      #define MAX_LINE    40
4:
5:      #define FILENAME "myfile.txt"
6:
7:      typedef struct {
8:        int id;
9:        float x_coord;
10:       float y_coord;
11:       char name[MAX_LINE+1];
12:     } MY_TYPE_T;
13:
14:     #define MAX_OBJECTS    3
15:
16:     /* Initialize an array of three objects */
17:     MY_TYPE_T objects[MAX_OBJECTS]={
18:       { 0, 1.5, 8.4, "First-object" },
19:       { 1, 9.2, 7.4, "Second-object" },
20:       { 2, 4.1, 5.6, "Final-object" }
21:     };
22:
23:     int main()
24:     {
25:       int i;
```

```
26:         FILE *fout;
27:
28:         /* Open the output file */
29:         fout = fopen( FILENAME, "w" );
30:         if (fout == (FILE *)0) exit(-1);
31:
32:         /* Emit each of the objects, one per line */
33:         for ( i = 0 ; i < MAX_OBJECTS ; i++ ) {
34:
35:           fprintf( fout, "%d %f %f %s\n",
36:                    objects[i].id,
37:                    objects[i].x_coord, objects[i].y_coord,
38:                    objects[i].name );
39:
40:         }
41:
42:         fclose( fout );
43:
44:         return 0;
45:       }
```

Listing 11.6 illustrates another string method for creating data files. You create a test structure (lines 7–12) to represent the data that you're going to write and then read. You initialize this structure at lines 17–21 with three rows of data. Now you can turn to the application. This one turns out to be very simple. At lines 29–30, you open and then check the `fout` file handle and then perform a `for` loop to emit our data to the file. You use the `fprintf` API function to emit this data. The format of the `fprintf` call is to first specify the output file pointer, followed by a format string, and then zero or more variables to be emitted. The format string mirrors your data structure. You're emitting an `int` (`%d`), two floating-point values (`%f`), and then finally a string (`%s`). This converts all data to string format and writes it to the output file. Finally, you close the output file at line 42 with the `fclose` call.

You could have achieved this with a `sprintf` *call (to create your output string) and then written this out as follows:*

```
char line[81];
...
snprintf( line, 80, "%d %f %f %s\n",
             objects[i].id
             objects[i].x_coord, objects[i].y_coord,
             objects[i].name );
fputs( line, fout );
```

The disadvantage is that local space must be declared for the string being emitted. This would not be required with a call to `fprintf` *directly (the C library uses its own space internally).*

The prototypes for both the `fprintf` and `sprintf` are shown here:

```
int fprintf( FILE* stream, const char *format, ... );
int sprintf( char *str, const char *format, ... );
```

From the file created in Listing 11.6, you read this file in Listing 11.7. This function utilizes the `fscanf` function to both read and interpret the data. After opening the input file (lines 21–22), you loop and read the data while the end of file has not been found. You detect the end of file marker using the `feof` function at line 25. The `fscanf` function utilizes the input stream (`fin`) and the format to be used to interpret the data. This string is identical to that used to write the data out (see Listing 11.6, line 35).

After a line of data has been read, it's immediately printed to standard-out using the `printf` function at lines 32–35. Finally, the input file is closed using the `fclose` call at line 39.

LISTING 11.7 Reading Structured Data in ASCII Format (on the CD-ROM at `./source/ch11/strucin.c`)

```
 1:  #include <stdio.h>
 2:
 3:  #define MAX_LINE    40
 4:
 5:  #define FILENAME "myfile.txt"
 6:
 7:  typedef struct {
 8:    int id;
 9:    float x_coord;
10:    float y_coord;
11:    char name[MAX_LINE+1];
12:  } MY_TYPE_T;
13:
14:  int main()
15:  {
16:    int i;
17:    FILE *fin;
18:    MY_TYPE_T object;
19:
20:    /* Open the input file */
```

```
21:     fin = fopen( FILENAME, "r" );
22:     if (fin == (FILE *)0) exit(-1);
23:
24:     /* Read the records from the file and emit */
25:     while ( !feof(fin) ) {
26:
27:       fscanf( fin, "%d %f %f %s\n",
28:                &object.id,
29:                &object.x_coord, &object.y_coord,
30:                object.name );
31:
32:       printf("%d %f %f %s\n",
33:                object.id,
34:                object.x_coord, object.y_coord,
35:                object.name );
36:
37:     }
38:
39:     fclose( fin );
40:
41:     return 0;
42:   }
```

You could have achieved this functionality with an `sscanf` *call (to parse our input string).*

```
char line[81];
...
fgets( fin, 80, line );
sscanf( line, 80, "%d %f %f %s\n",
             objects[i].id
             objects[i].x_coord, objects[i].y_coord,
             objects[i].name );
```

The disadvantage is that local space must be declared for the parse to be performed on the input string. This would not be required with a call to `fscanf` *directly.*

The `fscanf` and `sscanf` function prototypes are both shown here:

```
int fscanf( FILE *stream, const char *format, ... );
int sscanf( const char *str, const char *format, ... );
```

All of the methods discussed thus far require that you are dealing with ASCII text data. In the next section, you'll see API functions that permit dealing with binary data.

For survivability, it's important to not leave files open over long durations of time. When I/O is complete, the file should be closed with `fclose` *(or at a minimum, flushed with* `fflush`*). This has the effect of writing any buffered data to the actual file.*

Reading and Writing Binary Data

This section looks at a set of library functions that provides the ability to deal with both binary and ASCII text data. The `fwrite` and `fread` functions provide the ability to deal not only with the I/O of objects, but also with arrays of objects. The prototypes of the `fwrite` and `fread` functions are provided here:

```
size_t fread( void *ptr, size_t size, size_t nmemb, FILE *stream );
size_t fwrite( const void *ptr, size_t size,
                  size_t nmemb, FILE *stream );
```

Now take a look at a couple of simple examples of `fwrite` and `fread` to explore their use (see Listing 11.8). In this first example, you emit the `MY_TYPE_T` structure first encountered in Listing 11.6.

LISTING 11.8 Using `fwrite` to Emit Structured Data (on the CD-ROM at `./source/ch11/binout.c`)

```
1:      #include <stdio.h>
2:
3:      #define MAX_LINE    40
4:
5:      #define FILENAME "myfile.bin"
6:
7:      typedef struct {
8:        int id;
9:        float x_coord;
10:       float y_coord;
11:       char name[MAX_LINE+1];
12:     } MY_TYPE_T;
13:
14:     #define MAX_OBJECTS    3
15:
16:     MY_TYPE_T objects[MAX_OBJECTS]={
```

```
17:         { 0, 1.5, 8.4, "First-object" },
18:         { 1, 9.2, 7.4, "Second-object" },
19:         { 2, 4.1, 5.6, "Final-object" }
20:       };
21:
22:       int main()
23:       {
24:         int i;
25:         FILE *fout;
26:
27:         /* Open the output file */
28:         fout = fopen( FILENAME, "w" );
29:         if (fout == (FILE *)0) exit(-1);
30:
31:         /* Write out the entire object's structure */
32:         fwrite( (void *)objects, sizeof(MY_TYPE_T), 3, fout );
33:
34:         fclose( fout );
35:
36:         return 0;
37:       }
```

What's interesting to note about Listing 11.8 is that a single `fwrite` emits the entire structure. You specify the object that you're emitting (variable object, passed as a void pointer) and then the size of a row in this structure (the type `MY_TYPE_T`) using the `sizeof` operator. You then specify the number of elements in the array of types (`3`) and finally the output stream to which you want this object to be written.

Take a look now at the invocation of this application (called `binout`) and a method for inspecting the contents of the binary file (see Listing 11.9). After executing the `binout` executable, the file `myfile.bin` is generated. Attempting to use the `more` utility to inspect the file results in a blank line. This is because the first character in the file is a `NULL` character, which is interpreted by `more` as the end. Next, you use the `od` utility (octal dump) to emit the file without interpreting it. You specify `-x` as the option to emit the file in hexadecimal format. (For navigation purposes, the integer `id` field has been underlined.)

LISTING 11.9 Inspecting the Contents of the Generated Binary File

```
$ ./binout
$ more myfile.bin
$ od -x myfile.bin
0000000 0000 0000 0000 3fc0 6666 4106 6946 7372
0000020 2d74 626f 656a 7463 0000 0000 0000 0000
```

```
0000040 0000 0000 0000 0000 0000 0000 0000 0000
0000060 0000 0000 0000 0000 0001 0000 3333 4113
0000100 cccd 40ec 6553 6f63 646e 6f2d 6a62 6365
0000120 0074 0000 0000 0000 0000 0000 0000 0000
0000140 0000 0000 0000 0000 0000 0000 0000 0000
0000160 0002 0000 3333 4083 3333 40b3 6946 616e
0000200 2d6c 626f 656a 7463 0000 0000 0000 0000
0000220 0000 0000 0000 0000 0000 0000 0000 0000
0000240 0000 0000 0000 0000
0000250
$
```

One important item to note about reading and writing binary data is the issue of portability and endianness. Consider that you create your binary data on a Pentium system, but the binary file is moved to a PowerPC system to read. The data will be in the incorrect byte order and therefore essentially corrupt. The Pentium uses little endian byte order (least significant byte first in memory), whereas the PowerPC uses big endian (most significant byte first in memory). For portability, endianness should always be considered when dealing with binary data. Also consider the use of host and network byte swapping functions, as discussed in Chapter 12, "Programming with Pipes."

Now it's time to take a look at reading this file using `fread`, but rather than reading it sequentially, read it in a nonsequential way (otherwise known as random access).This example reads the records of the file in reverse order. This requires the use of two new functions that permit you to seek into a file (`fseek`) and also rewind back to the start (`rewind`):

```
void rewind( FILE *stream );
int fseek( FILE *stream, long offset, int whence );
```

The `rewind` function simply resets the file read pointer back to the start of the file, whereas the `fseek` function moves you to a new position given an index. The `whence` argument defines whether the position is relative to the start of the file (`SEEK_SET`), the current position (`SEEK_CUR`), or the end of the file (`SEEK_END`). See Table 11.2. The `lseek` function operates like `fseek`, but instead on a file descriptor.

```
int lseek( FILE *stream, long offset, int whence );
```

In this example (Listing 11.10), you open the file using `fopen`, which automatically sets the `read` index to the start of the file. Because you want to read the last record first, you seek into the file using `fseek` (line 26). The index that you specify

TABLE 11.2 Function `fseek`/`lseek` `whence` Arguments

Name	Description
`SEEK_SET`	Moves the file position to the position defined by `offset`.
`SEEK_CUR`	Moves the file position the number of bytes defined by `offset` from the current file position.
`SEEK_END`	Moves the file position to the number of bytes defined by `offset` from the end of the file.

is twice the size of the record size (`MY_TYPE_T`). This puts you at the first byte of the third record, which is then read with the `fread` function at line 28. The `read` index is now at the end of the file, so you reset the `read` position to the top of the file using the `rewind` function.

You repeat this process, setting the file `read` position to the second element at line 38, and then read again with `fread`. The final step is reading the first element in the file. This requires no `fseek` because after the `rewind` (at line 48), you are at the top of the file. You can then `fread` the first record at line 50.

LISTING 11.10 Using `fread` and `fseek`/`rewind` to Read Structured Data (on the CD-ROM at `./source/ch11/nonseq.c`)

```
1:      #include <stdio.h>
2:
3:      #define MAX_LINE    40
4:
5:      #define FILENAME "myfile.txt"
6:
7:      typedef struct {
8:        int id;
9:        float x_coord;
10:       float y_coord;
11:       char name[MAX_LINE+1];
12:     } MY_TYPE_T;
13:
14:     MY_TYPE_T object;
15:
16:     int main()
17:     {
18:       int i;
19:       FILE *fin;
```

```
20:
21:         /* Open the input file */
22:         fin = fopen( FILENAME, "r" );
23:         if (fin == (FILE *)0) exit(-1);
24:
25:         /* Get the last entry */
26:         fseek( fin, (2 * sizeof(MY_TYPE_T)), SEEK_SET );
27:
28:         fread( &object, sizeof(MY_TYPE_T), 1, fin );
29:
30:         printf("%d %f %f %s\n",
31:                 object.id,
32:                 object.x_coord, object.y_coord,
33:                 object.name );
34:
35:         /* Get the second to last entry */
36:         rewind( fin );
37:
38:         fseek( fin, (1 * sizeof(MY_TYPE_T)), SEEK_SET );
39:
40:         fread( &object, sizeof(MY_TYPE_T), 1, fin );
41:
42:         printf("%d %f %f %s\n",
43:                 object.id,
44:                 object.x_coord, object.y_coord,
45:                 object.name );
46:
47:         /* Get the first entry */
48:         rewind( fin );
49:
50:         fread( &object, sizeof(MY_TYPE_T), 1, fin );
51:
52:         printf("%d %f %f %s\n",
53:                 object.id,
54:                 object.x_coord, object.y_coord,
55:                 object.name );
56:
57:         fclose( fin );
58:
59:         return 0;
60:       }
```

The process of reading the third record is illustrated graphically in Figure 11.1. It illustrates `fopen`, `fseek`, `fread`, and finally `rewind`.

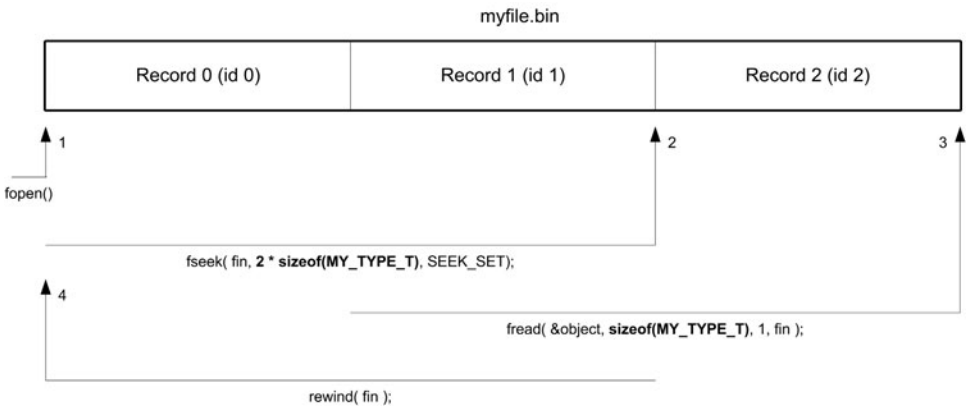

FIGURE 11.1 Nonsequential Reads in a Binary File

The function `ftell` provides the means to identify the current position. This function returns the current position as a long type and can be used to pass as the offset to `fseek` (with `SEEK_SET`) to reset to that position. The `ftell` prototype is provided here:

```
long ftell( FILE *stream );
```

An alternate API exists to `ftell` and `fseek`. The `fgetpos` and `fsetpos` provide the same functionality, but in a different form. Rather than an absolute position, an opaque type is used to represent the position (returned by `fgetpos`, passed into `fsetpos`). The prototypes for these functions are provided here:

```
int fgetpos( FILE *stream, fpos_t *pos );
int fsetpos( FILE *stream, fops_t *pos );
```

A sample code snippet of these functions is shown here:

```
fpos_t file_pos;
...
/* Get desired position */
fgetpos( fin, &file_pos );
...
rewind( fin );
/* Return to desired position */
fsetpos( fin, &file_pos );
```

It's recommended to use the `fgetpos` and `fsetpos` APIs over the `ftell` and `fseek` methods. Because the `ftell` and `fseek` methods don't abstract the details of the mechanism, the `fgetpos` and `fsetpos` functions are less likely to be deprecated in the future.

Base API

The `open`, `read`, and `write` functions can also be used for file I/O. The API differs somewhat, but this section also looks at how to switch between file and stream mode with `fdopen`.

These functions are referred to as the base API because they are the platform from which the standard I/O library is built.

The `open` function allows you to open or create a new file. Two variations are provided, with their APIs listed here:

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

The `pathname` argument defines the file (with path) to be opened or created (such as `temp.txt` or `/tmp/myfile.txt`). The `flags` argument is one of `O_RDONLY`, `O_WRONLY`, or `O_RDWR`. One or more of the flags shown in Table 11.3 might also be OR'd (bitwise OR operation) in, depending on the needs of the `open` call.

TABLE 11.3 Additional Flags for the `open` Function

Flag	Description
`O_CREAT`	Create the file if it doesn't exist.
`O_EXCL`	If used with `O_CREAT`, will return an error if the file already exists; otherwise the file is created.
`O_NOCTTY`	If the file descriptor refers to a TTY device, this process will not become the controlling terminal.
`O_TRUNC`	The file will be truncated (if it exists) and the length reset to zero if `write` privileges are permitted.
`O_APPEND`	The file pointer is repositioned to the end of the file prior to each `write`.

→

Flag	Description
O_NONBLOCK	Opens the file in nonblocking mode. Operations on the file will not block (such as read, write, and so on).
O_SYNC	write functions are blocked until the data is written to the physical device.
O_NOFOLLOW	Fail following symbolic links.
O_DIRECTORY	Fail the open if the file being opened is not a directory.
O_DIRECT	Attempts to minimize cache effects by doing I/O directly to/from user space buffers (synchronously as with O_SYNC).
O_ASYNC	Requests a signal when data is available on input or output of this file descriptor.
O_LARGEFILE	Request a large filesystem file to be opened on a 32-bit system whose size cannot be represented in 32 bits.

The third argument for the second open instance is a mode. This mode defines the permissions to be used when the file is created (used only with the flag O_CREAT). Table 11.4 lists the possible symbolic constants that can be OR'd together.

TABLE 11.4 Mode Arguments for the open System Call

Constant	Use
S_IRWXU	User has read/write/execute permissions.
S_IREAD	User has read permission.
S_IWRITE	User has write permission.
S_IEXEC	User has execute permission.
S_IRWXG	Group has read/write/execute permissions.
S_IRGRP	Group has read permission.
S_IWGRP	Group has write permission.
S_IXGRP	Group has execute permission.
S_IRWXO	Others have read/write/execute permissions.
S_IROTH	Others have read permission.
S_IWOTH	Others have write permission.
S_IXOTH	Others have execute permission.

To open a new file in the `tmp` directory, you could do the following:

```
int fd;
fd = open( "/tmp/newfile.txt", O_CREAT | O_WRONLY );
```

To instead open an existing file for read, you could `open` as follows:

```
int fd;
fd = open( "/tmp/newfile.txt", O_RDONLY );
```

Reading and writing to these files is done very simply with the `read` and `write` API functions:

```
ssize_t read( int fd, void *buf, size_t count );
ssize_t write( int fd, const void *buf, size_t count );
```

These are used simply with a buffer and a size to represent the number of bytes to read or write, such as:

```
unsigned char buffer[MAX_BUF+1];
int fd, ret;
...
ret = read( fd, (void *)buffer, MAX_BUF );
...
ret = write( fd, (void *)buffer, MAX_BUF );
```

You'll see more examples of these in Chapter 12 "Programming with Pipes" and Chapter 13 "Introduction to Sockets Programming." What's interesting here is that the same set of API functions to read and write data to a file can also be used for pipes and sockets. This represents a unique aspect of the UNIX-like operating systems, where many types of devices are represented as files. The result is that a common API can be used over a broad range of devices.

File and string handling are some of the strengths of the object-oriented scripting languages. This book explores two such languages (Ruby and Python) in Chapter 26, "Scripting with Ruby," and Chapter 27, "Scripting with Python."

Finally, a file descriptor can be attached to a stream by using the `fdopen` system call. This call has the following prototype:

```
FILE *fdopen( int filedes, const char *mode );
```

Therefore, if you've opened a device using the `open` function call, you can associate a stream with it using `fdopen` and then use stream system calls on the device (such as `fscanf` or `fprintf`). Consider the following example:

```
FILE *fp;
int fd;
fd = open( "/tmp/myfile.txt", O_RDWR );
fp = fdopen( fd, "rw" );
```

After this is done, you can use `read/write` with the `fd` descriptor or `fscanf/fprintf` with the `fp` descriptor.

One other useful API to consider is the `pread/pwrite` API. These functions require an offset into the file to read or write, but they do not affect the file pointer. These functions have the following prototype:

```
ssize_t pread( int filedes, void *buf, size_t nbyte, off_t offset );
ssize_t pwrite( int filedes, void *buf, size_t nbyte, off_t offset );
```

These functions require that the target be seekable (in other words, regular files) and used regularly for record I/O in databases.

Summary

In this chapter, the file handling APIs were discussed with examples provided for each. The character interfaces were first explored (`fputc`, `fgetc`), followed by the string interfaces (`fputs`, `fgets`). Some of the more structured methods for generating and parsing files were then investigated (such as the `fprintf` and `fscanf` functions), in addition to some of the other possibilities (`sprintf` and `sscanf`). Finally, the topics of binary files and random (nonsequential) access were discussed, including methods for saving and restoring file positions.

File Handling APIs

```
FILE *fopen( const char *filename, const char *mode );
FILE *fdopen( int filedes, const char *type );
int fputc( int c, FILE *stream );
int fgetc( FILE *stream );
int fputs( int c, FILE *stream );
char *fgets( char *s, int size, FILE *stream );
int fprintf( FILE* stream, const char *format, ... );
```

```
int sprintf( char *str, const char *format, ... );
int fscanf( FILE *stream, const char *format, ... );
int sscanf( const char *str, const char *format, ... );
void rewind( FILE *stream );
int fseek( FILE *stream, long offset, int whence );
int lseek( in filedes, long offset, int whence );
long ftell( FILE *stream );
int fgetpos( FILE *stream, fpos_t *pos );
int fsetpos( FILE *stream, fops_t *pos );
int fclose( FILE *stream );
int open( const char *pathname, int flags );
int open( const char *pathname, int flags, mode_t mode );
ssize_t read( int fd, void *buf, size_t count );
ssize_t write( int fd, const void *buf, size_t count );
ssize_t pread( int filedes, void *buf, size_t count, off_t offset );
ssize_t pwrite( int filedes, const void *buf,
                size_t count, off_t offset );
```

12 Programming with Pipes

In This Chapter

- Review of the Pipe Model of IPC
- Differences Between Anonymous Pipes and Named Pipes
- Creating Anonymous and Named Pipes
- Communicating Through Pipes
- Command-Line Creation and Use of Pipes

Introduction

This chapter explores the GNU/Linux pipes. The pipe model is an older but still useful mechanism for interprocess communication. It looks at what are known as half-duplex pipes and also named pipes. Each offers a first-in-first-out (FIFO) queuing model to permit communication between processes.

The Pipe Model

One way to visualize a pipe is a one-way connector between two entities. For example, consider the following GNU/Linux command:

```
ls -1 | wc -l
```

This command creates two processes, one for the `ls -1` and another for `wc -l`. It then connects the two together by setting the standard-input of the second process to the standard-output of the first process (see Figure 12.1). This has the effect of counting the number of files in the current subdirectory.

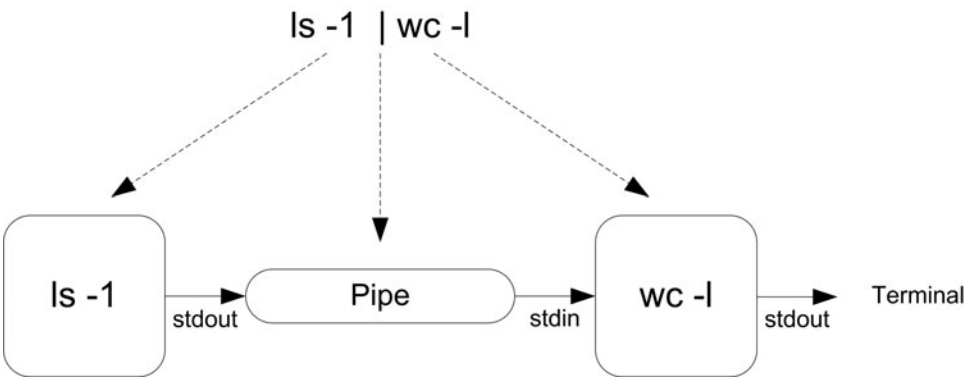

FIGURE 12.1 Simple pipe example.

This command, as illustrated in Figure 12.1, sets up a pipeline between two GNU/Linux commands. The `ls` command is performed, which generates output that is used as the input to the second command, `wc` (word count). This is a half-duplex pipe as communication occurs in one direction. The linkage between the two commands is facilitated by the GNU/Linux kernel, which takes care of connecting the two together. You can achieve this in applications as well, which this chapter demonstrates shortly.

Pipes and Named Pipes

A pipe, or half-duplex pipe, provides the means for a process to communicate with one of its ancestral subprocesses (of the anonymous variety). This is because no way exists in the operating system to locate the pipe (it's anonymous). Its most common use is to create a pipe at a parent process and then pass the pipe to the child so that they can communicate. Note that if full-duplex communication is required, the Sockets API should be considered instead.

Another type of pipe is called a *named pipe*. A named pipe works like a regular pipe but exists in the filesystem so that any process can find it. This means that processes not of the same ancestry are able to communicate with one another.

The following sections look at both half-duplex or anonymous pipes and named pipes. The chapter first takes a quick tour of pipes and then follows up with a more detailed look at the pipe API and GNU/Linux system-level commands that support pipes programming.

Whirlwind Tour

This section begins with a simple example of the pipe programming model. In this example, you create a pipe within a process, write a message to it, read the message back from the pipe, and then emit it (see Listing 12.1).

LISTING 12.1 Simple Pipe Example (on the CD-ROM at `./source/ch12/pipe1.c`)

```
1:      #include <unistd.h>
2:      #include <stdio.h>
3:      #include <string.h>
4:
5:      #define MAX_LINE        80
6:      #define PIPE_STDIN      0
7:      #define PIPE_STDOUT     1
8:
9:      int main()
10:     {
11:       const char *string={"A sample message."};
12:       int ret, myPipe[2];
13:       char buffer[MAX_LINE+1];
14:
15:       /* Create the pipe */
16:       ret = pipe( myPipe );
17:
18:       if (ret == 0) {
19:
20:         /* Write the message into the pipe */
21:         write( myPipe[PIPE_STDOUT], string, strlen(string) );
22:
23:         /* Read the message from the pipe */
24:         ret = read( myPipe[PIPE_STDIN], buffer, MAX_LINE );
25:
26:         /* Null terminate the string */
27:         buffer[ ret ] = 0;
28:
29:         printf("%s\n", buffer);
30:
31:       }
32:
33:       return 0;
34:     }
```

In Listing 12.1, you create your pipe using the `pipe` call at line 16. You pass in a two-element `int` array that represents your pipe. The pipe is defined as a pair of separate file descriptors, an input and an output. You can write to one end of the pipe and read from the other. The `pipe` API function returns zero on success. Upon return, the `myPipe` array contains two new file descriptors representing the input to the pipe (`myPipe[1]`) and the output from the pipe (`myPipe[0]`).

At line 21, you write your message to the pipe using the `write` function. You specify the `stdout` descriptor (from the perspective of the application, not the pipe). The pipe now contains the message and can be read at line 24 using the `read` function. Here again, from the perspective of the application, you use the `stdin` descriptor to read from the pipe. The `read` function stores what is read from the pipe in the `buffer` variable (argument three of the `read` function). You terminate it (add a `NULL` to the end) so that you can properly emit it at line 29 using `printf`. The pipe in this example is illustrated in Figure 12.2.

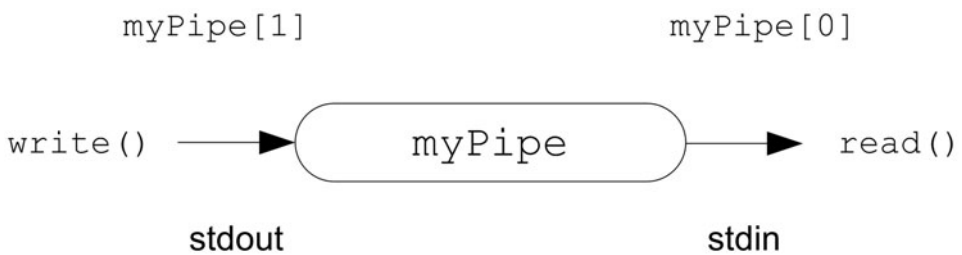

FIGURE 12.2 Half-duplex pipe example from Listing 12.1.

While this example was entertaining, communicating with yourself could be performed using any number of mechanisms. The detailed review looks at more complicated examples that provide communication between processes (both related and unrelated).

Detailed Review

While the `pipe` function is the majority of the pipe model, you need to understand a few other functions in their applicability toward pipe-based programming. Table 12.1 lists the functions that are detailed in this chapter.

This chapter also looks at some of the other functions that are applicable to pipe communication, specifically those that can be used to communicate using a pipe.

TABLE 12.1 API Functions for Pipe Programming

API Function	Use
`pipe`	Create a new anonymous pipe.
`dup`	Create a copy of a file descriptor.
`mkfifo`	Create a named pipe (`fifo`).

Remember that a pipe is nothing more than a pair of file descriptors, and therefore any functions that operate on file descriptors can be used. This includes but is not restricted to `select`, `read`, `write`, `fcntl`, `freopen`, *and such.*

pipe

The `pipe` API function creates a new pipe, represented by an array of two file descriptors. The `pipe` function has the following prototype:

```
#include <unistd.h>
int pipe( int fds[2] );
```

The `pipe` function returns 0 on success, or `-1` on failure, with `errno` set appropriately. On successful return, the `fds` array (which was passed by reference) is filled with two active file descriptors. The first element in the array is a file descriptor that can be read by the application, and the second element is a file descriptor that can be written to.

Now take a look at a slightly more complicated example of a pipe in a multiprocess application. In this application (see Listing 12.2), you create a pipe (line 14) and then fork your process into a parent and a child process (line 16). At the child, you attempt to read from the input file descriptor of your pipe (line 18), which suspends the process until something is available to read. When something is read, you terminate the string with a `NULL` and print out what was read. The parent simply writes a test string through the pipe using the `write` file descriptor (array offset 1 of the `pipe` structure) and then waits for the child to exit using the `wait` function.

Note that nothing is spectacular about this application except for the fact that the child process inherited the file descriptors that were created by the parent (using the `pipe` function) and then used them to communicate with one another. Recall that after the `fork` function is complete, the processes are independent (except that the child inherited features of the parent, such as the `pipe` file descriptors). Memory is separate, so the `pipe` method provides you with an interesting model to communicate between processes.

LISTING 12.2 Illustrating the Pipe Model with Two Processes (on the CD-ROM at `./source/ch12/fpipe.c`)

```
1:        #include <stdio.h>
2:        #include <unistd.h>
3:        #include <string.h>
4:        #include <wait.h>
5:
6:        #define MAX_LINE        80
7:
8:        int main()
9:        {
10:         int thePipe[2], ret;
11:         char buf[MAX_LINE+1];
12:         const char *testbuf={"a test string."};
13:
14:         if ( pipe( thePipe ) == 0 ) {
15:
16:           if (fork() == 0) {
17:
18:             ret = read( thePipe[0], buf, MAX_LINE );
19:             buf[ret] = 0;
20:             printf( "Child read %s\n", buf );
21:
22:           } else {
23:
24:             ret = write( thePipe[1], testbuf, strlen(testbuf) );
25:             ret = wait( NULL );
26:
27:           }
28:
29:         }
30:
31:         return 0;
32:       }
```

Note that so far these simple programs, have not discussed closing the pipe, because after the process finishes, the resources associated with the pipe are automatically freed. It's good programming practice, nonetheless, to close the descriptors of the pipe using the `close` call, as follows:

```
ret = pipe( myPipe );
...
close( myPipe[0] );
close( myPipe[1] );
```

If the write end of the pipe is closed and a process tries to read from the pipe, a zero is returned. This indicates that the pipe is no longer used and should be closed. If the read end of the pipe is closed and a process tries to write to it, a signal is generated. This signal (as discussed in Chapter 13, "Introduction to Sockets Programming") is called SIGPIPE. Applications that write to pipes commonly include a signal handler to catch just this situation.

dup AND dup2

The dup and dup2 calls are very useful functions that provide the ability to duplicate a file descriptor. They're most often used to redirect the stdin, stdout, or stderr of a process. The function prototypes for dup and dup2 are as follows:

```
#include <unistd.h>
int dup( int oldfd );
int dup2( int oldfd, int targetfd );
```

The dup function allows you to duplicate a descriptor. You pass in an existing descriptor, and it returns a new descriptor that is identical to the first. This means that both descriptors share the same internal structure. For example, if you perform an lseek (seek into the file) for one file descriptor, the file position is the same in the second. Use of the dup function is illustrated in the following code snippet:

```
int fd1, fd2;
...
fd2 = dup( fd1 );
```

Creating a descriptor prior to the fork call has the same effect as calling dup. The child process receives a duplicated descriptor, just like it would after calling dup.

The dup2 function is similar to dup but allows the caller to specify an active descriptor and the id of a target descriptor. Upon successful return of dup2, the new target descriptor is a duplicate of the first (targetfd = oldfd). Now take a look at a short code snippet that illustrates dup2:

```
int oldfd;
oldfd = open("app_log", (O_RDWR | O_CREATE), 0644 );
dup2( oldfd, 1 );
close( oldfd );
```

In this example, you open a new file called app_log and receive a file descriptor called fd1. You call dup2 with oldfd and 1, which has the effect of replacing the file

descriptor identified as `1` (`stdout`) with `oldfd` (the newly opened file). Anything written to `stdout` now goes instead to the file named `app_log`. Note that you close `oldfd` directly after duplicating it. This doesn't close your newly opened file, because file descriptor 1 now references it.

Now take a look at a more complex example. Recall that earlier in the chapter you investigated pipelining the output of `ls -1` to the input of `wc -l`. Now this example is explored in the context of a C application (see Listing 12.3).

You begin in Listing 12.3 by creating your pipe (line 9) and then forking the application into the child (lines 13–16) and parent (lines 20–23). In the child, we begin by closing the `stdout` descriptor (line 13). The child here provides the `ls -1` functionality and does not write to `stdout` but instead to the input to your pipe (redirected using `dup`). At line 14, you use `dup2` to redirect the `stdout` to your pipe (`pfds[1]`). After this is done, you close your input end of the pipe (as it will never be used). Finally, you use the `execlp` function to replace the child's image with that of the command `ls -1`. After this command executes, any output that is generated is sent to the input.

Now take a look at the receiving end of the pipe. The parent plays this role and follows a very similar pattern. You first close the `stdin` descriptor at line 20 (because you will accept nothing from it). Next, you use the `dup2` function again (line 21) to make the `stdin` the output end of the pipe. This is done by making file descriptor 0 (normal `stdin`) the same as `pfds[0]`. You close the `stdout` end of the pipe (`pfds[1]`) because you won't use it here (line 22). Finally, you `execlp` the command `wc -l`, which takes as its input the contents of the pipe (line 23).

Listing 12.3 Pipelining Commands in C (on the CD-ROM at `./source/ch12/dup.c`)

```
1:        #include <stdio.h>
2:        #include <stdlib.h>
3:        #include <unistd.h>
4:
5:        int main()
6:        {
7:          int pfds[2];
8:
9:          if ( pipe(pfds) == 0 ) {
10:
11:           if ( fork() == 0 ) {
12:
13:             close(1);
14:             dup2( pfds[1], 1 );
15:             close( pfds[0] );
16:             execlp( "ls", "ls", "-1", NULL );
```

```
17:
18:           } else {
19:
20:             close(0);
21:             dup2( pfds[0], 0 );
22:             close( pfds[1] );
23:             execlp( "wc", "wc", "-l", NULL );
24:
25:           }
26:
27:         }
28:
29:         return 0;
30:       }
```

What's important to note in this application is that your child process redirects its output to the input of the pipe, and the parent redirects its input to the output of the pipe—a very useful technique that is worth remembering.

mkfifo

The `mkfifo` function is used to create a file in the filesystem that provides FIFO functionality (otherwise known as a *named pipe*). Pipes that this chapter has discussed thus far are anonymous pipes. They're used exclusively between a process and its children. Named pipes are visible in the filesystem and therefore can be used by any (related or unrelated) process. The function prototype for `mkfifo` is defined as follows:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo( const char *pathname, mode_t mode );
```

The `mkfifo` command requires two arguments. The first (`pathname`) is the special file in the filesystem that is to be created. The second (`mode`) represents the read/write permissions for the FIFO. The `mkfifo` command returns 0 on success or `-1` on error (with `errno` filled appropriately). Take a look at an example of creating a `fifo` using the `mkfifo` function.

```
int ret;
...
ret = mkfifo( "/tmp/cmd_pipe", S_IFIFO | 0666 );
if (ret == 0) {
  // Named pipe successfully created
```

```
} else {
  // Failed to create named pipe
}
```

In this example, you create a `fifo` (named pipe) using the file `cmd_pipe` in the `/tmp` subdirectory. You can then open this file for `read` or `write` to communicate through it. After you open a named pipe, you can read from it using the typical I/O commands. For example, here's a snippet reading from the pipe using `fgets`:

```
pfp = fopen( "/tmp/cmd_pipe", "r" );
...
ret = fgets( buffer, MAX_LINE, pfp );
```

You can write to the pipe for this snippet using:

```
pfp = fopen( "/tmp/cmd_pipe", "w+ );
...
ret = fprintf( pfp, "Here's a test string!\n" );
```

What's interesting about named pipes, which is explored shortly in the discussion of the `mkfifo` system command, is that they work in what is known as a rendezvous model. A reader is unable to open the named pipe unless a writer has actively opened the other end of the pipe. The reader is blocked on the `open` call until a writer is present. Despite this limitation, the named pipe can be a useful mechanism for interprocess communication.

System Commands

Now it's time to take a look at a system command that is related to the pipe model for IPC. The `mkfifo` command, just like the `mkfifo` API function, allows you to create a named pipe from the command line.

mkfifo

The `mkfifo` command is one of two methods for creating a named pipe (`fifo` special file) at the command line. The general use of the `mkfifo` command is as follows:

```
mkfifo [options] name
```

where `[options]` are `-m` for mode (permissions) and `name` is the name of the named pipe to create (including path if needed). If permissions are not specified, the default is `0644`. Here's a sample use, creating a named pipe in `/tmp` called `cmd_pipe`:

```
$ mkfifo /tmp/cmd_pipe
```

You can adjust the options simply by specifying them with the `-m` option. Here's an example setting the permissions to `0644` (but deleting the original first):

```
$ rm cmd_pipe
$ mkfifo -m 0644 /tmp/cmd_pipe
```

After the permissions are created, you can communicate through this pipe via the command line. Consider the following scenario. In one terminal, you attempt to read from the pipe using the `cat` command:

```
$ cat cmd_pipe
```

Upon typing this command, you are suspended awaiting a writer opening the pipe. In another terminal, you write to the named pipe using the `echo` command, as follows:

```
$ echo Hi > cmd_pipe
```

When this command finishes, the reader wakes up and finishes (here's the complete reader command sequence again for clarity):

```
$ cat cmd_pipe
Hi
$
```

This illustrates that named pipes can be useful not only in C applications, but also in scripts (or combinations).

Named pipes can also be created with the `mknod` command (along with many other types of special files). You can create a named pipe (as with `mkfifo` before) as follows:

```
$ mknod cmd_pipe p
```

where the named pipe `cmd_pipe` is created in the current subdirectory (with type as `p` for named pipe).

Summary

This chapter was a very quick review of anonymous and named pipes. You reviewed application and command-line methods for creating pipes and also reviewed typical I/O mechanisms for communicating through them. You also reviewed the ability to redirect I/O using the `dup` and `dup2` commands. While useful for pipes, these commands are useful in many other scenarios as well (wherever a file descriptor is used, such as a socket or file).

Pipe Programming APIs

```
#include <unistd.h>
int pipe( int filedes[2] );
int dup( int oldfd );
int dup2( int oldfd, int targetfd );
int mkfifo( const char *pathname, mode_t mode );
```

13 Introduction to Sockets Programming

In This Chapter

- Understand the Sockets Programming Paradigm
- Learn the BSD4.4 Sockets API
- See Sample Source for a TCP/IP Server and Client
- Explore the Various Capabilities of Sockets (Control, I/O, Notification)
- Investigate Socket Patterns That Illustrate Sockets API Use
- Explore Other Transport Protocols such as SCTP
- Examine Sockets Programming in Other Languages

Introduction

This chapter takes a quick tour of Sockets programming. It discusses the Sockets programming paradigm, elements of Sockets applications, and the Sockets API. The Sockets API allows you to develop applications that communicate over a network. The network can be a local private network or the public Internet. An important item to note about Sockets programming is that it's neither operating system specific nor language specific. Sockets applications can be written in the Ruby scripting language on a GNU/Linux host or in C on an embedded controller. This freedom and flexibility are the reasons that the BSD4.4 Sockets API is so popular.

Layered Model of Networking

Sockets programming uses the layered model of packet communication (see Figure 13.1). At the top is the application layer, which is where applications exist (those that utilize Sockets for communication). Below the application layer you define the Sockets layer. This isn't actually a layer, but it is shown here simply to illustrate where the API is located. The Sockets layer sits on top of the transport layer. The transport layer provides the transport protocols. Next is the network layer, which provides among other things routing over the Internet. This layer is occupied by the Internet Protocol, or IP. Finally, you find the physical layer driver, which provides the means to introduce packets onto the physical network.

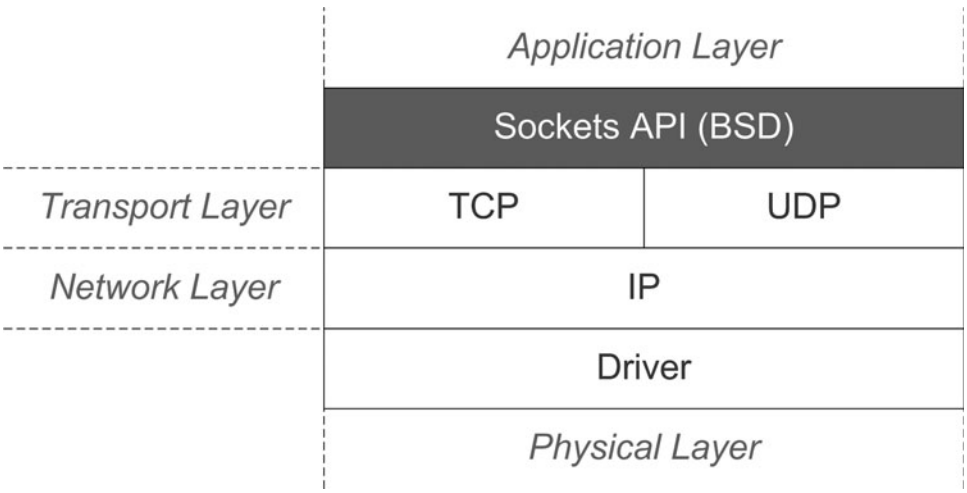

FIGURE 13.1 Layered model of communication.

Sockets Programming Paradigm

The Sockets paradigm involves a number of different elements that must be understood to use it properly. This section looks at the Sockets paradigm in a hierarchical fashion.

At the top of the hierarchy is the host. This is a source or destination node on a network to or from which packets are sent or received. (Technically, you would refer to interfaces as the source or destination, because a host might provide multiple interfaces, but this chapter is going to keep it simple.) The host implements a

set of protocols. These protocols define the manner in which communication occurs. Within each protocol is a set of ports. Each port defines an endpoint (the final source or destination). See Table 13.1 for a list of these elements (and Figure 13.2 for a graphical view of these relationships).

TABLE 13.1 Sockets Programming Element Hierarchy

Element	Description
Host (Interface)	Network address (a reachable network node)
Protocol	Specific protocol (such as TCP or UDP)
Port	Client or server process endpoint

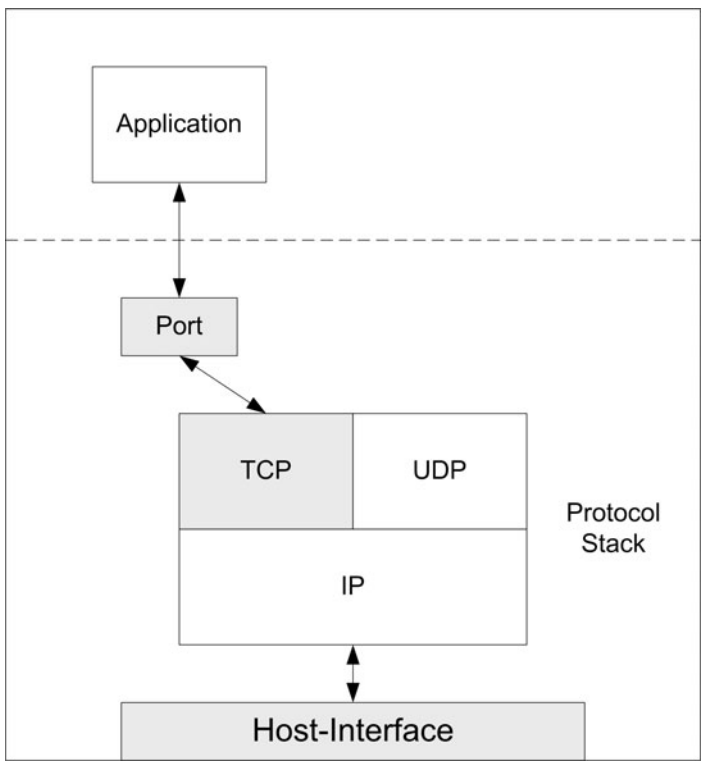

FIGURE 13.2 Graphical view of host/protocol/port relationship.

Hosts

Hosts are identified by addresses, and for IP these are called IP addresses. An IPv4 address (of the version 4 class) is defined as a 32-bit address. This address is represented by four 8-bit values. A sample address is illustrated as follows:

```
192.168.1.1    or    0xC0A80101
```

The first value shows the more popular form of IPv4 addresses, which is easily readable. The second notation is simply the first address in hexadecimal format (32 bits wide).

Protocol

The protocol specifies the details of communication over the socket. The two most common protocols used are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). TCP is a stream-based reliable protocol, and UDP is a datagram (message)-based protocol that can be unreliable. This chapter provides additional details of these protocols.

Port

The port is the endpoint for a given process (interface) for a protocol. This is the application's interface to the Socket interface. Ports are unique on a host (not interface) for a given protocol. Ports are commonly called "bound" when they are attached to a given socket.

Ports are numbers that are split basically into two ranges. Port numbers below 1024 are reserved for well-known services (called well-known addresses), assigned by the IETF. Port numbers above 1024 are typically used by applications.

The original intent of service port numbers (such as FTP, HTTP, and DNS) was that they fall below port number 1024. Of course, the number of services exceeded that number long ago. Now, many system services occupy the port number space greater than 1024 (for example, NFS at port number 2049 and X11 at port number 6000).

Addressing

From this discussion, you can see that a *tuple* uniquely identifies an endpoint from all other endpoints on a network. Consider the following tuple:

```
{ tcp, 192.168.1.1, 4097 }
```

This defines the endpoint on the host identified by the address `192.168.1.1` with the port `4097` using the TCP protocol.

The Socket

Simply put, a socket is an endpoint of a communications channel between two applications. An example of this is defined as two tuples:

```
{ tcp, 192.168.1.1, 4097 }
{ tcp, 10.0.0.1, 5820 }
```

The first item to note here is that a socket is an association of two endpoints that share the same protocol. The IP addresses are different here, but they don't have to be. You can communicate via sockets in the same host. The port numbers are also different here, but they can be the same unless they exist on the same host. Port numbers assigned by the TCP/IP stack are called *ephemeral ports*. This relationship is shown visually in Figure 13.3.

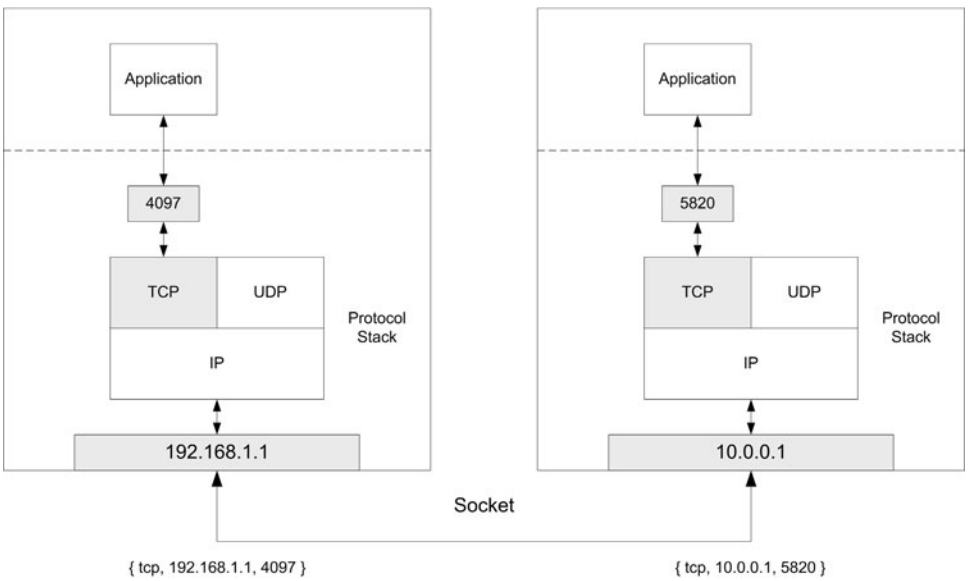

FIGURE 13.3 Visualization of a socket between two hosts.

Client/Server Model

In most Sockets applications, you have a server (responds to requests and provides responses) and a client (makes requests to the server). The Sockets API (which you explore in the next section) provides commands that are specific to clients and to servers. Figure 13.4 illustrates two simple applications that implement a client and a server.

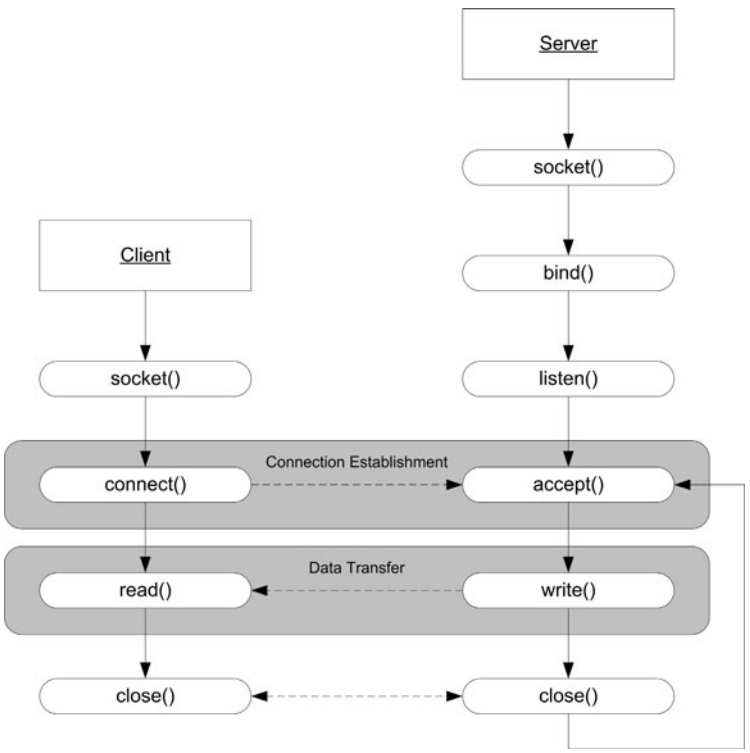

FIGURE 13.4 Client/server symmetry in Sockets applications.

The first step in a Sockets application is the creation of a socket. The socket is the communication endpoint that is created by the `socket` call. Note that in the sample flow (see Figure 13.4) both the server and client perform this step.

The server requires a bit more setup as part of registering a service to the host. The `bind` call binds an address and port to the server so that it's known. Letting the system choose the port can result in a service that can be difficult to find. If you choose the port, you know what it is. After you've bound the port, you call the `listen` function for the server. This makes the server accessible (puts it in the `listen` mode).

You establish the socket next, using `connect` at the client and `accept` at the server. The `connect` call starts what's known as the three-way handshake, with the purpose of setting up a connection between the client and server. At the server, the `accept` call creates a new server-side client socket. After `accept` finishes, a new socket connection exists between the client and server, and data flow can occur.

In the data transfer phase, you have an established socket through which communication can occur. Both the client and server can `send` and `recv` data asynchronously.

Finally, you can sever the connection between the client and server using the `close` call. This can occur asynchronously, but upon one endpoint closing the socket, the other side automatically receives an indication of the closure.

Sample Application

Now that you have a basic understanding of Sockets, you can look at a sample application that illustrates some of the functions available in the Sockets API. This section looks at the Sockets API from the perspective of two applications, a client and server, that implement the Daytime protocol. This protocol server is ASCII based and simply emits the current date and time when requested by a client. The client connects to the server and emits what is read. This implements the basic flow shown previously in Figure 13.2.

Daytime Server

Take a look at a C language server that implements the Daytime protocol. Recall that the Daytime server simply emits the current date and time in ASCII string format through the socket to the client. Upon emitting the data, the socket is closed, and the server awaits a new client connection. Now that you understand the concept behind Daytime protocol server, it's time to look at the actual implementation (see Listing 13.1).

LISTING 13.1 Daytime Server Written in the C Language (on the CD-ROM at `./source/ch13/dayserv.c`)

```
1:      #include <sys/socket.h>
2:      #include <arpa/inet.h>
3:      #include <stdio.h>
4:      #include <time.h>
5:      #include <string.h>
6:      #include <unistd.h>
7:
```

```
8:      #define MAX_BUFFER        128
9:      #define DAYTIME_SERVER_PORT    13
10:
11:     int main ( void )
12:     {
13:      int serverFd, connectionFd;
14:      struct sockaddr_in servaddr;
15:      char timebuffer[MAX_BUFFER+1];
16:      time_t currentTime;
17:
18:      serverFd = socket( AF_INET, SOCK_STREAM, 0 );
19:
20:      memset( &servaddr, 0, sizeof(servaddr) );
21:      servaddr.sin_family = AF_INET;
22:      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
23:      servaddr.sin_port = htons(DAYTIME_SERVER_PORT);
24:
25:      bind( serverFd,
26:         (struct sockaddr *)&servaddr, sizeof(servaddr) );
27:
28:      listen( serverFd, 5 );
29:
30:      while ( 1 ) {
31:
32:       connectionFd = accept( serverFd,
33:                   (struct sockaddr *)NULL, NULL );
34:
35:       if (connectionFd >= 0) {
36:
37:        currentTime = time(NULL);
38:        snprintf( timebuffer, MAX_BUFFER,
39:             "%s\n", ctime(&currentTime) );
40:
41:        write( connectionFd, timebuffer, strlen(timebuffer) );
42:        close( connectionFd );
43:
44:       }
45:
46:      }
47:
48:     }
```

Lines 1–6 include the header files for necessary types, symbolic, and function APIs. This includes not only the socket interfaces, but also `time.h`, which provides an interface to retrieve the current time. You specify the maximum size of the buffer that you operate upon using the symbolic constant `MAX_BUFFER` at line 8. The next symbolic constant at line 9, `DAYTIME_SERVER_PORT`, defines the port number to which you attach this socket server. This enables you to define the well-known port for the Daytime protocol (`13`).

You declare the `main` function at line 11, and then a series of variables are created in lines 13–16. You create two Socket identifiers (line 13), a socket address structure (line 14), a buffer to hold the string time (line 15), and the GNU/Linux time representation structure (line 16).

The first step in any Sockets program is to create the socket using the `socket` function (line 18). You specify that you're creating an IP socket (using the `AF_INET` domain) using a reliable stream protocol type (`SOCK_STREAM`). The zero as the third argument specifies to use the default protocol of the stream type, which is TCP.

Now that you have your socket, you bind an address and a port to it (lines 20–26). At line 20, you initialize the address structure by setting all elements to zero. You specify the socket domain again with `AF_INET` (it's an IPv4 socket). The `s_addr` element specifies an address, which in this case is the address from which you accept incoming socket connections. The special symbol `INADDR_ANY` says that you accept incoming connections from any available interface on the host. You then define the port to use, your prior symbolic constant `DAYTIME_SERVER_PORT`. The `htonl` (host-to-network-long) and `htons` (host-to-network-short) take care of ensuring that the values provided are in the proper byte order for network packets. The final step is using the `bind` function to bind the address structure previously created with your socket. The socket is now bound with the address, which identifies it in the network stack namespace.

The Internet operates in big endian, otherwise known as network byte order. Hosts operate in host byte order, which, depending upon architecture, can be either big or little endian. For example, the PowerPC architecture is big endian, and the Intel x86 architecture is little endian. This is a small performance advantage to big endian architectures because they need not perform any byte-swapping to change from host byte order to network byte order (they're already the same).

Before a client can connect to the socket, you must call the `listen` function (line 28). This tells the protocol stack that you're ready to accept connections (a maximum of five pending connections, per the argument to `listen`).

You enter an infinite loop at lines 30–46 to accept client connections and provide them the current time data. At lines 32–33, you call the `accept` function with your socket (`serverFd`) to accept a new client connection. When a client connects to you, the network stack creates a new socket representing the end of the connection and returns this socket from the `accept` function. With this new client socket (`connectionFd`), you can communicate with the peer client. Note that the existing server socket is untouched, allowing other connections to be received over it.

At line 35, you check the return socket to see if it's valid (otherwise, an error has occurred, and you ignore this client socket). If valid, you grab the current time at lines 37–39. You use the GNU/Linux `time` function to get the current time (the number of seconds that have elapsed from January 1, 1970). Passing this value to function `ctime` converts it into a human-readable format, which is used by `sprintf` to construct a response string. You send this to the peer client using the `connectionFd` socket using the `write` function. You pass your socket descriptor, the string to write (`timebuffer`), and its length. Finally, you close your client socket using the `close` function, which ends communication with that particular peer.

The loop then continues back to line 32, awaiting a new client connection with the `accept` function. When a new client connects, the process starts all over again.

From GNU/Linux, you can compile this application using GCC and execute it as follows (`filename server.c`):

```
[root@mtjones]$ gcc -o server server.c -Wall
[root@mtjones]$ ./server
```

When executing socket applications that bind to well-known ports (those under 1024), you must start from root. Otherwise, the application fails with the inability to bind to the reserved port number.

You can now test this server very simply using the Telnet application available in GNU/Linux. As shown, you Telnet to the local host (identified by `localhost`) and port 13 (the port you registered previously in the server). The Telnet client connects to the server and then prints out what was sent to it (the time is shown in bold).

```
$ telnet localhost 13
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Sat Jan 17 13:33:57 2004
Connection closed by foreign host.
[root@mtjones]$
```

The final item to note is that after the time is received, you see the message reported to you from Telnet: "Connection closed by foreign host." Recall from the server source in Listing 13.1 that after the `write` has completed (sending the time to the client), the socket `close` is immediately performed. The Telnet application is reporting this event to you so that you know the socket is no longer active. You can reproduce Telnet's operation with a socket client; this is investigated next.

Daytime Client

The Daytime protocol client is shown in Listing 13.2. This section avoids discussion of the source preliminaries and goes directly to the Sockets aspect of the application. As in the server, the first step in building a Sockets application is the creation of a socket (of the TCP variety using `SOCK_STREAM`) using the `socket` function (at line 16).

Recall in the server application that you build an address structure (`servaddr`) that is then bound to the socket representing the service. In the client, you also build an address structure, but in this case it's to define to whom you're connecting (Listing 13.2, lines 18–21). Note the similarities here between the server address structure creation (shown in Listing 13.1). The only difference is that the interface address is specified here in the client as `localhost`, where in the server you specify the wildcard to accept connections from any available interface.

Now that you have your socket and an address structure initialized with your destination, you can connect your socket to the server. This is done with the `connect` function shown in lines 23–24. In the `connect` function, you pass the socket descriptor (`connectionFd`), your address structure (`servaddr`), and its size. When this function returns, either you have connected to the server or an error has occurred. To minimize code size, the error check is omitted here, but the error code should be checked upon return from `connect` to ensure that the socket is truly connected.

Now that you are connected to the server, you perform a socket `read` function. This allows you to read any data that has been sent to you. Given the Daytime protocol, you know as a client that the server immediately sends you the current date and time. Therefore, you immediately read from the socket and store the contents into `timebuffer`. This is then null-terminated, and the result printed to standard-out. If you read from the socket and no characters are received (or an error occurs, indicated by a `-1` return), then you know that the server has closed the connection, and you exit gracefully. The next step is closure of your half of the socket, shown at line 34 using the `close` function.

LISTING 13.2 Daytime Client Written in the C Language (on the CD-ROM at `./source/ch13/daycli.c`)

```
1:      #include <sys/socket.h>
2:      #include <arpa/inet.h>
3:      #include <stdio.h>
4:      #include <unistd.h>
5:      #include <time.h>
6:
7:      #define MAX_BUFFER       128
8:      #define DAYTIME_SERVER_PORT   13
9:
10:     int main ()
11:     {
12:      int connectionFd, in, index = 0.limit = MAX_BUFFER;
13:      struct sockaddr_in servaddr;
14:      char timebuffer[MAX_BUFFER+1];
15:
16:      connectionFd = socket(AF_INET, SOCK_STREAM, 0);
17:
18:      memset(&servaddr, 0, sizeof(servaddr));
19:      servaddr.sin_family = AF_INET;
20:      servaddr.sin_port = htons(DAYTIME_SERVER_PORT);
21:      servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
22:
23:      connect(connectionFd,
24:          (struct sockaddr *)&servaddr, sizeof(servaddr));
25:
26:      while ((in=read(connectionFd, &timebuffer[index], limit))
         > 0) {
27:       index += in;
28:       limit -= in;
29:      }
30:
31:      timebuffer[index] = 0;
32:      printf("\n%s\n", timebuffer);
33:
34:      close(connectionFd);
35:
36:      return(0);
37:     }
```

Sockets API Summary

The networking API for C provides a mixed set of functions for the development of client and server applications. Some functions are used by only server-side sockets, whereas others are used solely by client-side sockets (most are available to both).

Creating and Destroying Sockets

You need to create a socket as the first step of any Sockets-based application. The `socket` function provides the following prototype:

```
int socket( int domain, int type, int protocol );
```

The `socket` object is represented as a simple integer and is returned by the `socket` function. Three parameters must be passed to define the type of socket to be created. Right now, you are interested primarily in stream (TCP) and datagram (UDP) sockets, but many other types of sockets can be created. In addition to stream and datagram, a raw socket is also illustrated by the following code snippets:

```
myStreamSocket = socket( AF_INET, SOCK_STREAM, 0 );
myDgramSocket = socket( AF_INET, SOCK_DGRAM, 0 );
myRawSocket = socket( AF_INET, SOCK_RAW, IPPROTO_RAW );
```

The `AF_INET` symbolic constant indicates that you are using the IPv4 Internet protocol. After this, the second parameter (`type`) defines the semantics of communication. For stream communication (using TCP), you use the `SOCK_STREAM` type, and for datagram communication (using UDP), you specify `SOCK_DGRAM`. The third parameter can define a particular protocol to use, but only the types exist for stream and datagram, so this third parameter is left as zero in those cases.

When you're finished with a socket, you must close it. The `close` prototype is defined as:

```
int close( sock );
```

After `close` is called, no further data can be received through the socket. Any data queued for transmission is given some amount of time to be sent before the connection physically closes.

Note in these examples that the `read` *and* `write` *calls are used identically to the file I/O examples shown in Chapter 11, "File Handling in GNU/Linux." One of the interesting features of UNIX (and GNU/Linux) is that many types of devices are represented as files. After a socket is open, you can treat it just like a file or pipe (for* `read`, `write`, `accept`, *and so on).*

Socket Addresses

For socket communication over the Internet (domain `AF_INET`), you use the `sockaddr_in` structure for naming purposes.

```
struct sockaddr_in {
        int16_t sin_family;
        uint16_t sin_port;
        struct in_addr sin_addr;
        char sin_zero[8];
};
struct in_addr {
        uint32_t s_addr;
};
```

For Internet communication, you use `AF_INET` solely for `sin_family`. Field `sin_port` defines your specified port number in network byte order. Therefore, you must use `htons` to load the port and `ntohs` to read it from this structure. Field `sin_addr` is, through `s_addr`, a 32-bit field that represents an IPv4 Internet address. Recall that IPv4 addresses are 4-byte addresses. Often the `sin_addr` is set to `INADDR_ANY`, which is the wildcard. When you're accepting connections (server socket), this wildcard says you accept connections from any available interface on the host. For client sockets, this is commonly left blank. For a client, if you set `sin_addr` to the IP address of a local interface, this restricts outgoing connections to that interface.

Now take look at a quick example of addressing for both a client and a server. First, in this example you create the socket address (later to be bound to your server socket) that permits incoming connections on any interface and port 48000.

```
int servsock;
struct sockaddr_in servaddr;
servsock = socket( AF_INET, SOCK_STREAM, 0);
memset( &servaddr, 0, sizeof(servaddr) );
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons( 48000 );
servaddr.sin_addr.s_addr = inet_addr( INADDR_ANY );
```

Next, you create a socket address that permits a client socket to connect to your previously created server socket.

```
int clisock;
struct sockaddr_in servaddr;
clisock = socket( AF_INET, SOCK_STREAM, 0);
memset( &servaddr, 0, sizeof(servaddr) );
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons( 48000 );
servaddr.sin_addr.s_addr = inet_addr( "192.168.1.1" );
```

Note the similarities between these two code segments. The difference, as you see later, is that the server uses the address to bind to itself as an advertisement. The client uses this information to define to whom it wants to connect.

Socket Primitives

In this section, you look at a number of other important server-side socket control primitives.

bind

The `bind` function provides a local naming capability to a socket. This can be used to name either client or server sockets, but it is used most often in the server case. The `bind` function is provided by the following prototype:

```
int bind( int sock, struct sockaddr *addr, int addrLen );
```

The socket to be named is provided by the `sock` argument, and the address structure previously defined is defined by `addr`. Note that the structure here differs from the address structure discussed previously. The `bind` function can be used with a variety of different protocols, but when you are using a socket created with `AF_INET`, you must use the `sockaddr_in`. Therefore, as shown in the following example, you cast your `sockaddr_in` structure as `sockaddr`.

```
err = bind( servsock, (struct sockaddr *)&servaddr,
sizeof(servaddr));
```

Using the address structure created in the server example in the previous address section, you bind the name defined by `servaddr` to our server socket `servsock`.

Recall that a client application can also call `bind` to name the client socket. This isn't used often, because the Sockets API dynamically assigns a port to us.

listen

Before a server socket can accept incoming client connections, it must call the `listen` function to declare this willingness. The `listen` function is provided by the following function prototype:

```
int listen( int sock, int backlog );
```

The `sock` argument represents the previously created server socket, and the `backlog` argument represents the number of outstanding client connections that might be queued. Within GNU/Linux, the `backlog` parameter (post 2.2 kernel version) represents the number of established connections pending on `accept` for the application layer protocol. Other operating systems might treat this differently.

accept

The `accept` call is the final call made by servers to accept incoming client connections. Before `accept` can be called, the server socket must be created, a name must be bound to it, and `listen` must be called. The `accept` function returns a socket descriptor for a client connection and is provided by the following function prototype:

```
int accept( int sock, struct sockaddr *addr, int *addrLen );
```

In practice, two examples of `accept` are commonly seen. The first represents the case in which you need to know who connected to you. This requires the creation of an address structure that is not initialized.

```
struct sockaddr_in cliaddr;
int cliLen;
cliLen = sizeof( struct sockaddr_in );
clisock = accept( servsock, (struct sockaddr *)cliaddr, &cliLen );
```

The call to `accept` blocks until a client connection is available. Upon return, the `clisock` return value contains the value of the new client socket, and `cliaddr` represents the address for the client peer (host address and port number).

The alternate example is commonly found when the server application isn't interested in the client information. This one typically appears as follows:

```
cliSock = accept( servsock, (struct sockaddr *)NULL, NULL );
```

In this case, `NULL` is passed for the address structure and length. The `accept` function then ignores these parameters.

connect

The connect function is used by client Sockets applications to connect to a server. Clients must have created a socket and then defined an address structure containing the host and port number to which they want to connect. The connect function is provided by the following function prototype:

```
int connect( int sock, (struct sockaddr *)servaddr, int addrLen );
```

The sock argument represents the client socket, created previously with the Sockets API function. The servaddr structure is the server peer to which you want to connect (as illustrated previously in the "Socket Addresses" section of this chapter). Finally, you must pass in the length of your servaddr structure so that connect knows you are passing in a sockaddr_in structure. The following code shows a complete example of connect:

```
int clisock;
struct sockaddr_in servaddr;
clisock = socket( AF_INET, SOCK_STREAM, 0);
memset( &servaddr, 0, sizeof(servaddr) );
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons( 48000 );
servaddr.sin_addr.s_addr = inet_addr( "192.168.1.1" );
connect( clisock, (struct sockaddr_in *)&servaddr, sizeof(servaddr) );
```

The connect function blocks until either an error occurs or the three-way handshake with the server finishes. Any error is returned by the connect function.

Sockets I/O

A variety of API functions exist to read data from a socket or write data to a socket. Two of the API functions (recv, send) are used exclusively by sockets that are connected (such as stream sockets), whereas an alternative pair (recvfrom, sendto) is used exclusively by sockets that are unconnected (such as datagram sockets).

Connected Socket Functions

The send and recv functions are used to send a message to the peer socket endpoint and to receive a message from the peer socket endpoint. These functions have the following prototypes:

```
int send( int sock, const void *msg, int len, unsigned int flags );
int recv( int sock, void *buf, int len, unsigned int flags );
```

The send function takes as its first argument the socket descriptor from which to send the msg. The msg is defined as a (const void *) because the object referenced by msg is not altered by the send function. The number of bytes to be sent in msg is contained by the len argument. Finally, a flags argument can alter the behavior of the send call. An example of sending a string through a previously created stream socket is shown as follows:

```
strcpy( buf, "Hello\n");
send( sock, (void *)buf, strlen(buf), 0);
```

In this example, your character array is initialized by the strcpy function. This buffer is then sent through sock to the peer endpoint, with a length defined by the string length function, strlen. To see flags use, take a look at one side effect of the send call. When send is called, it can block until all of the data contained within buf has been placed on the socket's send queue. If not enough space is available to do this, the send function blocks until space is available. If you want to avoid this blocking behavior and instead want the send call to simply return if sufficient space is available, you can set the MSG_DONTWAIT flag, such as follows:

```
send( sock, (void *)buf, strlen(buf), MSG_DONTWAIT);
```

The return value from send represents either an error (less than 0) or the number of bytes that were queued to be sent. Completion of the send function does not imply that the data was actually transmitted to the host, only that it is queued on the socket's send queue waiting to be transferred.

The recv function mirrors the send function in terms of an argument list. Instead of sending the data pointed to be msg, the recv function fills the buf argument with the bytes read from the socket. You must define the size of the buffer so that the network protocol stack doesn't overwrite the buffer, which is defined by the len argument. Finally, you can alter the behavior of the read call using the flags argument. The value returned by the recv function is the number of bytes now contained in the msg buffer, or -1 on error. An example of the recv function is as follows:

```
#define MAX_BUFFER_SIZE        50
char buffer[MAX_BUFFER_SIZE+1];
...
numBytes = recv( sock, buffer, MAX_BUFFER_SIZE, 0);
```

At completion of this example, numBytes contains the number of bytes that are contained within the buffer argument.

You can peek at the data that's available to read by using the `MSG_PEEK` flag. This performs a read, but it doesn't consume the data at the socket. This requires another `recv` to actually consume the available data. An example of this type of read is illustrated as follows:

```
numBytes = recv( sock, buffer, MAX_BUFFER_SIZE, MSG_PEEK);
```

This call requires an extra copy (the first to peek at the data, and the second to actually read and consume it). More often than not, this behavior is handled instead at the application layer by actually reading the data and then determining what action to take.

Unconnected Socket Functions

The `sendto` and `recvfrom` functions are used to send a message to the peer socket endpoint and receive a message from the peer socket endpoint. These functions have the following prototypes:

```
int sendto( int sock, const void *msg, int len,
                      unsigned int flags,
                      const struct sockaddr *to, int tolen );
    int recvfrom( int sock, void *buf, int len,
                        unsigned int flags,
                        struct sockaddr *from, int *fromlen );
```

The `sendto` function is used by an unconnected socket to send a datagram to a destination defined by an initialized address structure. The `sendto` function is similar to the previously discussed `send` function, except that the recipient is defined by the `to` structure. An example of the `sendto` function is shown in the following code:

```
struct sockaddr_in destaddr;
int sock;
char *buf;
...
memset( &destaddr, 0, sizeof(destaddr) );
destaddr.sin_family = AF_INET;
destaddr.sin_port = htons(581);
destaddr.sin_addr.s_addr = inet_addr("192.168.1.1");
sendto( sock, buf, strlen(buf), 0,
              (struct sockaddr *)&destaddr, sizeof(destaddr) );
```

In this example, the datagram (contained with `buf`) is sent to an application on host `192.168.1.1`, port number `581`. The `destaddr` structure defines the intended recipient for the datagram.

As with the `send` function, the number of characters queued for transmission is returned, or `-1` if an error occurs.

The `recvfrom` function provides the ability for an unconnected socket to receive datagrams. The `recvfrom` function is again similar to the `recv` function, but an address structure and length are provided. The address structure is used to return the sender of the datagram to the function caller. This information can be used with the `sendto` function to return a response datagram to the original sender.

An example of the `recvfrom` function is shown in the following code:

```
#define MAX_LEN    100
struct sockaddr_in fromaddr;
int sock, len, fromlen;
char buf[MAX_LEN+1];
...
fromlen = sizeof(fromaddr);
len = recvfrom( sock, buf, MAX_LEN, 0,
                (struct sockaddr *)&fromaddr, &fromlen );
```

This blocking call returns when either an error occurs (represented by a `-1` return) or a datagram is received (return value of `0` or greater). The datagram is contained within `buf` and has a length of `len`. The `fromaddr` contains the datagram sender, specifically the host address and port number of the originating application.

Socket Options

Socket options permit an application to change some of the modifiable behaviors of sockets and the functions that manipulate them. For example, an application can modify the sizes of the send or receive socket buffers or the size of the maximum segment used by the TCP layer for a given socket.

The functions for setting or retrieving options for a given socket are provided by the following function prototypes:

```
int getsockopt( int sock, int level, int optname,
                          void *optval, socklen_t *optlen );
    int setsockopt( int sock, int level, int optname,
                          const void *optval, socklen_t optlen );
```

First, you define the socket of interest using the `sock` argument. Next, you must define the level of the socket option that is being applied. The `level` argument can be `SOL_SOCKET` for socket-layer options, `IPPROTO_IP` for IP layer options, and

IPPROTO_TCP for TCP layer options. The specific option within the level is applied using the optname argument. Arguments optval and optlen define the specifics of the value of the option. optval is used to get or set the option value, and optlen defines the length of the option. This slightly complicated structure is used because structures can be used to define options.

Now take a look at an example for both setting and retrieving an option. In the first example, you retrieve the size of the send buffer for a socket.

```
int sock, size, len;
...
getsockopt( sock, SOL_SOCKET, SO_SNDBUF, (void *)&size,
(socklen_t *)&len );
printf( "Send buffer size is &d\n", size );
```

Now take a look at a slightly more complicated example. In this case, you're going to set the linger option. Socket linger allows you to change the behavior of a stream socket when the socket is closed and data is remaining to be sent. After close is called, any data remaining attempts to be sent for some amount of time. If after some duration the data cannot be sent, then the data to be sent is abandoned. The time after the close when the data is removed from the send queue is defined as the *linger time*. This can be set using a special structure called linger, as shown in the following example:

```
struct linger ling;
int sock;
...
ling.l_onoff = 1; /* Enable */
ling.l_linger = 10; /* 10 seconds */
setsockopt( sock, SOL_SOCKET, SO_LINGER,
    (void *)&ling, sizeof(struct linger) );
```

After this call is performed, the socket waits 10 seconds after the socket close before aborting the send.

Other Miscellaneous Functions

Now it's time to look at a few miscellaneous functions from the Sockets API and the capabilities they provide. The three function prototypes discussed in this section are shown in the following code:

```
struct hostent *gethostbyname( const char *name );
int getsockname( int sock, struct sockaddr *name, socklen_t
*namelen );
int getpeername( int sock, struct sockaddr *name, socklen_t
*namelen );
```

Function `gethostbyname` provides the means to resolve a host and domain name (otherwise known as a fully qualified domain name, or FQDN) to an IP address. For example, the FQDN of www.microsoft.com might resolve to the IP address `207.46.249.27`. Converting an FQDN to an IP address is important because all of the Sockets API functions work with number IP addresses (32-bit addresses) rather than FQDNs. An example of the `gethostbyname` function is shown next:

```
struct hostent *hptr;
hptr = gethostbyname( "www.microsoft.com");
if (hptr == NULL) // can't resolve...
else {
   printf( "Binary address is %x\n", hptr-> h_addr_list[0] );
}
```

Function `gethostbyname` returns a pointer to a structure that represents the numeric IP address for the FQDN (`hptr->h_addr_list[0]`). Otherwise, `gethostbyname` returns a `NULL`, which means that the FQDN could not be resolved by the local resolver. This call blocks while the local resolver communicates with the configured DNS servers.

Function `getsockname` permits an application to retrieve information about the local socket endpoint. This function, for example, can identify the dynamically assigned ephemeral port number for the local socket. An example of its use is shown in the following code:

```
int sock;
struct sockaddr localaddr;
int laddrlen;
// Socket for sock created and connected.
...
getsockname( sock, (struct sockaddr_in *)&localaddr, &laddrlen );
printf( "local port is %d\n", ntohs(localaddr.sin_port) );
```

The reciprocal function of `getsockname` is `getpeername`. This permits you to gather addressing information about the connected peer socket. An example, similar to the `getsockname` example, is shown in the following code:

```
int sock;
struct sockaddr remaddr;
int raddrlen;
// Socket for sock created and connected.
...
getpeername( sock, (struct sockaddr_in *)&remaddr, &raddrlen );
printf( "remote port is %d\n", ntohs(remaddr.sin_port) );
```

In both examples, the address can also be extracted using the `sin_addr` field of the `sockaddr` structure.

Other Transport Protocols

While TCP and UDP are by far the most popular transport layer protocols, others are useful and might be the next big thing in the transport layer. The most important is called the Stream Control Transmission Protocol (SCTP). SCTP is a new protocol, but instead of reinventing the wheel, SCTP instead combines the best features of the TCP and UDP protocols. This section explores the SCTP protocol and the advantages that it brings.

Features of SCTP

Recall that the advantage of TCP is that it's a reliable protocol that guarantees ordered delivery of data while managing congestion in a network. UDP, on the other hand, is a message-oriented protocol that guarantees framing of data (what's sent is what is received in terms of segment sizes), but guarantees no ordered delivery nor manages congestion in the network.

SCTP combines the reliability of TCP with the message-framing of UDP. SCTP also adds a number of additional features such as multi-homing and multi-streaming. It also includes a number of other features that are explored in the sections that follow.

Multi-homing

One of the two most novel features of SCTP is called multi-homing. Multi-homing allows for higher availability through connection failure. Recall that connections exist between two network interfaces and are static. SCTP's multi-homing feature provides for associations not between network interfaces but between hosts (containing multiple interfaces). This allows a connection to migrate from one network interface to another on the same host. A common architecture is for each network interface to traverse different networks so that if a network or interface fails, the other can be used for connection migration (see Figure 13.5).

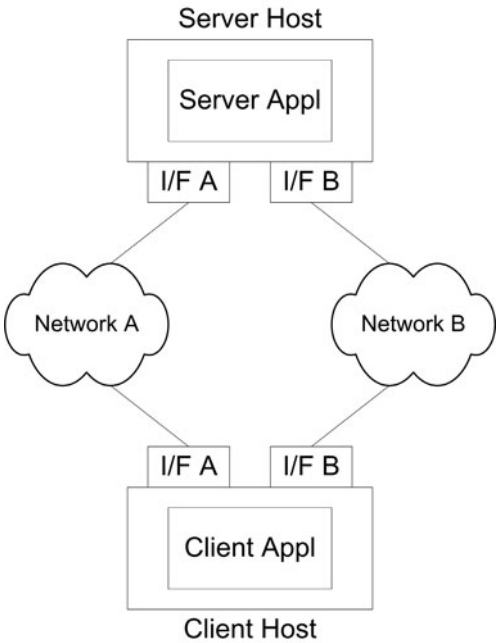

FIGURE 13.5 Multi-homing uses multiple interfaces.

Multi-Streaming

Multi-streaming is the other major novel feature of SCTP. Recall that TCP sockets are commonly referred to as stream sockets because data is treated as a stream of octets. TCP also includes urgent data, which is treated separately from the normal TCP stream and is used to communicate high-priority events. Instead of urgent data, SCTP includes the concept of multiple streams within a single SCTP socket. Each stream within the socket is independently addressable, allowing independent streams of data to be communicated between the two peers. This can be very useful in applications like FTP, where two sockets are normally created (one for commands and the other for high-speed data).

Figure 13.6 provides a simple illustration of this concept. Within a single connection (or in SCTP, where it's known as an association), two different streams are created. This allows the streams to operate independently of one another. For example, one connection can be used for occasional commands and the other used for data.

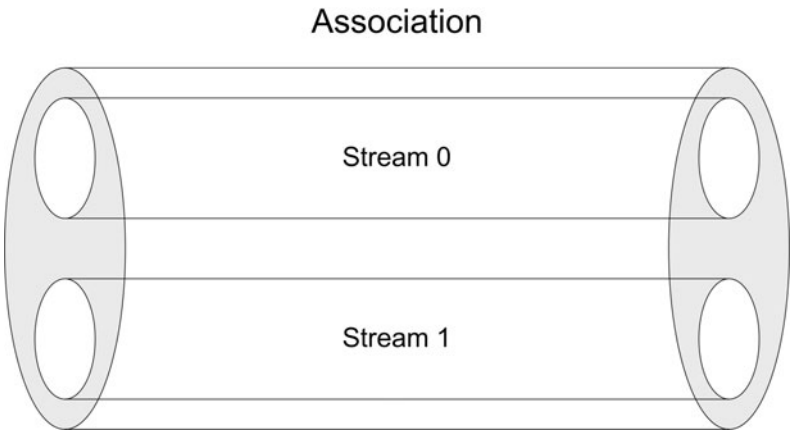

FIGURE 13.6 Multi-streaming in a single SCTP association.

Other SCTP Features

SCTP also includes a number of other features. One other important feature is implementing message framing (as is done with UDP). This allows SCTP to implement byte-streams like TCP and message framing like UDP. Framing simply means that when the writer puts 3 bytes into the socket followed by another 3 bytes, the peer reads 3 bytes and on a subsequent call another 3 bytes. Recall that in a TCP stream, 6 bytes would likely be read at the peer.

Finally, SCTP allows for unordered delivery of data. Data in UDP is delivered in order, but in SCTP this behavior can be configured.

SCTP provides other features such as connection initiation protection (to protect against denial-of-service attacks) and graceful shutdown (to remove TCP's half-close state). With such a great set of useful features, SCTP is worth a look.

Multilanguage Perspectives

This chapter has focused on the Sockets API from the perspective of the C language, but the Sockets API is available for any worthwhile language.

Consider first the Ruby language. Ruby is an object-oriented scripting language that is growing in popularity. It's simple and clean and useful in many domains. One domain that demonstrates the simplicity of the language is in network application development.

The Daytime protocol server is shown in Listing 13.3. Ruby provides numerous classes for networking development; the one illustrated here supports TCP server sockets (`TCPserver`). At line 4, you create your server socket and bind it to the Daytime protocol server (identified by the string `"daytime"`). At line 12, you await an incoming connection using the `accept` method. When one arrives, you emit the current time to the client at line 19 using the `write` method. Finally, the socket is closed at line 23 using the `close` method.

LISTING 13.3 Daytime Protocol Server in the Ruby Language (on the CD-ROM at `./source/ch13/dayserv.rb`)

```
1:      require 'Socket'
2:
3:      # Create a new TCP Server using port 13
4:      servsock = TCPserver::new("daytime")
5:
6:      # Debug data – emit the server socket info
7:      print("server address : ", servsock.addr::join(":"),"\n")
8:
9:      while true
10:
11:      # Await a connection from a client socket
12:      clisock = servsock::accept
13:
14:      # Emit some debugging data on the peer
15:      print("accepted ", clisock.peeraddr::join(":"),"\n")
16:      print(clisock, " is accepted\n")
17:
18:      # Emit the time through the socket to the client
19:      clisock.write( Time::new )
20:      clisock.write("\n" )
21:
22:      # Close the client connection
23:      clisock.close
24:
25:     end
```

The Sockets API is also useful in other types of languages, such as functional ones. The scheme language is Lisp-like in syntax, but it easily integrates the functionality of the Sockets API.

Listing 13.4 illustrates the Daytime protocol client in the scheme language. Lines 2 and 3 define two global constants used in the client. At line 5, you create the `stream-client` procedure. You create the socket at lines 6 and 7 of the stream type

using the `socket-connect` procedure. You provide the previously defined host and port values to identify to whom you should connect. This is bound to the `sock` variable using the `let` expression. Having a connected socket, you read from the socket at line 8 using another `let` expression. The return value of `read-string` is bound to result, which is then printed at line 9 using `write-string`. You emit a newline at line 10 and then close the socket using the `close-socket` procedure at line 11. The client is started at line 16 by calling the defined procedure `stream-client`.

LISTING 13.4 Daytime Protocol Client in the Scheme Language (on the CD-ROM at `./source/ch13/daycli.scm`)

```
1:      ; Define a couple of server constants
2:      (define host "localhost")
3:      (define port 13)
4:
5:      (define (stream-client)
6:       (let ((sock (socket-connect protocol-family/internet
7:             socket-type/stream host port)))
8:        (let ((result (read-string 100 (socket:inport sock))))
9:         (write-string result)
10:        (newline)
11:        (close-socket sock) ) ) )
12:
13:     ;
14:     ; Invoke the stream client
15:     ;
16:     (stream-client)
```

Space permitting, you could explore Sockets applications in a multitude of other applications (such as Python, Perl, C++, Java, and Tcl) [Jones03]. The key is that sockets aren't just a C language construct, but are useful in many languages.

Summary

This chapter provided a quick tour of Sockets programming in C. It investigated the Sockets programming paradigm, covering the basic elements of networking such as hosts, interfaces, protocols, and ports. The Sockets API was explored in a sample server and client in C and then in detail looking at the functions of the API. In addition to the typical TCP and UDP sockets, SCTP was also explored. Finally, the use of the Sockets API was discussed from a multilanguage perspective, illustrating its applicability to non-C language scenarios.

Sockets Programming APIs

```
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
int socket( int domain, int type, int protocol );
int bind( int sock, struct sockaddr *addr, int addrLen );
int listen( int sock, int backlog );
int accept( int sock, struct sockaddr *addr, int *addrLen );
int connect( int sock, (struct sockaddr *)servaddr, int addrLen );
int send( int sock, const void *msg, int len, unsigned int flags );
int recv( int sock, void *buf, int len, unsigned int flags );
int sendto( int sock, const void *msg, int len,

 unsigned int flags,
 const struct sockaddr *to, int tolen );
int recvfrom( int sock, void *buf, int len,
 unsigned int flags,
 struct sockaddr *from, int *fromlen );
int getsockopt( int sock, int level, int optname,
  void *optval, socklen_t *optlen );
int setsockopt( int sock, int level, int optname,
  const void *optval, socklen_t optlen );
int close( int sock );

struct sockaddr_in {
    int16_t sin_family;
        uint16_t sin_port;
        struct in_addr sin_addr;
        char sin_zero[8];
};

struct in_addr {
        uint32_t s_addr;
};

#include <netdb.h>
struct hostent *gethostbyname( const char *name );
int getsockname( int sock, struct sockaddr *name,
               socklen_t *namelen );
int getpeername( int sock, struct sockaddr *name,
               socklen_t *namelen );
struct hostent {
    char *h_name;
```

```
    char **h_aliases;
    int h_addrtype;
    int h_length;
    char **h_addr_list;
}
#define h_addr h_addr_list[0]
```

References

[Jones03] *BSD Sockets Programming from a Multilanguage Perspective*, M. Tim Jones, Charles River Media, 2003.

Resources

The Ruby Language at http://www.ruby-lang.org/en/.

scsh—The Scheme Shell at http://www.scsh.net.

Stevens, W. Richard, *Unix Network Programming—Networking APIs: Sockets and XTI Volume 1*, Prentice Hall PTR, 1998.

Stewart, Randall R., and Xie, Qiaobing, *Stream Control Transmission Protocol (SCTP): A Reference Guide*, Addison-Wesley Professional, 2002.

This page intentionally left blank

14 GNU/Linux Process Model

In This Chapter

- Creating Processes with `fork()`
- Review of Process-Related API Functions
- Raising and Catching Signals
- Available Signals and Their Uses
- GNU/Linux Process-Related Commands

Introduction

This chapter introduces the GNU/Linux process model. It defines elements of a process, how processes communicate with each other, and how to control and monitor them. First, the chapter addresses a quick review of fundamental APIs and then follows up with a more detailed review, complete with sample applications that illustrate each technique.

GNU/Linux Processes

GNU/Linux presents two fundamental types of processes. These are *kernel threads* and *user processes*. The focus here is on user processes (those created by `fork` and `clone`). Kernel threads are created within the kernel context via the `kernel_thread()` function.

When a subprocess is created (via `fork`), a new child task is created with a copy of the memory used by the original parent task. This memory is separate between the two processes. Any variables present when the `fork` takes place are available to the child. But after the `fork` completes, any changes that the parent makes to a variable are not seen by the child. This is important to consider when using the `fork` API function.

When a new task is created, the memory space used by the parent isn't actually copied to the child. Instead, both the parent and child reference the same memory space, with the memory pages marked as copy-on-write. When any of the processes attempt to write to the memory, a new set of memory pages is created for the process that is private to it alone. In this way, creating a new process is an efficient mechanism, with copying of the memory space deferred until writes take place. In the default case, the child process inherits open file descriptors, the memory image, and CPU state (such as the PC and assorted registers).

Certain elements are not copied from the parent and instead are created specifically for the child. The following sections take a look at examples of these. What's important to understand at this stage is that a process can create subprocesses (known as *children*) and generally control them.

Whirlwind Tour of Process APIs

As defined previously, you can create a new process with the `fork` or `clone` API function. But in fact, you create a new process every time you execute a command or start a program. Consider the simple program shown in Listing 14.1.

LISTING 14.1 First Process Example (on the CD-ROM at `./source/ch14/process.c`)

```
1:      #include <stdio.h>
2:      #include <unistd.h>
3:      #include <sys/types.h>
4:
5:      int main()
6:      {
7:        pid_t myPid;
8:        pid_t myParentPid;
9:        gid_t myGid;
10:       uid_t myUid;
11:
12:       myPid = getpid();
```

```
13:         myParentPid = getppid();
14:         myGid = getgid();
15:         myUid = getuid();
16:
17:         printf( "my process id is %d\n", myPid );
18:
19:         printf( "my parent's process id is %d\n", myParentPid );
20:
21:         printf( "my group id is %d\n", myGid );
22:
23:         printf( "my user id is %d\n", myUid );
24:
25:         return 0;
26:       }
```

Every process in GNU/Linux has a unique identifier called a process ID (or `pid`). Every process also has a parent (except for the `init` process). In Listing 14.1, you use the `getpid()` function to get the current process ID and the `getppid()` function to retrieve the process's parent ID. Then you grab the group ID and the user ID using `getuid()` and `getgid()`.

If you were to compile and then execute this application, you would see the following:

```
$ ./process
my process id is 10932
my parent's process id is 10795
my group id is 500
my user id is 500
$
```

You see the process ID is 10932, and the parent is 10795 (our bash shell). If you execute the application again, you see the following:

```
$ ./process
my process id is 10933
my parent's process id is 10795
my group id is 500
my user id is 500
$
```

Note that your process ID has changed, but all other values have remained the same. This is expected, because the only thing you've done is create a new process that performs its I/O and then exits. Each time a new process is created, a new process ID is allocated to it.

Creating a Subprocess with `fork`

Now it's time to move on to the real topic of this chapter, creating new processes within a given process. The `fork` API function is the most common method to achieve this.

The `fork` call is an oddity when you consider what is actually occurring. When the `fork` API function returns, the split occurs, and the return value from `fork` identifies in which context the process is running. Consider the following code snippet:

```
pid_t pid;
...
pid = fork();
if (pid > 0) {
  /* Parent context, child is pid */
} else if (pid == 0) {
  /* Child context */
} else {
  /* Parent context, error occurred, no child created */
}
```

You see here three possibilities from the return of the `fork` call. When the return value of `fork` is greater than zero, then you're in the parent context and the value represents the process ID of the child. When the return value is zero, then you're in the child process's context. Finally, any other value (less than zero) represents an error and is performed within the context of the parent.

Now it's time to look at a sample application of `fork` (shown in Listing 14.2). This working example illustrates the `fork` call, identifying the contexts. At line 11, you call `fork` to split your process into parent and child. Both the parent and child emit some text to standard-out so you can see each execution. Note that a shared variable (`role`) is updated by both parent and child and emitted at line 45.

LISTING 14.2 Working Example of the `fork` Call (on the CD-ROM at `./source/ch14/smplfork.c`)

```
1:      #include <sys/types.h>
2:      #include <unistd.h>
3:      #include <errno.h>
4:
5:      int main()
6:      {
7:        pid_t ret;
8:        int   status, i;
```

```
9:          int   role = -1;
10:
11:         ret = fork();
12:
13:         if (ret > 0) {
14:
15:           printf("Parent: This is the parent process (pid %d)\n",
16:                   getpid());
17:
18:           for (i = 0 ; i < 10 ; i++) {
19:             printf("Parent: At count %d\n", i);
20:             sleep(1);
21:           }
22:
23:           ret = wait( &status );
24:
25:           role = 0;
26:
27:         } else if (ret == 0) {
28:
29:           printf("Child: This is the child process (pid %d)\n",
30:                   getpid());
31:
32:           for (i = 0 ; i < 10 ; i++) {
33:             printf("Child: At count %d\n", i);
34:             sleep(1);
35:           }
36:
37:           role = 1;
38:
39:         } else {
40:
41:           printf("Parent: Error trying to fork() (%d)\n", errno);
42:
43:         }
44:
45:         printf("%s: Exiting...\n",
46:                 ((role == 0) ? "Parent" : "Child"));
47:
48:         return 0;
49:       }
```

The output of the application shown in Listing 14.2 is shown in the following snippet. You see that the child is started and in this case immediately emits some out-

put (its process ID and the first `count` line). The parent and the child then switch off from the GNU/Linux scheduler, each sleeping for one second and emitting a new `count`.

```
# ./smplfork
Child: This is the child process (pid 11024)
Child: At count 0
Parent: This is the parent process (pid 11023)
Parent: At count 0
Parent: At count 1
Child: At count 1
Parent: At count 2
Child: At count 2
Parent: At count 3
Child: At count 3
Parent: At count 4
Child: At count 4
Parent: At count 5
Child: At count 5
Child: Exiting...
Parent: Exiting...
#
```

At the end, you see the `role` variable used to emit the role of the process (parent or child). In this case, whereas the `role` variable was shared between the two processes, after the write occurs, the memory is split, and each process has its own variable, independent of the other. How this occurs is really unimportant. What's important to note is that each process has a copy of its own set of variables.

Synchronizing with the Creator Process

One element of Listing 14.2 was ignored, but this section now digs into it. At line 23, the `wait` function was called within the context of the parent. The `wait` function suspends the parent until the child exits. If the `wait` function is not called by the parent and the child exits, the child becomes what is known as a "zombie" process (neither alive nor dead). It can be problematic to have these processes lying around because of the resources that they waste, so handling child `exit` is necessary. Note that if the parent exits first, the children that have been spawned are inherited by the `init` process.

Another way to avoid zombie processes is to tell the parent to ignore child exit signals when they occur. This can be achieved using the `signal` *API function, which is explored in the next section, "Catching a Signal." In any case, after the child has stopped, any system resources that were used by the process are immediately released.*

The first two methods that this chapter discusses for synchronizing the exit of a child process are the `wait` and `waitpid` API functions. The `waitpid` API function provides greater control over the `wait` process; however, for now, this section looks exclusively at the `wait` API function.

The `wait` function suspends the caller (in this case, the parent) awaiting the `exit` of the child. After the child exits, the integer value reference (passed to `wait`) is filled in with the particular exit status. Sample use of the `wait` function, including parsing of the successful status code, is shown in the following code snippet:

```
int status;
pid_t pid;
...
pid = wait( &status );
if ( WIFEXITED(status) ) {
  printf( "Process %d exited normally\n", pid );
}
```

The `wait` function can set other potential status values, which are investigated in the "`wait`" section later in this chapter.

Catching a Signal

A signal is fundamentally an asynchronous callback for processes in GNU/Linux. You can register to receive a signal when an event occurs for a process or register to ignore signals when a default action exists. GNU/Linux supports a variety of signals, which are covered later in this chapter. Signals are an important topic here in process management because they allow processes to communicate with one another.

To catch a signal, you provide a signal handler for the process (a kind of callback function) and the signal that we're interested in for this particular callback. You can now look at an example of registering for a signal. In this example, you register for the `SIGINT` signal. This particular signal identifies that a Ctrl+C was received.

The main program in Listing 14.3 (lines 14–24) begins with registering your callback function (also known as the signal handler). You use the `signal` API function to register your handler (at line 17). You specify first the signal of interest and then the handler function that reacts to the signal. At line 21, you pause, which suspends the process until a signal is received.

The signal handler is shown at Listing 14.3 at lines 6–12. You simply emit a message to `stdout` and then flush it to ensure that it has been emitted. You return from your signal handler, which allows your main function to continue from the `pause` call and exit.

LISTING 14.3 Registering for Catching a Signal (on the CD-ROM at `./source/ch14/sigcatch.c`)

```
1:      #include <stdio.h>
2:      #include <sys/types.h>
3:      #include <signal.h>
4:      #include <unistd.h>
5:
6:      void catch_ctlc( int sig_num )
7:      {
8:        printf( "Caught Control-C\n" );
9:        fflush( stdout );
10:
11:       return;
12:     }
13:
14:     int main()
15:     {
16:
17:       signal( SIGINT, catch_ctlc );
18:
19:       printf("Go ahead, make my day.\n");
20:
21:       pause();
22:
23:       return 0;
24:     }
```

Raising a Signal

The previous example illustrated a process receiving a signal. You can also have a process send a signal to another process using the `kill` API function. The `kill` API function takes a process ID (to whom the signal is to be sent) and the signal to send.

Take a look at a simple example of two processes communicating via a signal. This example uses the classic parent/child process creation via `fork` (see Listing 14.4).

At lines 8–13, you declare your signal handler. This handler is very simple, as shown, and simply emits some text to `stdout` indicating that the signal was received, in addition to the process context (identified by the process ID).

The `main` (lines 15–61) is a simple parent/child `fork` example. The parent context (starting at line 25) installs the signal handler and then pauses (awaiting the receipt of a signal). It then continues by awaiting the exit of the child process.

The child context (starting at line 39) sleeps for one second (allowing the parent context to execute and install its signal handler) and then raises a signal. Note that you use the `kill` API function (line 47) to direct the signal to the parent process ID (via `getppid`). The signal you use is `SIGUSR1`, which is a user-definable signal. After the signal has been raised, the child sleeps another two seconds and then exits.

LISTING 14.4 Raising a Signal from a Child to a Parent Process (on the CD-ROM at `./source/ch14/raise.c`)

```
1:      #include <stdio.h>
2:      #include <sys/types.h>
3:      #include <sys/wait.h>
4:      #include <unistd.h>
5:      #include <signal.h>
6:      #include <errno.h>
7:
8:      void usr1_handler( int sig_num )
9:      {
10:
11:       printf( "Parent (%d) got the SIGUSR1\n", getpid() );
12:
13:     }
14:
15:     int main()
16:     {
17:       pid_t ret;
18:       int   status;
19:       int   role = -1;
20:
21:       ret = fork();
22:
23:       if (ret > 0) {                /* Parent Context */
24:
25:         printf( "Parent: This is the parent process (pid %d)\n",
26:                  getpid() );
27:
28:         signal( SIGUSR1, usr1_handler );
29:
30:         role = 0;
31:
32:         pause();
33:
34:         printf( "Parent: Awaiting child exit\n" );
```

```
35:           ret = wait( &status );
36:
37:         } else if (ret == 0) {        /* Child Context */
38:
39:           printf( "Child: This is the child process (pid %d)\n",
40:                    getpid() );
41:
42:           role = 1;
43:
44:           sleep( 1 );
45:
46:           printf( "Child: Sending SIGUSR1 to pid %d\n",
              getppid() );
47:           kill( getppid(), SIGUSR1 );
48:
49:           sleep( 2 );
50:
51:         } else {                    /* Parent Context – Error */
52:
53:           printf( "Parent: Error trying to fork() (%d)\n",
              errno );
54:
55:         }
56:
57:         printf( "%s: Exiting...\n",
58:                  ((role == 0) ? "Parent" : "Child") );
59:
60:         return 0;
61:       }
```

While this example is probably self-explanatory, looking at its output can be beneficial to understanding exactly what's going on. The output for the application shown in Listing 14.4 is as follows:

```
$ ./raise
Child: This is the child process (pid 14960)
Parent: This is the parent process (pid 14959)
Child: Sending SIGUSR1 to pid 14959
Parent (14959) got the SIGUSR1
Parent: Awaiting child exit
Child: Exiting...
Parent: Exiting...
$
```

You can see that the child performs its first `printf` first (the `fork` gave control of the CPU to the child first). The child then sleeps, allowing the parent to perform its first `printf`, install the signal handler, and then pause awaiting a signal. Now that the parent has suspended, the child can then execute again (after the one-second sleep has finished). It emits its message, indicating that the signal is being raised, and then raises the signal using the `kill` API function. The parent then performs the `printf` within the signal handler (in the context of the parent process as shown by the process ID) and then suspends again awaiting child exit via the `wait` API function. The child process can then execute again, and after the two-second sleep has finished, it exits, releasing the parent from the `wait` call so that it, too, can exit.

It's fairly simple to understand, but it's a powerful mechanism for coordination and synchronization between processes. The entire thread is shown graphically in Figure 14.1. This illustrates the coordination points that exist within your application (shown as dashed horizontal lines from the child to the parent).

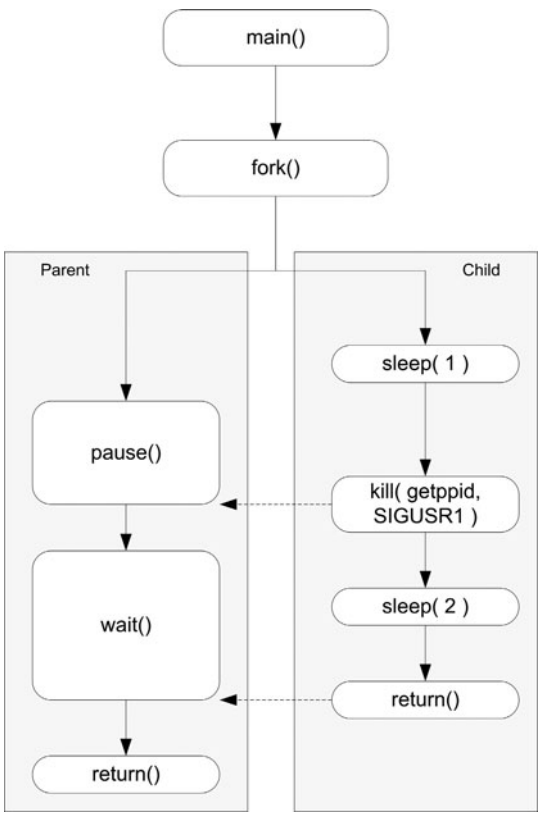

FIGURE 14.1 Graphical illustration of Listing 14.4.

If you're raising a signal to yourself (the same process), you can also use the `raise` *API function. This takes the signal to be raised but no process ID argument (because it's automatically* `getpid`*).*

Traditional Process API

Now that you've looked at a number of different API functions that relate to the GNU/Linux process model, you can now dig further into these functions (and others) and explore them in greater detail. Table 14.1 provides a list of the functions that are explored in the remainder of this section, including their uses.

TABLE 14.1 Traditional Process and Related APIs

API Function	Use
`fork`	Create a new child process.
`wait`	Suspend execution until a child process exits.
`waitpid`	Suspend execution until a specific child process exits.
`signal`	Install a new signal handler.
`pause`	Suspend execution until a signal is caught.
`kill`	Raise a signal to a specified process.
`raise`	Raise a signal to the current process.
`exec`	Replace the current process image with a new process image.
`exit`	Cause normal program termination of the current process.

The remainder of this chapter addresses each of these functions in detail, illustrated in sample applications.

`fork`

The `fork` API function provides the means to create a new child subprocess from an existing parent process. The new child process is identical to the parent process in almost every way. Some differences include the process ID (a new ID for the child) and that the parent process ID is set to the parent. File locks and signals that are pending to the parent are not inherited by the child process. The prototype for the `fork` function is defined as follows:

```
pid_t fork( void );
```

The fork API function takes no arguments and returns a pid (process identifier). The fork call has a unique structure in that the return value identifies the context in which the process is running. If the return value is zero, then the current process is the newly created child process. If the return value is greater than zero, then the current process is the parent, and the return value represents the process ID of the child. This is illustrated in the following snippet:

```
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
...
pid_t ret;
ret = fork();
if        ( ret > 0 ) {
  /* Parent Process */
  printf( "My pid is %d and my child's is %d\n",
               getpid(), ret );
} else if ( ret == 0 ) {
  /* Child Process */
  printf( "My pid is %d and my parent's is %d\n",
               getpid(), getppid() );
} else {
  /* Parent Process – error */
  printf( "An error occurred in the fork (%d)\n", errno );
}
```

Within the fork() call, the process is duplicated, and then control is returned to the unique process (parent and child). If the return value of fork is less than zero, then an error has occurred. The errno value represents either EAGAIN or ENOMEM. Both errors arise from a lack of available memory.

The fork API function is very efficient in GNU/Linux because of its unique implementation. Rather than copy the page tables for the memory when the fork takes place, the parent and child share the same page tables but are not permitted to write to them. When a write takes place to one of the shared page tables, the page table is copied for the writing process so that it has its own copy. This is called copy-on-write in GNU/Linux and permits the fork to take place very quickly. Only as writes occur to the shared data memory does the segregation of the page tables take place.

wait

The purpose of the wait API function is to suspend the calling process until a child process (created by this process) exits or until a signal is delivered. If the parent isn't

currently waiting on the child to exit, the child exits, and the child process becomes a zombie process.

The `wait` function provides an asynchronous mechanism as well. If the child process exits before the parent has had a chance to call `wait`, then the child becomes a zombie. However, it is then freed after `wait` is called. The `wait` function, in this case, returns immediately.

The prototype for the `wait` function is defined as follows:

```
pid_t wait( int *status );
```

The `wait` function returns the `pid` value of the child that exited, or –1 if an error occurred. The `status` variable (whose reference is passed into `wait` as its only argument) returns status information about the child exit. This variable can be evaluated using a number of macros. These macros are listed in Table 14.2.

TABLE 14.2 Macro Functions to Evaluate `wait` Status

Macro	Description
`WIFEXITED`	Nonzero if the child exited normally
`WEXITSTATUS`	Returns the `exit` status of the child
`WIFSIGNALED`	Returns `true` if child exited because of a signal that wasn't caught by the child
`WTERMSIG`	Returns the signal number that caused the child to exit (relevant only if `WIFSIGNALED` is `true`)

The general form of the status evaluation macro is demonstrated in the following code snippet:

```
pid = wait( &status );
if        ( WIFEXITED(status) ) {
  printf( "Child exited normally with status %d\n",
               WEXITSTATUS(status) );
} else if ( WIFSIGNALED(status) ) {
  printf( "Child exited by signal with status %d\n",
               WTERMSIG(status) );
}
```

In some cases, you're not interested in the `exit` status of your child processes. In the `signal` API function discussion, you can see a way to ignore this status so that `wait` does not need to be called by the parent to avoid child zombie processes.

waitpid

Whereas the wait API function suspends the parent until a child exits (any child), the waitpid API function suspends until a specific child exits. The waitpid function provides some other capabilities, which are explored here. The waitpid function prototype is defined as follows:

```
pid_t waitpid( pid_t pid, int *status, int options );
```

The return value for waitpid is the process identifier for the child that exited. The return value can also be zero if the options argument is set to WNOHANG and no child process has exited (returns immediately).

The arguments to waitpid are a pid value, a reference to a return status, and a set of options. The pid value can be a child process ID or other values that provide different behaviors. Table 14.3 lists the possible pid values for waitpid.

TABLE 14.3 pid Arguments for waitpid

Value	Description
> 0	Suspend until the child identified by the pid value has exited
0	Suspend until any child exits whose group ID matches that of the calling process
−1	Suspend until any child exits (identical to the wait function)
< −1	Suspend until any child exits whose group ID is equal to the absolute value of the pid argument

The status argument for waitpid is identical to the wait function, except that two new status macros are possible (see Table 14.4). These macros are seen only if the WUNTRACED option is specified.

TABLE 14.4 Extended Macro Functions for waitpid

Macro	Description
WIFSTOPPED	Returns true if the child process is currently stopped
WSTOPSIG	Returns the signal that caused the child to stop (relevant only if WIFSTOPPED was nonzero)

The final argument to waitpid is the options argument. Two options are available: WNOHANG and WUNTRACED. WNOHANG, as discussed, avoids suspension of the parent

process and returns only if a child has exited. The WUNTRACED option returns for children that have been stopped and not yet reported.

Now it's time to take a look at some examples of the waitpid function. In the first code snippet, you fork off a new child process and then await it explicitly (rather than as with the wait method that waits for any child).

```
pid_t child_pid, ret;
int status;
...
child_pid = fork();
if (child_pid == 0) {
  // Child process...
} else if (child_pid > 0) {
  ret = waitpid( child_pid, &status, 0 );
  /* Note ret should equal child_pid on success */
  if ( WIFEXITED(status) ) {
    printf( "Child exited normally with status %d\n",
                WEXITSTATUS(status) );
  }
}
```

In this example, you fork off your child and then use waitpid with the child's process ID. Note here that you can use the status macro functions that were defined with wait (as demonstrated with WIFEXITED). If you don't want to wait for the child, you can specify WNOHANG as an option. This requires you to call waitpid periodically to handle the child exit:

```
ret = waitpid( child_pid, &status, WNOHANG );
```

The following line awaits a child process exiting the defined group. Note that you negate the group ID in the call to waitpid. Also notable is passing NULL as the status reference. In this case, you're not interested in getting the child's exit status. In any case, the return value is the process ID for the child process that exited.

```
pid_t group_id;
...
ret = waitpid( -group_id, NULL, 0 );
```

signal

The signal API function allows you to install a signal handler for a process. The signal handler passed to the signal API function has the following form:

```
void signal_handler( int signal_number );
```

After it is installed, the function is called for the process when the particular signal is raised to the process. The prototype for the `signal` API function is defined as follows:

```
sighandler_t signal( int signum, sighandler_t handler );
```

where the `sighandler_t typedef` is as follows:

```
typedef void (*sighandler_t)(int);
```

The `signal` function returns the previous signal handler that was installed, which allows the new handler to chain the older handlers together (if necessary).

A process can install handlers to catch signals, and it can also define that signals should be ignored (`SIG_IGN`). To ignore a signal for a process, the following code snippet can be used:

```
signal( SIGCHLD, SIG_IGN );
```

After this particular code is executed, it is not necessary for a parent process to wait for the child to exit using `wait` or `waitpid`.

Signal handlers for a process can be of three different types. They can be ignored (via `SIG_IGN`), the default handler for the particular signal type (`SIG_DFL`), or a user-defined handler (installed via `signal`).

A large number of signals exist for GNU/Linux. They are provided in Tables 14.5–14.8 with their meanings. The signals are split into four groups, based upon default action for the signal.

TABLE 14.5 GNU/Linux Signals That Default to Terminate

Signal	Description
SIGHUP	Hang up—commonly used to restart a task
SIGINT	Interrupt from the keyboard
SIGKILL	Kill signal
SIGUSR1	User-defined signal
SIGUSR2	User-defined signal
SIGPIPE	Broken pipe (no reader for write)
SIGALRM	Timer signal (from API function `alarm`)
SIGTERM	Termination signal
SIGPROF	Profiling timer expired

TABLE 14.6 GNU/Linux Signals That Default to Ignore

Signal	Description
SIGCHLD	Child stopped or terminated
SIGCLD	Same as SIGCHLD
SIGURG	Urgent data on a socket

TABLE 14.7 GNU/Linux Signals That Default to Stop

Signal	Description
SIGSTOP	Stop process
SIGTSTP	Stop initiated from TTY
SIGTTIN	Background process has TTY input
SIGTTOU	Background process has TTY output

TABLE 14.8 GNU/Linux Signals That Default to Core Dump

Signal	Description
SIGQUIT	quit signal from keyboard
SIGILL	Illegal instruction encountered
SIGTRAP	Trace or breakpoint trap
SIGABRT	Abort signal (from API function abort)
SIGIOT	IOT trap, same as SIGABRT
SIGBUS	Bus error (invalid memory access)
SIGFPE	Floating-point exception
SIGSEGV	Segment violation (invalid memory access)

The first group (terminate) lists the signals whose default action is to terminate the process. The second group (ignore) lists the signals for which the default action is to ignore the signal. The third group (stop) stops the process (suspends rather than terminates). Finally, the fourth group (core) lists those signals whose action is to both terminate the process and perform a core dump (generate a core dump file).

It's important to note that the SIGSTOP and SIGKILL signals cannot be ignored or caught by the application. One other signal not categorized in the preceding information is the SIGCONT signal, which is used to continue a process if it was previously stopped.

GNU/Linux also supports 32 real-time signals (of POSIX 1003.1-2001). The signals are numbered from 32 (SIGRTMIN) up to 63 (SIGRTMAX) and can be sent using the sigqueue API function. The receiving process must use sigaction to install the signal handler (discussed later in this chapter) to collect other data provided in this signaling mechanism.

Now it's time to look at a simple application that installs a signal handler at the parent, which is inherited by the child (see Listing 14.5). In this listing, you first declare a signal handler (lines 8–13) that is installed by the parent prior to the fork (at line 21). Installing the handler prior to the fork means that the child inherits this signal handler as well.

After the fork (at line 23), the parent and child context emit an identification string to stdout and then call the pause API function (which suspends each process until a signal is received). When a signal is received, the signal handler prints out the context in which it caught the signal (via getpid) and then either exits (child process) or awaits the exit of the child (parent process).

LISTING 14.5 Signal Demonstration with a Parent and Child Process (on the CD-ROM at ./source/ch14/sigtest.c)

```
1:      #include <stdio.h>
2:      #include <sys/types.h>
3:      #include <sys/wait.h>
4:      #include <unistd.h>
5:      #include <signal.h>
6:      #include <errno.h>
7:
8:      void usr1_handler( int sig_num )
9:      {
10:
11:       printf( "Process (%d) got the SIGUSR1\n", getpid() );
12:
13:     }
14:
15:     int main()
16:     {
17:       pid_t ret;
18:       int   status;
```

```
19:         int   role = -1;
20:
21:         signal( SIGUSR1, usr1_handler );
22:
23:         ret = fork();
24:
25:         if (ret > 0) {                /* Parent Context */
26:
27:           printf( "Parent: This is the parent process (pid %d)\n",
28:                    getpid() );
29:
30:           role = 0;
31:
32:           pause();
33:
34:           printf( "Parent: Awaiting child exit\n" );
35:           ret = wait( &status );
36:
37:         } else if (ret == 0) {        /* Child Context */
38:
39:           printf( "Child: This is the child process (pid %d)\n",
40:                    getpid() );
41:
42:           role = 1;
43:
44:           pause();
45:
46:         } else {                     /* Parent Context – Error */
47:
48:           printf( "Parent: Error trying to fork() (%d)\n",
              errno );
49:
50:         }
51:
52:         printf( "%s: Exiting...\n",
53:                  ((role == 0) ? "Parent" : "Child") );
54:
55:         return 0;
56:       }
```

Now consider the sample output for this application to better understand what happens. Note that neither the parent nor the child raises any signals to each other. This example takes care of sending the signal at the command line, using the `kill` command.

```
# ./sigtest &
[1] 20152
# Child: This is the child process (pid 20153)
Parent: This is the parent process (pid 20152)

# kill -10 20152
Process (20152) got the SIGUSR1
Parent: Awaiting child exit
# kill -10 20153
Process (20153) got the SIGUSR1
Child: Exiting...
Parent: Exiting...
#
```

You begin by running the application (called `sigtest`) and placing it in the background (via the `&` symbol). You see the expected outputs from the child and parent processes identifying that the `fork` has occurred and that both processes are now active and awaiting signals at the respective `pause` calls. You use the `kill` command with the signal of interest (`–10`, or `SIGUSR1`) and the process identifier to which to send the signal. In this case, you send the first `SIGUSR1` to the parent process (20152). The parent immediately identifies receipt of the signal via the signal handler, but note that it executes within the context of the parent process (as identified by the process ID of 20152). The parent then returns from the `pause` function and awaits the exit of the child via the `wait` function. You then send another `SIGUSR1` signal to the child using the `kill` command. In this case, you direct the `kill` command to the child by its process ID (20153). The child also indicates receipt of the signal by the signal handler and in its own context. The child then exits and permits the parent to return from the `wait` function and exit also.

Despite the simplicity of the signals mechanism, it can be a powerful method to communicate with processes in an asynchronous fashion.

pause

The `pause` function suspends the calling process until a signal is received. After the signal is received, the calling process returns from the `pause` function, permitting it to continue. The prototype for the `pause` API function is as follows:

```
int pause( void );
```

If the process has installed a signal handler for the signal that was caught, then the `pause` function returns after the signal handler has been called and returns.

kill

The `kill` API function raises a signal to a process or set of processes. A return of zero indicates that the signal was successfully sent, otherwise –1 is returned. The `kill` function prototype is as follows:

```
int kill( pid_t pid, int sig_num );
```

The `sig_num` argument represents the signal to send. The `pid` argument can be a variety of different values (as shown in Table 14.9).

TABLE 14.9 Values of `pid` Argument for `kill` Function

`pid`	**Description**
0	Signal sent to the process defined by `pid`
0	Signal sent to all processes within the process group
1	Signal sent to all processes (except for the `init` process)
–1	Signal sent to all processes within the process group defined by the absolute value of `pid`

Some simple examples of the `kill` function follow. You can send a signal to yourself using the following code snippet:

```
kill( getpid(), SIGHUP );
```

The process group enables you to collect a set of processes together that can be signaled together as a group. API functions such as `getpgrp` (get process group) and `setpgrp` (set process group) can be used to read and set the process group identifier. You can send a signal to all processes within a defined process group as follows:

```
kill( 0, SIGUSR1 );
```

or to another process group as follows:

```
pid_t group;
...
kill( -group, SIGUSR1 );
```

You can also mimic the behavior of sending to the current process group by identifying the group and then passing the negative of this value to signal:

```
pid_t group = getpgrp();
...
kill( -group, SIGUSR1 );
```

Finally, you can send a signal to all processes (except for `init`) using the –1 `pid` identifier. This, of course, requires that you have permission to do this.

```
kill( -1, SIGUSR1 );
```

raise

The `raise` API function can be used to send a specific signal to the current process (the process context in which the `raise` function is called). The prototype for the `raise` function is as follows:

```
int raise( int sig_num );
```

The `raise` function is a constrained version of the `kill` API function that targets only the current process (`getpid()`).

exec Variants

The `fork` API function provided a mechanism to split an application into separate parent and child processes, sharing the same code but potentially serving different roles. The `exec` family of functions replaces the current process image altogether.

The `exec` *function starts a new program, replacing the current process, while retaining the current* `pid`.

The prototypes for the variants of `exec` are provided here:

```
int execl( const char *path, const char *arg, ... );
int execlp( const char *path, const char *arg, ... );
int execle( const char *path, const char *arg, ...,
            char * const envp[] );
int execv( const char *path, char *const argv[] );
int execvp( const char *file, char *const argv[] );
int execve( const char *filename, char *const argv[],
          char *const envp[] );
```

One of the notable differences between these functions is that one set takes a list of parameters (`arg0`, `arg1`, and so on) and the other takes an `argv` array. The `path` argument specifies the program to run, and the remaining parameters specify the arguments to pass to the program.

The `exec` commands permit the current process context to be replaced with the program (or command) specified as the first argument. Take a look at a quick example of `execcl` to achieve this:

```
execl( "/bin/ls", "ls", "-la", NULL );
```

This command replaces the current process with the `ls` image (list directory). You specify the command to execute as the first argument (including its path). The second argument is the command again (recall that `arg0` of the `main` program call is the name of the program). The third argument is an option that you pass to `ls`, and finally, you identify the end of your list with a `NULL`. Invoking an application that performs this command results in an `ls -la`.

The important item to note here is that the current process context is replaced by the command requested via `execl`. Therefore, when the preceding command is successfully executed, it never returns.

One additional item to note is that `execl` includes the absolute path to the command. If you choose to execute `execlp` instead, the full path is not required because the parent's `PATH` definition is used to find the command.

One interesting example of `execlp` is its use in creating a simple shell (on top of an existing shell). You support only simple commands within this shell (those that take no arguments). See Listing 14.6 for an example.

LISTING 14.6 Simple Shell Interpreter Using `execlp` (on the CD-ROM at `./source/ch14/simpshell.c`)

```
1:      #include <sys/types.h>
2:      #include <sys/wait.h>
3:      #include <unistd.h>
4:      #include <stdio.h>
5:      #include <stdlib.h>
6:      #include <string.h>
7:
8:      #define MAX_LINE        80
9:
10:     int main()
11:     {
12:       int status;
13:       pid_t childpid;
```

```
14:         char cmd[MAX_LINE+1];
15:         char *sret;
16:
17:         while (1) {
18:
19:           printf("mysh>");
20:
21:           sret = fgets( cmd, sizeof(cmd), stdin );
22:
23:           if (sret == NULL) exit(-1);
24:
25:           cmd[ strlen(cmd)-1] = 0;
26:
27:           if (!strncmp(cmd, "bye", 3)) exit(0);
28:
29:           childpid = fork();
30:
31:           if (childpid == 0) {
32:
33:             execlp( cmd, cmd, NULL );
34:
35:           } else if (childpid > 0) {
36:
37:             waitpid( childpid, &status, 0 );
38:
39:           }
40:
41:           printf("\n");
42:
43:         }
44:
45:         return 0;
46:       }
```

This shell interpreter is built around the simple parent/child `fork` application. The parent forks off the child (at line 29) and then awaits completion. The child takes the command read from the user (at line 21) and executes this using `execlp` (line 33). You simply specify the command as the command to execute and also include it for `arg0` (second argument). The `NULL` terminates the argument list; in this case no arguments are passed for the command. The child process never returns, but its `exit` status is recognized by the parent at the `waitpid` function (line 37).

As the user types in commands, they are executed via `execlp`. Typing in the command `bye` causes the application to exit.

Because no arguments are passed to the command (via `execlp`), the user can type in only commands and no arguments. Any arguments that are provided are simply ignored by the interpreter.

A sample execution of this application is shown here:

```
$ ./simpshell
mysh>date
Sat Apr 24 13:47:48 MDT 2004
mysh>ls
simpshell    simpshell.c
mysh>bye
$
```

You can see that after executing the shell, the prompt is displayed, indicating that commands can be entered. The `date` command is entered first, which provides the current date and time. Next, you do an `ls`, which gives the contents of the current directory. Finally, you exit the shell using the `bye` internal command.

Now take a look at one final `exec` variant as a way to explore the argument and environment aspects of a process. The `execve` variant allows an application to provide a command with a list of command-line arguments (as a vector) as well as an environment for the new process (as a vector of environment variables). Now take a look back at the `execve` prototype:

```
int execve( const char *filename, char *const argv[],
            char *const envp[] );
```

The `filename` argument is the program to execute, which must be a binary executable or a script that includes the `#!` interpreter spec at the top of the file. The `argv` argument is an array of arguments for the command, with the first argument being the command itself (the same as with the `filename` argument). Finally, the `envp` argument is an array of key/value strings containing environment variables. Consider the following simple example that retrieves the environment variables through the `main` function (on the CD-ROM at `./source/ch14/sigenv.c`):

```
#include <stdio.h>
#include <unistd.h>
int main( int argc, char *argv[], char *envp[] )
{
  int ret;
  char *args[]={ "ls", "-la", NULL };
```

```
    ret = execve( "/bin/ls", args, envp );
    fprintf( stderr, "execve failed\n" );
    return 0;
}
```

The first item to note in this example is the `main` function definition. You use a variant that passes in a third parameter that lists the environment for the process. This can also be gathered by the program using the special `environ` variable, which has the following definition:

```
extern char *environ[];
```

POSIX systems do not support the `envp` *argument to* `main`*, so it's best to use the* `environ` *variable.*

You specify your argument vector (`args`), which contains your command name and arguments, terminated by a `NULL`. This is provided as the argument vector to `execve`, along with the environment (passed in through the `main` function). This particular example simply performs an `ls` operation (by replacing the process with the `ls` command). Note also that you provide the `-la` option.

You can also specify your own environment similar to the `args` vector. For example, the following specifies a new environment for the process:

```
char *envp[] = { "PATH=/bin", "FOO=99", NULL };
...
ret = execve( command, args, envp );
```

The `envp` variable provides the set of variables that define the environment for the newly created process.

alarm

The `alarm` API function can be very useful to time out other functions. The `alarm` function works by raising a `SIGALRM` signal after the number of seconds passed to `alarm` has expired. The function prototype for `alarm` is as follows:

```
unsigned int alarm( unsigned int secs );
```

The user passes in the number of seconds to wait before sending the `SIGALRM` signal. The `alarm` function returns zero if no alarm was previously scheduled; otherwise, it returns the number of seconds pending on the previous alarm.

Here's an example of `alarm` to kill the current process if the user isn't able to enter a password in a reasonable amount of time (see Listing 14.7). At line 18, you install your signal handler for the `SIGALRM` signal. The signal handler is for the `wakeup` function (lines 6–9), which simply raises the `SIGKILL` signal. This terminates the application. You then emit the message to enter the password within three seconds and try to read the password from the keyboard (`stdin`). If the `read` call succeeds, you disable the alarm (by calling `alarm` with an argument of zero). The `else` portion of the test (line 30) checks the user password and continue. If the alarm times out, a `SIGALRM` is generated, resulting in a `SIGKILL` signal, which terminates the program.

LISTING 14.7 Sample Use of `alarm` and Signal Capture (on the CD-ROM at `./source/ch14/alarm.c`)

```
1:      #include <stdio.h>
2:      #include <unistd.h>
3:      #include <signal.h>
4:      #include <string.h>
5:
6:      void wakeup( int sig_num )
7:      {
8:        raise(SIGKILL);
9:      }
10:
11:     #define MAX_BUFFER      80
12:
13:     int main()
14:     {
15:       char buffer[MAX_BUFFER+1];
16:       int ret;
17:
18:       signal( SIGALRM, wakeup );
19:
20:       printf("You have 3 seconds to enter the password\n");
21:
22:       alarm(3);
23:
24:       ret = read( 0, buffer, MAX_BUFFER );
25:
26:       alarm(0);
27:
28:       if (ret == -1) {
29:
```

```
30:         } else {
31:
32:           buffer[strlen(buffer)-1] = 0;
33:           printf("User entered %s\n", buffer);
34:
35:         }
36:
37:       }
```

exit

The `exit` API function terminates the calling process. The argument passed to `exit` is returned to the parent process as the status of the parent's `wait` or `waitpid` call. The function prototype for `exit` is as follows:

```
void exit( int status );
```

The process calling `exit` also raises a `SIGCHLD` to the parent process and frees the resources allocated by the process (such as open file descriptors). If the process has registered a function with `atexit` or `on_exit`, these are called (in the reverse order to their registration).

This call is very important because it indicates success or failure to the shell environment. Scripts that rely on a program's `exit` status can behave improperly if the application does not provide an adequate status. This call provides that linkage to the scripting environment. Returning zero to the script indicates a `TRUE` or `SUCCESS` indication.

POSIX Signals

Before this discussion of process-related functions ends, you need to take a quick look at the POSIX signal APIs. The POSIX-compliant signals were introduced first in BSD and provide a portable API over the use of the `signal` API function. Have a look at a multiprocess application that uses the `sigaction` function to install a signal handler. The `sigaction` API function has the following prototype:

```
#include <signal.h>
int sigaction( int signum,
               const struct sigaction *act,
               struct sigaction *oldact );
```

`signum` is the signal for which you're installing the handler, `act` specifies the action to take for `signum`, and `oldact` is used to store the previous action. The `sigaction` structure contains a number of elements that can be configured:

```
struct sigaction {
    void (*sa_handler)( int );
    void (*sa_sigaction)( int, siginfo_t *, void * );
    sigset_t sa_mask;
    int sa_flags;
};
```

The `sa_handler` is a traditional signal handler that accepts a single argument (and `int` represents the signal). The `sa_sigaction` is a more refined version of a signal handler. The first `int` argument is the signal, and the third `void*` argument is a context variable (provided by the user). The second argument (`siginfo_t`) is a special structure that provides more detailed information about the signal that was generated:

```
siginfo_t {
  int         si_signo;     /* Signal number */
  int         si_errno;     /* Errno value */
  int         si_code;      /* Signal code */
  pid_t       si_pid;       /* Pid of signal sending process */
  uid_t       si_uid;       /* User id of signal sending process */
  int         si_status;    /* Exit value or signal */
  clock_t     si_utime;     /* User time consumed */
  clock_t     si_stime;     /* System time consumed */
  sigval_t    si_value      /* Signal value */
  int         si_int;       /* POSIX.1b signal */
  void *      si_ptr        /* POSIX.1b signal */
  void *      si_addr       /* Memory location which caused fault */
  int         si_band;      /* Band Event */
  int         si_fd;        /* File Descriptor */
}
```

One of the interesting items to note from `siginfo_t` is that with this API, you can identify the source of the signal (`si_pid`). The `si_code` field can be used to identify how the signal was raised. For example, if its value is `SI_USER`, then it was raised by a `kill`, `raise`, or `sigsend` API function. If its value is `SI_KERNEL`, then it was raised by the kernel. `SI_TIMER` indicates that a timer expired and resulted in the signal generation.

The `si_signo`, `si_errno`, and `si_code` are set for all signals. The `si_addr` field (indicating the memory location where the fault occurred) is set for `SIGILL`, `SIGFPE`, `SIGSEGV`, and `SIGBUS`. The `sigaction` main page identifies which fields are relevant for which signals.

The sa_flags argument of sigaction allows a modification of the behavior of the sigaction function. For example, if you provide SA_SIGINFO, then the sigaction uses the sa_sigaction field to identify the signal handler instead of sa_handler. Flag SA_ONESHOT can be used to restore the signal handler to the prior state after the signal handler has been called once. The SA_NOMASK (or SA_NODEFER) flag can be used to not inhibit the reception of the signal while in the signal handler (use with care).

A sample function is provided in Listing 14.8. The only real difference you see here from other examples is that sigaction is used at line 49 to install your signal handler. You create a sigaction structure at line 42, then initialize it with your function at line 48, and also identify that you're using the new sigaction handler via the SA_SIGINFO flag at line 47. When your signal finally fires (at line 34 in the parent process), your signal handler emits the originating process ID at line 12 (using the si_pid field of the siginfo reference).

LISTING 14.8 Simple Application Illustrating sigaction for Signal Installation (on the CD-ROM at ./source/ch14/posixsig.c)

```
1:      #include <sys/types.h>
2:      #include <sys/wait.h>
3:      #include <signal.h>
4:      #include <stdio.h>
5:      #include <unistd.h>
6:      #include <errno.h>
7:
8:      static int stopChild = 0;
9:
10:     void sigHandler( int sig, siginfo_t *siginfo, void *ignore )
11:     {
12:       printf("Got SIGUSR1 from %d\n", siginfo->si_pid);
13:       stopChild=1;
14:
15:       return;
16:     }
17:
18:     int main()
19:     {
20:       pid_t ret;
21:       int   status;
22:       int   role = -1;
23:
24:       ret = fork();
25:
```

```
26:         if (ret > 0) {
27:
28:           printf("Parent: This is the parent process (pid %d)\n",
29:                   getpid());
30:
31:           /* Let the child init */
32:           sleep(1);
33:
34:           kill( ret, SIGUSR1 );
35:
36:           ret = wait( &status );
37:
38:           role = 0;
39:
40:         } else if (ret == 0) {
41:
42:           struct sigaction act;
43:
44:           printf("Child: This is the child process (pid %d)\n",
45:                   getpid());
46:
47:           act.sa_flags = SA_SIGINFO;
48:           act.sa_sigaction = sigHandler;
49:           sigaction( SIGUSR1, &act, 0 );
50:
51:           printf("Child Waiting...\n");
52:           while (!stopChild);
53:
54:           role = 1;
55:
56:         } else {
57:
58:           printf("Parent: Error trying to fork() (%d)\n", errno);
59:
60:         }
61:
62:         printf("%s: Exiting...\n",
63:                 ((role == 0) ? "Parent" : "Child"));
64:
65:         return 0;
66:       }
```

The `sigaction` function provides a more advanced mechanism for signal handling, in addition to greater portability. For this reason, `sigaction` should be used over `signal`.

System Commands

This section takes a look at a few of the GNU/Linux commands that work with the previously mentioned API functions. It looks at commands that permit you to inspect the process list and send a signal to a process or to an entire process group.

ps

The `ps` command provides a snapshot in time of the current set of processes active on a given system. The `ps` command takes a large variety of options; this section explores a few.

In the simplest form, you can type `ps` at the keyboard to see a subset of the processes that are active:

```
$ ps
  PID TTY          TIME CMD
22001 pts/0    00:00:00 bash
22186 pts/0    00:00:00 ps
$
```

First, you see your `bash` session (your own process) and your `ps` command process (every command in GNU/Linux is executed within its own subprocess). You can see all of the processes running using the `-a` option (this list is shortened for brevity):

```
$ ps -a
  PID TTY          TIME CMD
    1 ?        00:00:05 init
    2 ?        00:00:00 keventd
    3 ?        00:00:00 kapmd
    4 ?        00:00:00 ksoftirqd_CPU0
...
22001 pts/0    00:00:00 bash
22074 ?        00:00:00 sendmail
22189 pts/0    00:00:00 ps
$
```

In this example, you see a number of other processes including the mother of all processes (`init`, process ID 1) and assorted kernel threads. If you want to see only those processes that are associated with your user, you can accomplish this with the `–User` option:

```
$ ps –User mtj
  PID TTY          TIME CMD
22000 ?        00:00:00 sshd
22001 pts/0    00:00:00 bash
22190 pts/0    00:00:00 ps
$
```

Another very useful option is `-H`, which tells you the process hierarchy. In the next example, you request all processes for user `mtj` but then also request their hierarchy (parent/child relationships):

```
$ ps –User mtj -H
  PID TTY          TIME CMD
22000 ?        00:00:00 sshd
22001 pts/0    00:00:00   bash
22206 pts/0    00:00:00     ps
#
```

Here you see that the base process is an `sshd` session (because you are connected to this server via the secure shell). This is the parent of the `bash` session, which in turn is the parent of the `ps` command that you just executed.

The `ps` command can be very useful, especially when you're interested in finding your process identifiers to kill a process or send it a signal.

top

The `top` command is related to `ps`, but `top` runs in real time and lists the activity of the processes for the given CPU. In addition to the process list, you can also see statistics about the CPU (number of processes, number of zombies, memory used, and so on). You're obviously in need of a memory upgrade here (only 4 MB free). This sample list has again been shortened for brevity.

```
 19:27:49  up 79 days, 10:04,  2 users,  load average: 0.00, 0.00, 0.00
47 processes: 44 sleeping, 3 running, 0 zombie, 0 stopped
CPU states:   0.0% user   0.1% system   0.0% nice   0.0% iowait  99.8% idle
Mem:   124984k av,  120892k used,    4092k free,       0k shrd,   52572k buff
                     79408k actv,       4k in_d,     860k in_c
```

```
Swap:  257032k av,    5208k used,  251824k free                  37452k
                                                                 cached
  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND
22226 mtj       15   0  1132 1132   868 R     0.1  0.9   0:00   0 top
    1 root      15   0   100   76    52 S     0.0  0.0   0:05   0 init
    2 root      15   0     0    0     0 SW    0.0  0.0   0:00   0 keventd
    3 root      15   0     0    0     0 RW    0.0  0.0   0:00   0 kapmd
    4 root      34  19     0    0     0 SWN   0.0  0.0   0:00   0 ksoftirqd_
                                                                 CPU0
...
 1708 root      15   0   196    4     0 S     0.0  0.0   0:00   0 login
 1709 root      15   0   284    4     0 S     0.0  0.0   0:00   0 bash
22001 mtj       15   0  1512 1512  1148 S     0.0  1.2   0:00   0 bash
```

The rate of sampling can also be adjusted for `top`, in addition to a number of other options (see the `top` man page for more details).

kill

The `kill` command, like the `kill` API function, allows you to send a signal to a process. You can also use it to list the signals that are relevant for the given processor architecture. For example, if you want to see the signals that are available for the given processor, you use the `-l` option:

```
# kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      33) SIGRTMIN    34) SIGRTMIN+1
35) SIGRTMIN+2  36) SIGRTMIN+3  37) SIGRTMIN+4  38) SIGRTMIN+5
39) SIGRTMIN+6  40) SIGRTMIN+7  41) SIGRTMIN+8  42) SIGRTMIN+9
43) SIGRTMIN+10 44) SIGRTMIN+11 45) SIGRTMIN+12 46) SIGRTMIN+13
47) SIGRTMIN+14 48) SIGRTMIN+15 49) SIGRTMAX-14 50) SIGRTMAX-13
51) SIGRTMAX-12 52) SIGRTMAX-11 53) SIGRTMAX-10 54) SIGRTMAX-9
55) SIGRTMAX-8  56) SIGRTMAX-7  57) SIGRTMAX-6  58) SIGRTMAX-5
59) SIGRTMAX-4  60) SIGRTMAX-3  61) SIGRTMAX-2  62) SIGRTMAX-1
63) SIGRTMAX
#
```

For a running process, you can send a signal as follows. In this example, you send the SIGSTOP signal to the process identified by the process ID 23000.

```
# kill -s SIGSTOP 23000
```

This places the process in the STOPPED state (not running). You can start the process up again by giving it the SIGCONT signal, as follows:

```
# kill -s SIGCONT 23000
```

Like the kill API function, you can signal an entire process group by providing a pid of 0. Similarly, all processes within the process group can be sent a signal by sending the negative of the process group.

Summary

This chapter explored the traditional process API provided in GNU/Linux. You investigated process creation with fork, validating the status return of fork, and various process-related API functions such as getpid (get process ID) and getppid (get parent process ID). The chapter then looked at process support functions such as wait and waitpid and the signal mechanism that permits processes to communicate with one another. Finally, you looked at a number of GNU/Linux commands that enable you to review active processes and also the commands to signal them.

Proc Filesystem

The /proc filesystem is the root source of information about the processes within a GNU/Linux system. Within /proc, you'll find a set of directories with numbered filenames. These numbers represent the process IDs (pids) of active processes within the system. The root-level view of /proc is provided in the following:

```
# ls /proc
1      4     5671  7225  9780       crypto       kcore       stat
10     4307  6     7255  9783       devices      key-users   swaps
19110  4524  6265  7265  9786       diskstats    kmsg        sys
2      5     6387  7360  9787       dma          loadavg     sysrq-
trigger
2132   5009  6416  8134  9788       driver       locks       sysvipc
21747  5015  6446  9     9789       execdomains  mdstat      tty
```

```
21748  5312  6671  94    9790       fb           meminfo     uptime
21749  5313  6672  95    9791       filesystems  misc        version
21751  5340  6673  9650  9795       fs           modules     vmstat
2232   5341  6674  9684  9797       ide          mounts      zoneinfo
24065  5342  6675  9688  acpi       interrupts   mtrr
2623   5343  683   9689  buddyinfo  iomem        net
3      5344  6994  9714  bus        ioports      partitions
32618  5560  7     9715  cmdline    irq          self
3651   5670  7224  9773  cpuinfo    kallsyms     slabinfo
#
```

Each `pid` directory presents a hierarchy of information about that process including the command line that started it, a symlink to the root filesystem (which can be different from the current root if the executable was chrooted), a symlink to the directory (current working directory) where the process was started, and others. The following is a look at a `pid` directory hierarchy:

```
# ls /proc/1
attr     cpuset   exe   mem      oom_score  smaps  status
auxv     cwd      fd    mounts   root       stat   task
cmdline  environ  maps  oom_adj  seccomp    statm  wchan
#
```

Recall that `pid 1` is the first process to execute and is always the `init` process. You can view this from the status file that contains basic information about the process including its state, memory usage, signal masks, etc.

```
# cat status
Name:   init
State:  S (sleeping)
SleepAVG:       88%
Tgid:   1
Pid:    1
PPid:   0
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 32
Groups:
...
#
```

The /proc filesystem also contains a large number of other nonprocess-specific elements, some of which can be written to change the behavior of the overall operating system. Many utilities use information from the /proc filesystem to present data to the user (for example, the ps command uses /proc to get its process list).

References

GNU/Linux signal and sigaction man pages.

API Summary

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>
pid_t fork( void );
pid_t wait( int *status );
pid_t waitpid( pid_t pid, int *status, int options );
sighandler_t signal( int signum, sighandler_t handler );
int pause( void );
int kill( pid_t pid, int sig_num );
int raise( int sig_num );
int execl( const char *path, const char *arg, ... );
int execlp( const char *path, const char *arg, ... );
int execle( const char *path, const char *arg, ...,
            char * const envp[] );
int execv( const char *path, char *const argv[] );
int execvp( const char *file, char *const argv[] );
int execve( const char *filename, char *const argv[],
          char *const envp[] );
unsigned int alarm( unsigned int secs );
void exit( int status );
int sigaction( int signum,
           const struct sigaction *act,
           struct sigaction *oldact );
```

15 POSIX Threads (pthreads) Programming

In This Chapter

- Threads and Processes
- Creating Threads
- Synchronizing Threads
- Communicating Between Threads
- POSIX Signals API
- Threaded Application Development Topics

Introduction

Multithreaded applications are a useful paradigm for system development because they offer many facilities not available to traditional GNU/Linux processes. This chapter explores `pthreads` programming and the functionality provided by the `pthreads` API.

The 2.4 GNU/Linux kernel POSIX thread library was based upon the LinuxThreads implementation (introduced in 1996), which was built on the existing GNU/Linux process model. The 2.6 kernel utilizes the new Native POSIX Thread Library, or NPTL (introduced in 2002), which is a higher performance implementation with numerous advantages over the older component. For example, NPTL provides real thread groups (within a process), compared to one thread per process in the prior model. This chapter outlines those differences when they are useful to know.

To know which `pthreads` *library is being used, issue the following command:*

```
$ getconf GNU_LIBPTHREAD_VERSION
```

This provides either LinuxThreads or NPTL, each with a version number.

What's a Thread?

To define a thread, you need to look back at Linux processes to understand their makeup. Both processes and threads have control flows and can run concurrently, but they differ in some very distinct ways. Threads, for example, share data, where processes explicitly don't. When a process is forked (recall from Chapter 13, "Introduction to Sockets Programming"), a new process is created with its own globals and stack (see Figure 15.1). When a thread is created, the only new element created is a stack that is unique for the thread (see Figure 15.2). The code and global data are common between the threads. This is advantageous, but the shared nature of threads can also be problematic. This chapter investigates this later.

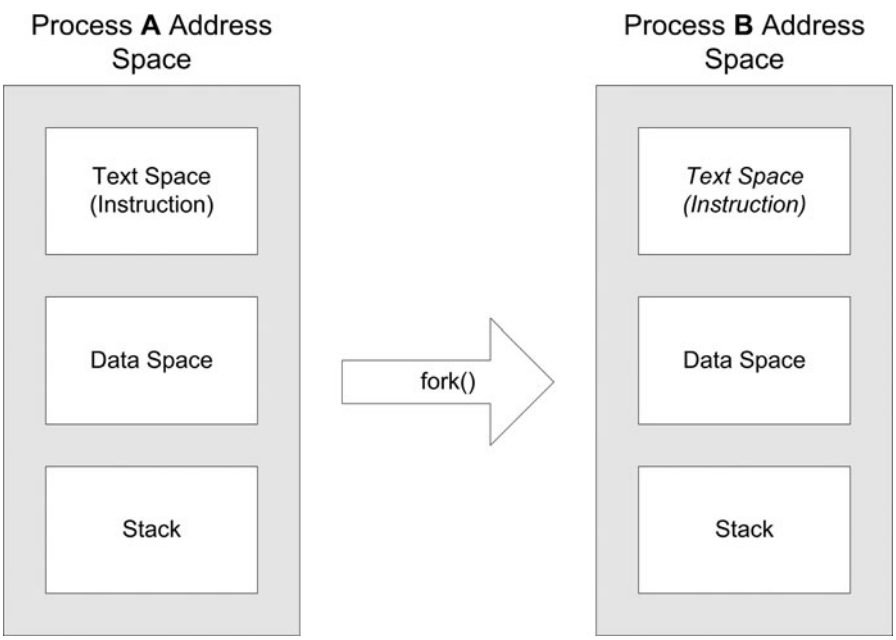

FIGURE 15.1 Forking a new process.

A GNU/Linux process can create and manage numerous threads. Each thread is identified by a thread identifier that is unique for every thread in a system. Each thread also has its own stack (as shown in Figure 15.2) and also a unique context (program counter, save registers, and so forth). But because the data space is shared by threads, they share more than just user data. For example, file descriptors for open files or sockets are shared also. Therefore, when a multithreaded application uses a socket or file, the access to the resource must be protected against multiple accesses. This chapter looks at methods for achieving that.

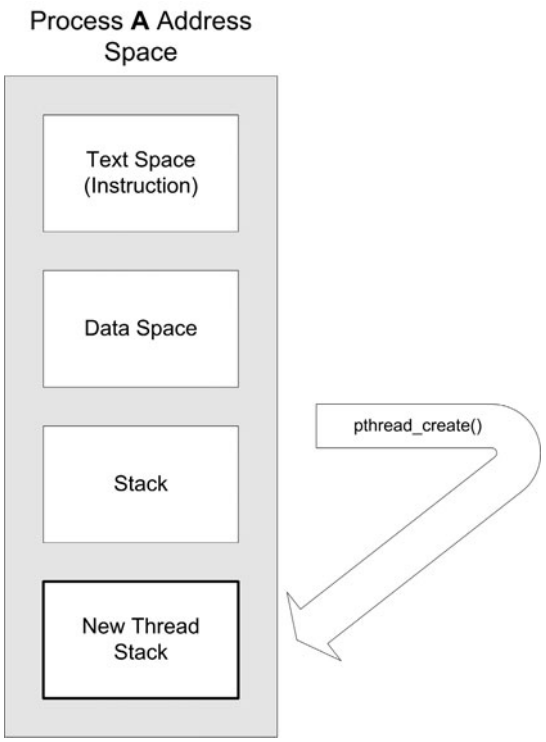

FIGURE 15.2 Creating a new thread.

While writing multithreaded applications can be easier in some ways than traditional process-based applications, you do encounter problems you need to understand. The shared data aspect of threads is probably the most difficult to design around, but it is also powerful and can lead to simpler applications with higher performance. The key is to strongly consider shared data while developing threaded applications. Another important consideration is that serious multithreaded application development needs to utilize the 2.6 kernel rather than the 2.4 kernel (given the new NPTL threads implementation).

Thread Function Basics

The APIs discussed thus far follow a fairly uniform model of returning –1 when an error occurs, with the actual error value in the `errno` process variable. The threads API returns 0 on success but a positive value to indicate an error.

The `pthreads` API

While the `pthreads` API is comprehensive, it's quite easy to understand and use. This section explores the `pthreads` API, looking at the basics of thread creation through the specialized communication and synchronization methods that are available.

All multithreaded programs must make the `pthread` function prototypes and symbols available for use. This is accomplished by including the `pthread` standard header, as follows:

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

The examples that follow are written for brevity, and in some cases, return values are not checked. To avoid debugging surprises, you are strongly encouraged to check all system call return values and never assume that a function is successful.

Thread Basics

All multithreaded applications must create threads and ultimately destroy them. This is provided in two functions by the `pthreads` API:

```
int pthread_create( pthread_t *thread,
                    pthread_attr_t *attr,
                void *(*start_routine)(void *), void *arg );
int pthread_exit( void *retval );
```

The `pthread_create` function permits the creation of a new thread, whereas `pthread_exit` allows a thread to terminate itself. You also have a function to permit one thread to terminate another, but that is investigated later.

To create a new thread, you call `pthread_create` and associate your `pthread_t` object with a function (`start_routine`). This function represents the top level code that is executed within the thread. You can optionally provide a set of attributes via `pthread_attr_t` (via `pthread_attr_init`). Finally, the fourth argument (`arg`) is an optional argument that is passed to the thread upon creation.

Now it's time to take a look at a short example of thread creation (see Listing 15.1). In the `main` function, you first create a `pthread_t` object at line 10. This object represents your new thread. You call `pthread_create` at line 12 and provide the `pthread_t` object (which is filled in by the `pthread_create` function) in addition to your function that contains the code for the thread (argument 3, `myThread`). A zero return indicates successful creation of the thread.

LISTING 15.1 Creating a Thread with `pthread_create` (on the CD-ROM at `./source/ch15/ptcreate.c`)

```
1:        #include <pthread.h>
2:        #include <stdlib.h>
3:        #include <stdio.h>
4:        #include <string.h>
5:        #include <errno.h>
6:
7:        int main()
8:        {
9:          int ret;
10:         pthread_t mythread;
11:
12:         ret = pthread_create( &mythread, NULL, myThread, NULL );
13:
14:         if (ret != 0) {
15:           printf( "Can't create pthread (%s)\n", strerror(
              errno ) );
16:           exit(-1);
17:         }
18:
19:         return 0;
20:       }
```

The `pthread_create` function returns zero if successful; otherwise, a nonzero value is returned. Now it's time to take a look at the thread function itself, which also demonstrates the `pthread_exit` function (see Listing 15.2). The thread simply emits a message to `stdout` that it ran and then terminated at line 6 with `pthread_exit`.

LISTING 15.2 Terminating a Thread with `pthread_exit` (on the CD-ROM at `./source/ch15/ptcreate.c`)

```
1:        void *myThread( void *arg )
2:        {
```

```
3:          printf("Thread ran!\n");
4:
5:          /* Terminate the thread */
6:          pthread_exit( NULL );
7:        }
```

This thread didn't use the `void` pointer argument, but this could be used to provide the thread with a specific personality, passed in at creation (see the fourth argument of line 12 in Listing 15.1). The argument can represent a scalar value or a structure containing a variety of elements. The `exit` value presented to `pthread_exit` must not be of local scope; otherwise, it won't exist after the thread is destroyed. The `pthread_exit` function does not return.

The startup cost for new threads is minimal in the new NPTL implementation, compared to the older LinuxThreads. In addition to significant improvements and optimizations in the NPTL, the allocation of thread memory structures is improved (thread data structures and thread local storage are now provided on the local thread stack).

Thread Management

Before digging into thread synchronization and coordination, it's time to look at a couple of miscellaneous thread functions that can be of use. The first is the `pthread_self` function, which can be used by a thread to retrieve its unique identifier. Recall in `pthread_create` that a `pthread_t` object reference is passed in as the first argument. This permits the thread creator to know the identifier for the thread just created. The thread itself can also retrieve this identifier by calling `pthread_self`.

```
pthread_t pthread_self( void );
```

Consider the updated thread function in Listing 15.3, which illustrates retrieving the `pthread_t` handle. At line 5, you call `pthread_self` to grab the handle and then emit it to `stdout` at line 7 (converting it to an `int`).

LISTING 15.3 Retrieving the `pthread_t` Handle with `pthread_self` (on the CD-ROM at `./source/ch15/ptcreate.c`)

```
1:        void *myThread( void *arg )
2:        {
3:          pthread_t pt;
4:
```

```
5:          pt = pthread_self();
6:
7:          printf("Thread %x ran!\n", (int)pt );
8:
9:          pthread_exit( NULL );
10:       }
```

Most applications require some type of initialization, but with threaded applications, the job can be difficult. The `pthread_once` function allows a developer to create an initialization routine that is invoked for a multithreaded application only once (even though multiple threads might attempt to invoke it).

The `pthread_once` function requires two objects: a `pthread_once_t` object (that has been preinitialized with `pthread_once_init`) and an initialization function. Consider the partial example in Listing 15.4. The first thread to call `pthread_once` invokes the initialization function (`initialize_app`), but subsequent calls to `pthread_once` result in no calls to `initialize_app`.

LISTING 15.4 Providing a Single-Use Initialization Function with `pthread_once`

```
1:        #include <pthread.h>
2:
3:        pthread_once_t my_init_mutex = pthread_once_init;
4:
5:        void initialize_app( void )
6:        {
7:          /* Single-time init here */
8:        }
9:
10:       void *myThread( void *arg )
11:       {
12:         ...
13:
14:         pthread_once( &my_init_mutex, initialize_app );
15:
16:         ...
17:       }
```

The number of threads in LinuxThreads was a compile-time option (1000), whereas NPTL supports a dynamic number of threads. NPTL can support up to 2 billion threads on an IA-32 system [Drepper and Molnar03].

Thread Synchronization

The ability to synchronize threads is an important aspect of multithreaded application development. This chapter looks at a number of methods, but first you need to take a look at the most basic method, the ability for the creator thread to wait for the created thread to finish (otherwise known as a `join`). This activity is provided by the `pthread_join` API function. When called, the `pthread_join` call suspends the calling thread until a `join` is complete. When the `join` is done, the caller receives the joined thread's termination status as the return from `pthread_join`. The `pthread_join` function (somewhat equivalent to the `wait` function for processes) has the following prototype:

```
int pthread_join( pthread_t th, void **thread_return );
```

The `th` argument is the thread to which you want to join. This argument is returned from `pthread_create` or passed via the thread itself via `pthread_self`. The `thread_return` can be `NULL`, which means you do not capture the return status of the thread. Otherwise, the return value from the thread is stored in `thread_return`.

A thread is automatically joinable when using the default attributes of `pthread_create`. *If the attribute for the thread is defined as detached, then the thread can't be joined (because it's detached from the creating thread).*

To join with a thread, you must have the thread's identifier, which is retrieved from the `pthread_create` function. Take a look at a complete example (see Listing 15.5).

In this example, you permit the creation of five distinct threads by calling `pthread_create` within a loop (lines 18–23) and storing the resulting thread identifiers in a `pthread_t` array (line 16). After the threads are created, you begin the `join` process, again in a loop (lines 25–32). The `pthread_join` returns zero on success, and upon success, the `status` variable is emitted (note that this value is returned at line 8 within the thread itself).

LISTING 15.5 Joining Threads with `pthread_join` (on the CD-ROM at `./source/ch15/ptjoin.c`)

```
1:      #include <pthread.h>
2:      #include <stdio.h>
3:
4:      void *myThread( void *arg )
5:      {
```

```
6:          printf( "Thread %d started\n", (int)arg );
7:
8:          pthread_exit( arg );
9:        }
10:
11:       #define MAX_THREADS     5
12:
13:       int main()
14:       {
15:         int ret, i, status;
16:         pthread_t threadIds[MAX_THREADS];
17:
18:         for (i = 0 ; i < MAX_THREADS ; i++) {
19:           ret = pthread_create( &threadIds[i], NULL, myThread,
              (void *)i );
20:           if (ret != 0) {
21:             printf( "Error creating thread %d\n",
                (int)threadIds[i] );
22:           }
23:         }
24:
25:         for (i = 0 ; i < MAX_THREADS ; i++) {
26:           ret = pthread_join( threadIds[i], (void **)&status );
27:           if (ret != 0) {
28:             printf( "Error joining thread %d\n",
                (int)threadIds[i] );
29:           } else {
30:             printf( "Status = %d\n", status );
31:           }
32:         }
33:
34:         return 0;
35:       }
```

The `pthread_join` function suspends the caller until the requested thread has been joined. In many cases, you simply don't care about the thread after it's created. In these cases, you can identify this by detaching the thread. The creator or the thread itself can detach itself. You can also specify that the thread is detached when you create the thread (as part of the attributes). After a thread is detached, it can never be joined. The `pthread_detach` function has the following prototype:

```
int pthread_detach( pthread_t th );
```

Now take a look at the process of detaching the thread within the thread itself (see Listing 15.6). Recall that a thread can identify its own identifier by calling `thread_self`.

LISTING 15.6 Detaching a Thread from Within with `pthread_detach`

```
1:        void *myThread( void *arg )
2:        {
3:          printf( "Thread %d started\n", (int)arg );
4:
5:          pthread_detach( pthread_self() );
6:
7:          pthread_exit( arg );
8:         }
```

At line 5, you simply call `pthread_detach`, specifying the thread identifier by calling `pthread_self`. When this thread exits, all resources are immediately freed (as it's detached and will never be joined by another thread). The `pthread_detach` function returns zero on success, nonzero if an error occurs.

GNU/Linux automatically places a newly created thread into the joinable state. This is not the case in other implementations, which can default to detached.

Thread Mutexes

A mutex is a variable that permits threads to implement critical sections. These sections enforce exclusive access to variables by threads, which if left unprotected result in data corruption. This topic is discussed in detail in Chapter 17, "Synchronization with Semaphores."

This section starts by reviewing the mutex API, and then illustrates the problem being solved. To create a mutex, you simply declare a variable that represents your mutex and initializes it with a special symbolic constant. The mutex is of type `pthread_mutex_t` and is demonstrated as follows:

```
pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER
```

As shown here, the initialization makes this mutex a fast mutex. The mutex initializer can actually be of one of three types, as shown in Table 15.1.

TABLE 15.1 Mutex Initializers

Type	Description
PTHREAD_MUTEX_INITIALIZER	Fast mutex
PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP	Recursive mutex
PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP	Error-checking mutex

The recursive mutex is a special mutex that allows the mutex to be locked several times (without blocking), as long as it's locked by the same thread. Even though the mutex can be locked multiple times without blocking, the thread must unlock the mutex the same number of times that it was locked. The error-checking mutex can be used to help find errors when debugging. Note that the `_NP` suffix for recursive and error-checking mutexes indicates that they are not portable.

Now that you have a mutex, you can lock and unlock it to create your critical section. This is done with the `pthread_mutex_lock` and `pthread_mutex_unlock` API functions. Another function called `pthread_mutex_trylock` can be used to try to lock a mutex, but it won't block if the mutex is already locked. Finally, you can destroy an existing mutex using `pthread_mutex_destroy`. These have the prototype as follows:

```
int pthread_mutex_lock( pthread_mutex_t *mutex );
int pthread_mutex_trylock( pthread_mutex_t *mutex );
int pthread_mutex_unlock( pthread_mutex_t *mutex );
int pthread_mutex_destroy( pthread_mutex_t *mutex );
```

All functions return zero on success or a nonzero error code. All errors returned from `pthread_mutex_lock` and `pthread_mutex_unlock` are assertable (not recoverable). Therefore, you use the return of these functions to abort your program.

Locking a thread is the means by which you enter a critical section. After your mutex is locked, you can safely enter the section without having to worry about data corruption or multiple access. To exit your critical section, you unlock the semaphore and you're done. The following code snippet illustrates a simple critical section:

```
pthread_mutex_t cntr_mutex = PTHREAD_MUTEX_INITIALIZER;
...
assert( pthread_mutex_lock( &cntr_mutex ) == 0 );
/* Critical Section */
```

```
/* Increment protected counter */
counter++;
/* Critical Section */
  assert( pthread_mutex_unlock( &cntr_mutex ) == 0 );
```

A critical section is a section of code that can be executed by at most one process at a time. The critical section exists to protect shared resources from multiple access.

The `pthread_mutex_trylock` operates under the assumption that if you can't lock your mutex, you should do something else instead of blocking on the `pthread_mutex_lock` call. This call is demonstrated as follows:

```
ret = pthread_mutex_trylock( &cntr_mutex );
if (ret == EBUSY) {
  /* Couldn't lock, do something else */
} else if (ret == EINVAL) {
  /* Critical error */
  assert(0);
} else {
  /* Critical Section */
  ret = thread_mutex_unlock( &cntr_mutex );
}
```

Finally, to destroy your mutex, you simply provide it to the `pthread_mutex_destroy` function. The `pthread_mutex_destroy` function succeeds only if no thread currently has the mutex locked. If the mutex is locked, the function fails and returns the `EBUSY` error code. The `pthread_mutex_destroy` call is demonstrated with the following snippet:

```
ret = pthread_mutex_destroy( &cntr_mutex );
if (ret == EBUSY) {
  /* Mutex is locked, can't destroy */
} else {
  /* Mutex was destroyed */
}
```

Now take a look at an example that ties these functions together to illustrate why mutexes are important in multithreaded applications. In this example, you build on the previous applications that provide a basic infrastructure for task creation and joining. Consider the example in Listing 15.7. At line 4, you create your mutex and initialize it as a fast mutex. In your thread, your job is to increment the `protVariable` counter some number of times. This occurs for each thread (here you

create 10), so you need to protect the variable from multiple access. You place the variable increment within a critical section by first locking the mutex and then, after incrementing the protected variable, unlocking it. This ensures that each task has sole access to the resource when the increment is performed and protects it from corruption. Finally, at line 52, you destroy your mutex using the `pthread_mutex_destroy` API function.

LISTING 15.7 Protecting a Variable in a Critical Section with Mutexes (on the CD-ROM at `./source/ch15/ptmutex.c`)

```
1:      #include <pthread.h>
2:      #include <stdio.h>
3:
4:      pthread_mutex_t cntr_mutex = PTHREAD_MUTEX_INITIALIZER;
5:
6:      long protVariable = 0L;
7:
8:      void *myThread( void *arg )
9:      {
10:       int i, ret;
11:
12:       for (i = 0 ; i < 10000 ; i++) {
13:
14:         ret = pthread_mutex_lock( &cntr_mutex );
15:
16:         assert( ret == 0 );
17:
18:         protVariable++;
19:
20:         ret = pthread_mutex_unlock( &cntr_mutex );
21:
22:         assert( ret == 0 );
23:
24:       }
25:
26:       pthread_exit( NULL );
27:     }
28:
29:     #define MAX_THREADS     10
30:
31:     int main()
32:     {
33:       int ret, i;
```

```
34:         pthread_t threadIds[MAX_THREADS];
35:
36:         for (i = 0 ; i < MAX_THREADS ; i++) {
37:           ret = pthread_create( &threadIds[i], NULL, myThread,
              NULL );
38:           if (ret != 0) {
39:             printf( "Error creating thread %d\n",
                (int)threadIds[i] );
40:           }
41:         }
42:
43:         for (i = 0 ; i < MAX_THREADS ; i++) {
44:           ret = pthread_join( threadIds[i], NULL );
45:           if (ret != 0) {
46:             printf( "Error joining thread %d\n",
                (int)threadIds[i] );
47:           }
48:         }
49:
50:         printf( "The protected variable value is %ld\n",
            protVariable );
51:
52:         ret = pthread_mutex_destroy( &cntr_mutex );
53:
54:         if (ret != 0) {
55:           printf( “Couldn’t destroy the mutex\n”);
56:         }
57:
58:         return 0;
59:       }
```

When using mutexes, it’s important to minimize the amount of work done in the critical section to what really needs to be done. Because other threads block until a mutex is unlocked, minimizing the critical section time can lead to better performance.

Thread Condition Variables

Now that you have mutexes out of the way, you can explore condition variables. A condition variable is a special thread construct that allows a thread to wake up another thread based upon a condition. Whereas mutexes provide a simple form of synchronization (based upon the lock status of the mutex), condition variables are

a means for one thread to wait for an event and another to signal it that the event has occurred. An event can mean anything here. A thread blocks on a mutex but can wait on any condition variable. Think of them as wait queues, which is exactly what the implementation does in GNU/Linux.

Consider this problem of a thread awaiting a particular condition being met. If you use only mutexes, the thread has to poll to acquire the mutex, check the condition, and then release the mutex if no work is found to do (the condition isn't met). That kind of busy looping can lead to poorly performing applications and needs to be avoided.

The `pthreads` API provides a number of functions supporting condition variables. These functions provide condition variable creation, waiting, signaling, and destruction. The condition variable API functions are presented as follows:

```
int pthread_cond_wait( pthread_cond_t *cond,
                       pthread_mutex_t *mutex );
int pthread_cond_timedwait( pthread_cond_t *cond,
                            pthread_mutex_t *mutex,
                            const struct timespec *abstime );
int pthread_cond_signal( pthread_cond_t *cond );
int pthread_cond_broadcast( pthread_cond_t *cond );
int pthread_cond_destroy( pthread_cond_t *cond );
```

To create a condition variable, you simply create a variable of type `pthread_cond_t`. You initialize this by setting it to `PTHREAD_COND_INITIALIZER` (similar to mutex creation and initialization). This is demonstrated as follows:

```
pthread_cond_t recoveryCond = PTHREAD_COND_INITIALIZER;
```

Condition variables require the existence of a mutex that is associated with them, which you create as you learned previously:

```
pthread_mutex_t recoveryMutex = PTHREAD_MUTEX_INITIALIZER;
```

Now take a look at a thread awaiting a condition. In this example, say you have a thread whose job is to warn of overload conditions. Work comes in on a queue, with an accompanying counter identifying the amount of work to do. When the amount of work exceeds a certain value (`MAX_NORMAL_WORKLOAD`), then your thread wakes up and performs a recovery. Your fault thread for synchronizing with the alert thread is illustrated as follows:

```
/* Fault Recovery Thread Loop */
while ( 1 ) {
  assert( pthread_mutex_lock( &recoveryMutex ) == 0);
  while (workload < MAX_NORMAL_WORKLOAD) {
    pthread_cond_wait( &recoveryCond, &recoveryMutex );
  }
  /*————————*/
  /* Recovery Code. */
  /*————————*/
  assert( pthread_mutex_unlock( &recoveryMutex ) == 0);
}
```

This is the standard pattern when dealing with condition variables. You start by locking the mutex, entering `pthread_cond_wait`, and upon waking up from your condition, unlocking the mutex. The mutex must be locked first because upon entry to `pthread_cond_wait`, the mutex is automatically unlocked. When you return from `pthread_cond_wait`, the mutex has been reacquired, meaning that you need to unlock it afterward. The mutex is necessary here to handle race conditions that exist in this call sequence. To ensure that your condition is met, you loop around the `pthread_cond_wait`, and if the condition is not satisfied (in this case, your workload is normal), then you reenter the `pthread_cond_wait` call. Note that because the mutex is locked upon return from `pthread_cond_wait`, you don't need to call `pthread_mutex_lock` here.

Now take a look at the signal code. This is considerably simpler than that code necessary to wait for the condition. Two possibilities exist for signaling: sending a single signal or broadcasting to all waiting threads.

The first case is signaling one thread. In either case, you first lock the mutex before calling the `signal` function and then unlock when you're done. To signal one thread, you call the `pthread_cond_signal` function, as follows:

```
pthread_mutex_lock( &recoveryMutex );
pthread_cond_signal( &recoveryCond );
pthread_mutex_unlock( &recovery_Mutex );
```

After the mutex is unlocked, exactly one thread is signaled and allowed to execute. Each function returns zero on success or an error code. If your architecture supports multiple threads for recovery, you can instead use the `pthread_cond_broadcast`. This function wakes up all threads currently awaiting the condition. This is demonstrated as follows:

```
pthread_mutex_lock( &recoveryMutex );
pthread_cond_broadcast( &recoveryCond );
pthread_mutex_unlock( &recovery_Mutex );
```

After the mutex is unlocked, the series of threads is then permitted to perform recovery (though one by one because they're dependent upon the mutex).

The pthreads API also supports a version of timed-wait for a condition variable. This function, pthread_cond_timedwait, allows the caller to specify an absolute time representing when to give up and return to the caller. The return value is ETIMEDOUT, to indicate that the function returned because of a timeout rather than because of a successful return. The following code snippet illustrates its use:

```
struct timeval currentTime;
struct timespec  expireTime;
int ret;
...
assert( pthread_mutex_lock( &recoveryMutex ) == 0);
gettimeofday( &currentTime );
expireTime.tv_sec = currentTime.tv_sec + 1;
expireTime.tv_nsec = currentTime.tv_usec * 1000;
ret = 0;
while ((workload < MAX_NORMAL_WORKLOAD) && (ret != ETIMEDOUT) {
  ret = pthread_cond_timedwait( &recoveryCond, &recoveryMutex,
                &expireTime );
}
if (ret == ETIMEDOUT) {
  /* Timeout – perform timeout processing */
} else {
  /* Condition met – perform condition recovery processing */
}
assert( pthread_mutex_unlock( &recoveryMutex ) == 0);
```

The first item to note is the generation of a timeout. You use the gettimeofday function to get the current time and then add one second to it in the timespec structure. This is passed to pthread_cond_timedwait to identify the time at which you desire a timeout if the condition has not been met. In this case, which is very similar to the standard pthread_cond_wait example, you check in your loop that the pthread_cond_timedwait function has not returned ETIMEDOUT. If it has, you exit your loop and then check again to perform timeout processing. Otherwise, you perform your standard condition processing (recovery for this example) and then reacquire the mutex.

The final function to note here is pthread_cond_destroy. You simply pass the condition variable to the function, as follows:

```
pthread_mutex_destroy( &recoveryCond );
```

It's important to note that in the GNU/Linux implementation no resources are actually attached to the condition variable, so this function simply checks to see if any threads are currently pending on the condition variable.

Now it's time to look at a complete example that brings together all of the elements just discussed for condition variables. This example illustrates condition variables in the context of producers and consumers. You create a producer thread that creates work and then *N* consumer threads that operate on the (simulated) work.

The first listing (Listing 15.8) shows the `main` program. This listing is similar to the previous examples of creating and then joining threads, with a few changes. You create two types of threads in this listing. At lines 18–21, you create a number of consumer threads, and at line 24, you create a single producer thread. You will take a look at these shortly. After creation of the last thread, you join the producer thread (resulting in a suspend of the main application until it has completed). You then wait for the work to complete (as identified by a simple counter, `workCount`). You want to allow the consumer threads to complete their work, so you wait until this variable is zero, indicating that all work is consumed.

The block of code at lines 33–36 shows joins for the consumer threads, with one interesting change. In this example, the consumer threads never quit, so you cancel them here using the `pthread_cancel` function. This function has the following prototype:

```
int pthread_cancel( pthread_t thread );
```

This permits you to terminate another thread when you're done with it. In this example, you have produced the work that you need the consumers to work on, so you cancel each thread in turn (line 34). Finally, you destroy your condition variable and mutex at lines 37 and 38, respectively.

LISTING 15.8 Producer/Consumer Example Initialization and `main` (on the CD-ROM at `./source/ch15/ptcond.c`)

```
1:      #include <pthread.h>
2:      #include <stdio.h>
3:
4:      pthread_mutex_t cond_mutex = PTHREAD_MUTEX_INITIALIZER;
5:      pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
6:
7:      int workCount = 0;
8:
9:      #define MAX_CONSUMERS   10
10:
```

```
11:     int main()
12:     {
13:       int i;
14:       pthread_t consumers[MAX_CONSUMERS];
15:       pthread_t producer;
16:
17:       /* Spawn the consumer thread */
18:       for ( i = 0 ; i < MAX_CONSUMERS ; i++ ) {
19:         pthread_create( &consumers[i], NULL,
20:                           consumerThread, NULL );
21:       }
22:
23:       /* Spawn the single producer thread */
24:       pthread_create( &producer, NULL,
25:                         producerThread, NULL );
26:
27:       /* Wait for the producer thread */
28:       pthread_join( producer, NULL );
29:
30:       while ((workCount > 0));
31:
32:       /* Cancel and join the consumer threads */
33:       for ( i = 0 ; i < MAX_CONSUMERS ; i++ ) {
34:         pthread_cancel( consumers[i] );
35:       }
36:
37:       pthread_mutex_destroy( &cond_mutex );
38:       pthread_cond_destroy( &condition );
39:
40:       return 0;
41:     }
```

Next, you can take a look at the producer thread function (Listing 15.9). The purpose of the producer thread is to produce work, simulated by incrementing the `workCount` variable. A nonzero `workCount` indicates that work is available to do. You loop for a number of times to create work, as is shown at lines 8–22. As shown in the condition variable sample, you first lock your mutex at line 10 and then create work to do (increment `workCount`). You then notify the awaiting consumer (worker) threads at line 14 using the `pthread_cond_broadcast` function. This notifies any awaiting consumer threads that work is now available to do. Next, at line 15, you unlock the mutex, allowing the consumer threads to lock the mutex and perform their work.

At lines 20–22, you simply do some busy work to allow the kernel to schedule another task (thereby avoiding synchronous behavior, for illustration purposes).

When all of the work has been produced, you permit the producer thread to exit (which is joined in the `main` function at line 28 of Listing 15.8).

LISTING 15.9 Producer Thread Example for Condition Variables (on the CD-ROM at `./source/ch15/ptcond.c`)

```
1:      void *producerThread( void *arg )
2:      {
3:        int i, j, ret;
4:        double result=0.0;
5:
6:        printf("Producer started\n");
7:
8:        for ( i = 0 ; i < 30 ; i++ ) {
9:
10:         ret = pthread_mutex_lock( &cond_mutex );
11:         if (ret == 0) {
12:           printf( "Producer: Creating work (%d)\n", workCount );
13:           workCount++;
14:           pthread_cond_broadcast( &condition );
15:           pthread_mutex_unlock( &cond_mutex );
16:         } else {
17:           assert(0);
18:         }
19:
20:         for ( j = 0 ; j < 60000 ; j++ ) {
21:           result = result + (double)random();
22:         }
23:
24:       }
25:
26:       printf("Producer finished\n");
27:
28:       pthread_exit( NULL );
29:     }
```

Now it's time to look at the consumer thread (see Listing 15.10). Your first task is to detach yourself (line 5), because you won't ever join with the creating thread. Then you go into your work loop (lines 9–22) to process the workload. You first lock the condition mutex at line 11 and then wait for the condition to occur at line 12. You then check to make sure that the condition is true (that work exists to do)

at line 14. Note that because you're broadcasting to threads, you might not have work to do for every thread, so you test before you assume that work is available.

After you've completed your work (in this case, simply decrementing the work count at line 15), you release the mutex at line 19 and wait again for work at line 11. Note that because you cancel your thread, you never see the `printf` at line 23, nor do you exit the thread at line 25. The `pthread_cancel` function terminates the thread so that the thread does not terminate normally.

LISTING 15.10 Consumer Thread Example for Condition Variables (on the CD-ROM at `./source/ch15/ptcond.c`)

```
1:        void *consumerThread( void *arg )
2:        {
3:          int ret;
4:
5:          pthread_detach( pthread_self() );
6:
7:          printf( "Consumer %x: Started\n", pthread_self() );
8:
9:          while( 1 ) {
10:
11:           assert( pthread_mutex_lock( &cond_mutex ) == 0);
12:           assert( pthread_cond_wait( &condition, &cond_mutex )
              == 0 );
13:
14:           if (workCount) {
15:             workCount--;
16:             printf( "Consumer %x: Performed work (%d)\n",
17:                       pthread_self(), workCount );
18:           }
19:           assert( pthread_mutex_unlock( &cond_mutex ) == 0);
20:
21:         }
22:
23:         printf( "Consumer %x: Finished\n", pthread_self() );
24:
25:         pthread_exit( NULL );
26:       }
```

Now take a look at this application in action. For brevity, this example shows only the first 30 lines emitted, but this gives you a good indication of how the application behaves (see Listing 15.11). You can see the consumer threads starting up, the producer starting, and then work being created and consumed in turn.

LISTING 15.11 Application Output for Condition Variable Application

```
$ ./ptcond
Consumer 4082cd40: Started
Consumer 4102ccc0: Started
Consumer 4182cc40: Started
Consumer 42932bc0: Started
Consumer 43132b40: Started
Consumer 43932ac0: Started
Consumer 44132a40: Started
Consumer 449329c0: Started
Consumer 45132940: Started
Consumer 459328c0: Started
Producer started
Producer: Creating work (0)
Producer: Creating work (1)
Consumer 4082cd40: Performed work (1)
Consumer 4102ccc0: Performed work (0)
Producer: Creating work (0)
Consumer 4082cd40: Performed work (0)
Producer: Creating work (0)
Producer: Creating work (1)
Producer: Creating work (2)
Producer: Creating work (3)
Producer: Creating work (4)
Producer: Creating work (5)
Consumer 4082cd40: Performed work (5)
Consumer 4102ccc0: Performed work (4)
Consumer 4182cc40: Performed work (3)
Consumer 42932bc0: Performed work (2)
Consumer 43132b40: Performed work (1)
Consumer 43932ac0: Performed work (0)
Producer: Creating work (0)
```

The design of multithreaded applications follows a small number of patterns (or models). The master/servant model is common where a single master doles out work to a collection of servants. The pipeline model splits work up into stages where one or more threads make up each of the work phases.

Building Threaded Applications

Building pthread-based applications is very simple. All that's necessary is to specify the `pthreads` library during compilation as follows:

```
gcc -pthread threadapp.c -o threadapp –lpthread
```

This links your application with the `pthread` library, making the `pthread` functions available for use. Note also that you specify the `-pthread` option, which adds support for multithreading to the application (such as re-entrancy). The option also ensures that certain global system variables (such as `errno`) are provided on a per-thread basis.

One topic that's important to discuss in multithreaded applications is that of re-entrancy. Consider two threads, each of which uses the `strtok` function. This function uses an internal buffer for token processing of a string. This internal buffer can be used by only one user at a time, which is fine in the process world (forked processes), but in the thread world runs into problems. If each thread attempts to call `strtok`, then the internal buffer is corrupted, leading to undesirable (and unpredictable) behavior. To fix this, rather than using an internal buffer, you can use a thread-supplied buffer instead. This is exactly what happens with the thread-safe version of `strtok`, called `strtok_r`. The suffix `_r` indicates that the function is thread-safe.

Summary

Multithreaded application development is a powerful model for the development of high-performance software systems. GNU/Linux provides the POSIX `pthreads` API for a standard and portable programming model. This chapter explored the standard thread creation, termination, and synchronization functions. This includes the basic synchronization using a join, but also more advanced coordination using mutexes and condition variables. Finally, building `pthread` applications was investigated, along with some of the pitfalls (such as re-entrancy) and how to deal with them. The GNU/Linux 2.6 kernel (using NPTL) provides a closer POSIX implementation and more efficient IPC and kernel support than the prior LinuxThreads version provided.

References

[Drepper and Molnar03] Drepper, Ulrich and Molnar, Ingo. (2003) *The Native POSIX Thread Library for Linux*. Red Hat, Inc.

API Summary

```
#include <pthread.h>
int pthread_create( pthread_t *thread,
                    pthread_attr_t *attr,
                    void *(*start_routine)(void *), void *arg );
int pthread_exit( void *retval );
pthread_t pthread_self( void );
int pthread_join( pthread_t th, void **thread_return );
int pthread_detach( pthread_t th );
int pthread_mutex_lock( pthread_mutex_t *mutex );
int pthread_mutex_trylock( pthread_mutex_t *mutex );
int pthread_mutex_unlock( pthread_mutex_t *mutex );
int pthread_mutex_destroy( pthread_mutex_t *mutex );
int pthread_cond_wait( pthread_cond_t *cond,
                       pthread_mutex_t *mutex );
int pthread_cond_timedwait( pthread_cond_t *cond,
                            pthread_mutex_t *mutex,
                            const struct timespec *abstime );
int pthread_cond_signal( pthread_cond_t *cond );
int pthread_cond_broadcast( pthread_cond_t *cond );
int pthread_cancel( pthread_t thread );
```

16 IPC with Message Queues

In This Chapter

- Introduction to Message Queues
- Creating and Configuring Message Queues
- Creating Messages Suitable for Message Queues
- Sending and Receiving Messages
- Adjusting Message Queue Behavior
- The `ipcs` Utility

Introduction

The topic of interprocess communication is an important one because it allows you the ability to build systems out of numerous communicating asynchronous processes. This is beneficial because you can naturally segment the functionality of a large system into a number of distinct elements. Because GNU/Linux processes utilize independent memory spaces, a function in one process cannot call another in a different process. Message queues provide one means to permit communication and coordination between processes. This chapter reviews the message queue model (which conforms to the SystemV UNIX model), as well as explores some sample code that utilizes the message queue API.

Quick Overview of Message Queues

This chapter begins by taking a whirlwind tour of the POSIX-compliant message queue API. You will take a look at code examples that illustrate creating a message queue, configuring its size, sending and receiving a message, and then removing the message queue. After you have had a taste of the message queue API, you can dive in deeper in the following sections.

Using the message queue API requires that the function prototypes and symbols be available to the application. This is done by including the `msg.h` header file as follows:

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

You first introduce a common header file that defines some common information needed for the writer and reader of the message (see Listing 16.1). You define your system-wide queue ID (111) at line 3. This isn't the best way to define the queue, but later on you will see a way to define a unique system ID. Lines 5–10 define your message type, with the required long type at the head of the structure (line 6).

LISTING 16.1 Common Header File Used by the Sample Applications (on the CD-ROM at `./source/ch16/common.h`)

```
1:      #define MAX_LINE        80
2:
3:      #define MY_MQ_ID        111
4:
5:      typedef struct {
6:              long  type;             // Msg Type (> 0)
7:              float fval;             // User Message
8:              unsigned int uival;     // User Message
9:              char strval[MAX_LINE+1]; // User Message
10:     } MY_TYPE_T;
```

Creating a Message Queue

To create a message queue, you use the `msgget` API function. This function takes a message queue ID (a unique identifier, or key, within a given host) and another argument identifying the message flags. The flags in the queue create example (see Listing 16.2) specify that a queue is to be created (`IPC_CREAT`) as well as the access permissions of the message queue (read/write permission for system, user, and group).

The result of the `msgget` *function is a handle, which is similar to a file descriptor, pointing to the message queue with the particular ID.*

LISTING 16.2 Creating a Message Queue with `msgget` (on the CD-ROM at `./source/ch16/mqcreate.c`)

```
1:        #include <stdio.h>
2:        #include <sys/msg.h>
3:        #include "common.h"
4:
5:        int main()
6:        {
7:          int msgid;
8:
9:          /* Create the message queue with the id MY_MQ_ID */
10:         msgid = msgget( MY_MQ_ID, 0666 | IPC_CREAT );
11:
12:         if (msgid >= 0) {
13:
14:           printf( "Created a Message Queue %d\n", msgid );
15:
16:         }
17:
18:         return 0;
19:       }
```

Upon creating the message queue at line 10 (in Listing 16.2), you get a return integer that represents a handle for the message queue. This message queue ID can be used in subsequent message queue calls to send or receive messages.

Configuring a Message Queue

When you create a message queue, some of the details of the process that created the queue are automatically stored with it (for permissions) as well as a default queue size in bytes (16 KB). You can adjust this size using the `msgctl` API function. Listing 16.3 illustrates reading the defaults for the message queue, adjusting the queue size, and then configuring the queue with the new set.

LISTING 16.3 Configuring a Message Queue with `msgctl` (on the CD-ROM at `./source/ch16/mqconf.c`)

```
1:        #include <stdio.h>
2:        #include <sys/msg.h>
3:        #include "common.h"
4:
5:        int main()
6:        {
```

```
7:          int msgid, ret;
8:          struct msqid_ds buf;
9:
10:         /* Get the message queue for the id MY_MQ_ID */
11:         msgid = msgget( MY_MQ_ID, 0 );
12:
13:         /* Check successful completion of msgget */
14:         if (msgid >= 0) {
15:
16:           ret = msgctl( msgid, IPC_STAT, &buf );
17:
18:           buf.msg_qbytes = 4096;
19:
20:           ret = msgctl( msgid, IPC_SET, &buf );
21:
22:           if (ret == 0) {
23:
24:             printf( "Size successfully changed for queue
                %d.\n", msgid );
25:
26:           }
27:
28:         }
29:
30:         return 0;
31:       }
```

First, at line 11, you get the message queue ID using `msgget`. Note that the second argument here is zero because you're not creating the message queue, just retrieving its ID. You use this at line 16 to get the current queue data structure using the `IPC_STAT` command and your local buffer (for which the function fills in the defaults). You adjust the queue size at line 18 (by modifying the `msg_qbytes` field of the structure) and then write it back at line 20 using the `msgctl` API function with the `IPC_SET` command. You can also modify the user or group ID of the message queue or its mode. This chapter discusses these capabilities in more detail later.

Writing a Message to a Message Queue

Now take a look at actually sending a message through a message queue. A message within the context of a message queue has only one constraint. The object that's being sent must include a long variable at its head that defines the message type. This is discussed more later in the chapter, but it's simply a way to differentiate

messages that have been loaded onto a queue (and also how those messages can be read from the queue). The general structure for a message is as follows:

```
typedef struct {
  long type;
  char message[80];
} MSG_TYPE_T;
```

In this example (`MSG_TYPE_T`), you have your required `long` at the head of the message, followed by the user-defined message (in this case, a string of 80 characters).

To send a message to a message queue (see Listing 16.4), you use the `msgsnd` API function. Following a similar pattern to the previous examples, you first identify the message queue ID using the `msgget` API function (line 11). After this is known, you can send a message to it. Next, you initialize your message at lines 16–19. This includes specifying the mandatory type (must be greater than zero), a floating-point value (`fval`) and unsigned int value (`uival`), and a character string (`strval`). To send this message, you call the `msgsnd` API function. The arguments for this function are the message queue ID (`qid`), your message (a reference to `myObject`), the size of the message you're sending (the size of `MY_TYPE_T`), and finally a set of message flags (for now, `0`, but you'll investigate more later in the chapter).

LISTING 16.4 Sending a Message with `msgsnd` (on the CD-ROM at `./source/ch16/mqsend.c`)

```
1:        #include <sys/msg.h>
2:        #include <stdio.h>
3:        #include "common.h"
4:
5:        int main()
6:        {
7:          MY_TYPE_T myObject;
8:          int qid, ret;
9:
10:         /* Get the queue ID for the existing queue */
11:         qid = msgget( MY_MQ_ID, 0 );
12:
13:         if (qid >= 0) {
14:
15:           /* Create our message with a message queue type of 1 */
16:           myObject.type = 1L;
17:           myObject.fval = 128.256;
18:           myObject.uival = 512;
```

```
19:           strncpy( myObject.strval, "This is a test.\n",
              MAX_LINE );
20:
21:           /* Send the message to the queue defined by the queue
              ID */
22:           ret = msgsnd( qid, (struct msgbuf *)&myObject,
23:                         sizeof(MY_TYPE_T), 0 );
24:
25:           if (ret != -1) {
26:
27:             printf( "Message successfully sent to queue %d\n",
                qid );
28:
29:           }
30:
31:         }
32:
33:         return 0;
34:       }
```

That's it! This message is now held in the message queue, and at any point in the future, it can be read (and consumed) by the same or a different process.

Reading a Message from a Message Queue

Now that you have a message in your message queue, you can look at reading that message and displaying its contents (see Listing 16.5). You retrieve the ID of the message queue using `msgget` at line 12 and then use this as the target queue from which to read using the `msgrcv` API function at lines 16–17. The arguments to `msgrcv` are first the message queue ID (`qid`), the message buffer into which your message is to be copied (`myObject`), the size of the object (`sizeof(MY_TYPE_T)`), the message type that you want to read (`1`), and the message flags (`0`). Note that when you sent your message (in Listing 16.4), you specified our message type as `1`. You use this same value here to read the message from the queue. Had you used another value, the message would not have been read. More on this subject in the "`msgrcv`" section later in this chapter.

LISTING 16.5 Reading a Message with `msgrcv` (on the CD-ROM at `./source/ch16/mqrecv.c`)

```
1:        #include <sys/msg.h>
2:        #include <stdio.h>
```

```
3:        #include "common.h"
4:
5:        int main()
6:        {
7:          MY_TYPE_T myObject;
8:          int qid, ret;
9:
10:         qid = msgget( MY_MQ_ID, 0 );
11:
12:         if (qid >= 0) {
13:
14:           ret = msgrcv( qid, (struct msgbuf *)&myObject,
15:                         sizeof(MY_TYPE_T), 1, 0 );
16:
17:           if (ret != -1) {
18:
19:             printf( "Message Type: %ld\n", myObject.type );
20:             printf( "Float Value:  %f\n",  myObject.fval );
21:             printf( "Uint Value:   %d\n",  myObject.uival );
22:             printf( "String Value: %s\n",  myObject.strval );
23:
24:           }
25:
26:         }
27:
28:         return 0;
29:       }
```

The final step in your application in Listing 16.5 is to emit the message read from the message queue. You use your object type to access the fields in the structure and simply emit them with `printf`.

Removing a Message Queue

As a final step, take a look at how you can remove a message queue (and any messages that might be held on it). You use the `msgctl` API function for this purpose with the command of `IPC_RMID`. This is illustrated in Listing 16.6.

LISTING 16.6 Removing a Message Queue with `msgctl` (on the CD-ROM at `./source/ch16/mqdel.c`)

```
1:        #include <stdio.h>
2:        #include <sys/msg.h>
3:        #include "common.h"
```

```
4:
5:        int main()
6:        {
7:          int   msgid, ret;
8:
9:          msgid = msgget( MY_MQ_ID, 0 );
10:
11:         if (msgid >= 0) {
12:
13:           /* Remove the message queue */
14:           ret = msgctl( msgid, IPC_RMID, NULL );
15:
16:           if (ret != -1) {
17:
18:             printf( "Queue %d successfully removed.\n", msgid );
19:
20:           }
21:
22:         }
23:
24:         return 0;
25:       }
```

In Listing 16.6, you first identify the message queue ID using `msgget` and then use this with `msgctl` to remove the message queue. Any messages that happen to be on the message queue when `msgctl` is called are immediately removed.

That does it for our whirlwind tour. The next section digs deeper into the message queue API and looks at some of the behaviors of the commands that weren't covered already.

The Message Queue API

Now it's time to dig into the message queue API and investigate each of the functions in more detail. For a quick review, Table 16.1 provides the API functions and their purposes.

Figure 16.1 graphically illustrates the message queue API functions and their relationship in the process.

The next sections address these functions in detail, identifying each of the uses with descriptive examples.

TABLE 16.1 Message Queue API Functions and Uses

API Function	Uses
`msgget`	Create a new message queue.
	Get a message queue ID.
`msgsnd`	Send a message to a message queue.
`msgrcv`	Receive a message from a message queue.
`msgctl`	Get the info about a message queue.
	Set the info for a message queue.
	Remove a message queue.

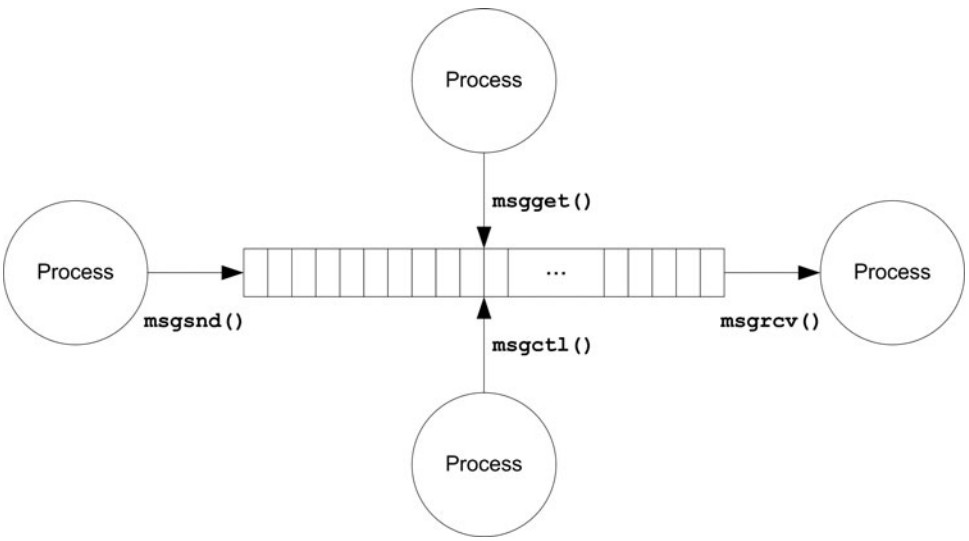

FIGURE 16.1 Message queue API functions.

msgget

The `msgget` API function serves two basic roles: to create a message queue or to get the identifier of a message queue that already exists. The result of the `msgget` function (unless an error occurs) is the message queue identifier (used by all other message queue API functions). The prototype for the `msgget` function is defined as follows:

```
int msgget( key_t key, int msgflag );
```

The key argument defines a system-wide identifier that uniquely identifies a message queue. key must be a nonzero value or the special symbol IPC_PRIVATE. The IPC_PRIVATE variable simply tells the msgget function that no key is provided and to simply make one up. The problem with this is that no other process can then find the message queue, but for local message queues (private queues), this method works fine.

The msgflag argument allows the user to specify two distinct parameters: a command and an optional set of access permissions. Permissions replicate those found as modes for the file creation functions (see Table 16.2). The command can take three forms. The first is simply IPC_CREAT, which instructs msgget to create a new message queue (or return the ID for the queue if it already exists). The second includes two commands (IPC_CREAT | IPC_EXCL), which request that the message queue be created, but if it already exists, the API function should fail and return an error response (EEXIST). The third possible command argument is simply 0. This form tells msgget that the message queue identifier for an existing queue is being requested.

TABLE 16.2 Message Queue Permissions for the msgget msgflag Argument

Symbol	Value	Meaning
S_IRUSR	0400	User has read permission.
S_IWUSR	0200	User has write permission.
S_IRGRP	0040	Group has read permission.
S_IWGRP	0020	Group has write permission.
S_IROTH	0004	Other has read permission.
S_IWOTH	0002	Other has write permission.

Now it's time to take a look at a few examples of the msgget function to create message queues or access existing ones. Assume in the following code snippets that msgid is an int value (int msgid). You can start by creating a private queue (no key is provided).

```
msgid = msgget( IPC_PRIVATE, IPC_CREAT | 0666 );
```

If the msgget API function fails, -1 is returned with the actual error value provided within the process's errno variable.

Now say that you want to create a message queue with a key value of `0x111`. You also want to know if the queue already exists, so you use the `IPC_EXCL` in this example:

```
// Create a new message queue
msgid = msgget( 0x111, IPC_CREAT | IPC_EXCL | 0666 );
 if (msgid == -1) {
   printf("Queue already exists...\n");
 } else {
   printf("Queue created...\n");
 }
```

An interesting question you've probably asked yourself now is how can you coordinate the creation of queues using IDs that might not be unique? What happens if someone already used the `0x111` key? Luckily, you have a way to create keys in a system-wide fashion that ensures uniqueness. The `ftok` system function provides the means to create system-wide unique keys using a file in the filesystem and a number. As the file (and its path) is by default unique in the filesystem, a unique key can be created easily. Take a look at an example of using `ftok` to create a unique key. Assume that the file with path `/home/mtj/queues/myqueue` exists.

```
key_t myKey;
int   msgid;
// Create a key based upon the defined path and number
myKey = ftok( "/home/mtj/queues/myqueue", 0 );
msgid = msgget( myKey, IPC_CREAT | 0666 );
```

This creates a key for this path and number. Each time `ftok` is called with this path and number, the same key is generated. Therefore, it provides a useful way to generate a key based upon a file in the filesystem.

One last example is getting the message queue ID of an existing message queue. The only difference in this example is that you provide no command, only the key:

```
msgid = msgget( 0x111, 0 );
if (msgid == -1) {
  printf("Queue doesn't exist...\n");
}
```

The `msgflags` (second argument to `msgget`) is zero in this case, which indicates to this API function that an existing message queue is being sought.

One final note on message queues relates to the default settings that are given to a message queue when it is created. The configuration of the message queue is noted in the parameters shown in Table 16.3. Note that you have no way to change these defaults within `msgget`. In the next section, you take look at some of the parameters that can be changed and their effects.

TABLE 16.3 Message Queue Configuration and Defaults in `msgget`

Parameter	Default Value
`msg_perm.cuid`	Effective user ID of the calling process (creator)
`msg_perm.uid`	Effective user ID of the calling process (owner)
`msg_perm.cgid`	Effective group ID of the calling process (creator)
`msg_perm.gid`	Effective group ID of the calling process (owner)
`msg_perm.mode`	Permissions (lower 9 bits of `msgflag`)
`msg_qnum`	0 (Number of messages in the queue)
`msg_lspid`	0 (Process ID of last `msgsnd`)
`msg_lrpid`	0 (Process ID of last `msgrcv`)
`msg_stime`	0 (last `msgsnd` time)
`msg_rtime`	0 (Last `msgrcv` time)
`msg_ctime`	Current time (last change time)
`msg_qbytes`	Queue size in bytes (system limit)—(16 KB)

The user can override the `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode`, and `msg_qbytes` directly. More on this topic in the next section.

`msgctl`

The `msgctl` API function provides three distinct features for message queues. The first is the ability to read the current set of message queue defaults (via the `IPC_STAT` command). The second is the ability to modify a subset of the defaults (via `IPC_SET`). Finally, the ability to remove a message queue is provided (via `IPC_RMID`). The `msgctl` prototype function is defined as follows:

```
#include <sys/msg.h>
int msgctl( int msgid, int cmd, struct msqid_ds *buf );
```

You can start by looking at `msgctl` as a means to remove a message queue from the system. This is the simplest use of `msgctl` and can be demonstrated very easily. To remove a message queue, you need only the message queue identifier that is returned by `msgctl`.

Whereas a system-wide unique key is required to create a message queue, only the message queue ID (returned from `msgget`*) is required to configure a queue, send a message from a queue, receive a message from a queue, or remove a queue.*

Now take a look at an example of message queue removal using `msgctl`. Whenever the shared resource is no longer needed, the application should remove it. You first get the message queue identifier using `msgget` and then use this ID in your call to `msgctl`.

```
int msgid, ret;
...
msgid = msgget( QUEUE_KEY, 0 );
if (msgid != -1) {
  ret = msgctl( msgid, IPC_RMID, NULL );
      if (ret == 0) {
    // queue was successfully removed.
  }
}
```

If any processes are currently blocked on a `msgsnd` or `msgrcv` API function, those functions return with an error (`-1`) with the `errno` process variable set to `EIDRM`. The process performing the `IPC_RMID` must have adequate permissions to remove the message queue. If permissions do not allow the removal, an error return is generated with an `errno` variable set to `EPERM`.

Now take a look at `IPC_STAT` (read configuration) and `IPC_SET` (write configuration) commands together for `msgctl`. In the previous section, you identified the range of parameters that make up the configuration and status parameters. Now it's time to look at which of the parameters can be directly manipulated or used by the application developer. Table 16.4 lists the parameters that can be updated after a message queue has been created.

Changing these parameters is a very simple process. The process should be that the application first reads the current set of parameters (via `IPC_STAT`) and then modifies the parameters of interest before writing them back out (via `IPC_SET`). See Listing 16.7 for an illustration of this process.

TABLE 16.4 Message Queue Parameters That Can Be Updated

Parameter	Description
`msg_perm.uid`	Message queue user owner
`msg_perm.gid`	Message queue group owner
`msg_perm.mode`	Permissions (see Table 16.2)
`msg_qbytes`	Size of message queue in bytes

LISTING 16.7 Setting All Possible Options in `msgctl` (on the CD-ROM at `./source/ch16/mqrdset.c`)

```
#include <stdio.h>
#include <sys/msg.h>
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include "common.h"

int main()
{
  int msgid, ret;
  struct msqid_ds buf;

  /* Get the message queue for the id MY_MQ_ID */
  msgid = msgget( MY_MQ_ID, 0 );

  /* Check successful completion of msgget */
  if (msgid >= 0) {

    ret = msgctl( msgid, IPC_STAT, &buf );

    buf.msg_perm.uid = geteuid();
    buf.msg_perm.gid = getegid();
    buf.msg_perm.mode = 0644;
    buf.msg_qbytes = 4096;

    ret = msgctl( msgid, IPC_SET, &buf );

    if (ret == 0) {

```

```
30:             printf( "Parameters successfully changed.\n");
31:
32:           } else {
33:
34:             printf( "Error %d\n", errno );
35:
36:           }
37:
38:         }
39:
40:         return 0;
41:       }
```

At line 14, you get your message queue identifier, and then you use this at line 19 to retrieve the current set of parameters. At line 21, you set the `msg_perm.uid` (effective user ID) with the current effective user ID using the `geteuid()` function. Similarly, you set the `msg_perm.gid` (effective group ID) using the `getegid()` function at line 22. At line 23 you set the mode, and at line 24 you set the maximum queue size (in bytes). In this case you set it to 4 KB. You now take this structure and set the parameters for the current message queue using the `msgctl` API function. This is done with the `IPC_SET` command in `msgctl`.

When setting the `msg_perm.mode` *(permissions), you need to know that this is traditionally defined as an octal value. Note at line 23 of Listing 16.7 that a leading zero is shown, indicating that the value is octal. If, for example, a decimal value of 666 were provided instead of octal 0666, permissions would be invalid, and therefore undesirable behavior would result. For this reason, it can be beneficial to use the symbols as shown in Table 16.2.*

You can also use the `msgctl` API function to identify certain message queue-specific parameters, such as the number of messages currently on the message queue. Listing 16.8 illustrates the collection and printing of the accessible parameters.

LISTING 16.8 Reading Current Message Queue Settings (on the CD-ROM at `./source/ch16/mqstats.c`)

```
1:      #include <stdio.h>
2:      #include <sys/msg.h>
3:      #include <unistd.h>
4:      #include <sys/types.h>
5:      #include <time.h>
6:      #include "common.h"
7:
```

```
8:        int main()
9:        {
10:         int msgid, ret;
11:         struct msqid_ds buf;
12:
13:         /* Get the message queue for the id MY_MQ_ID */
14:         msgid = msgget( MY_MQ_ID, 0 );
15:
16:         /* Check successful completion of msgget */
17:         if (msgid >= 0) {
18:
19:           ret = msgctl( msgid, IPC_STAT, &buf );
20:
21:           if (ret == 0) {
22:
23:             printf( "Number of messages queued: %ld\n",
24:                      buf.msg_qnum );
25:             printf( "Number of bytes on queue : %ld\n",
26:                      buf.msg_cbytes );
27:             printf( "Limit of bytes on queue  : %ld\n",
28:                      buf.msg_qbytes );
29:
30:             printf( "Last message writer (pid): %d\n",
31:                      buf.msg_lspid );
32:             printf( "Last message reader (pid): %d\n",
33:                      buf.msg_lrpid );
34:
35:             printf( “Last change time         : %s”,
36:                      ctime(&buf.msg_ctime) );
37:
38:             if (buf.msg_stime) {
39:               printf( "Last msgsnd time         : %s”,
40:                        ctime(&buf.msg_stime) );
41:             }
42:             if (buf.msg_rtime) {
43:               printf( "Last msgrcv time         : %s”,
44:                        ctime(&buf.msg_rtime) );
45:             }
46:
47:           }
48:
49:         }
50:
51:         return 0;
52:       }
```

Listing 16.8 begins as most other message queue examples, with the collection of the message queue ID from `msgget`. After you have your ID, you use this to collect the message queue structure using `msgctl` and the command `IPC_STAT`. You pass in a reference to the `msqid_ds` structure, which is filled in by the `msgctl` API function. You then emit the information collected in lines 23–45.

At lines 23–24, you emit the number of messages that are currently enqueued on the message queue (`msg_qnum`). The current total number of bytes that are enqueued is identified by `msg_cbytes` (lines 25–26), and the maximum number of bytes that can be enqueued is defined by `msg_qbytes` (lines 27–28).

You can also identify the last reader and writer process IDs (lines 30–33). These refer to the effective process ID of the calling process that called `msgrcv` or `msgsnd`.

The `msg_ctime` element refers to the last time the message queue was changed (or when it was created). It's in standard `time_t` format, so you pass `msg_ctime` to `ctime` to grab the ASCII text version of the calendar date and time. You do the same for `msg_stime` (last `msgsnd` time) and `msg_rtime` (last `msgrcv` time). Note that in the case of `msg_stime` and `msg_rtime`, you emit the string dates only if their values are nonzero. If the values are zero, no `msgrcv` or `msgsnd` API functions have been called.

msgsnd

The `msgsnd` API function allows a process to send a message to a queue. As you saw in the introduction, the message is purely user-defined except that the first element in the message must be a `long` word for the type field. The function prototype for the `msgsnd` function is defined as follows:

```
int msgsnd( int msgid, struct msgbuf *msgp, size_t msgsz,
                  int msgflg );
```

The `msgid` argument is the message queue ID (returned from the `msgget` function). The `msgbuf` represents the message to be sent; at a minimum it is a `long` value representing the message type. The `msgsz` argument identifies the size of the `msgbuf` passed in to `msgsend`, in bytes. Finally, the `msgflag` argument allows you to alter the behavior of the `msgsnd` API function.

The `msgsnd` function has some default behavior that you should consider. If insufficient room exists on the message queue to write the message, the process is blocked until sufficient room exists. Otherwise, if room exists, the call succeeds immediately with a zero return to the caller.

Because you have already looked at some of the standard uses of `msgsnd`, here's your chance to look at some of the more specialized cases. The blocking behavior is desirable in most cases because it can be the most efficient. In some cases, you

might want to try to send a message, and if you're unable (because of the insufficient space on the message queue), do something else. Take a look at this example in the following code snippet:

```
ret = msgsnd( msgid, (struct msgbuf *)&myMessage,
                    sizeof(myMessage), IPC_NOWAIT );
if (ret == 0) {
  // Message was successfully enqueued
} else {
  if (errno == EAGAIN) {
    // Insufficient space, do something else...
  }
}
```

The `IPC_NOWAIT` symbol (passed in as the `msgflags`) tells the `msgsnd` API function that if insufficient space exists, don't block but instead return immediately. You know this because an error was returned (indicated by the `-1` return value), and the `errno` variable was set to `EAGAIN`. Otherwise, with a zero return, the message was successfully enqueued on the message queue for the receiver.

While a message queue should not be deleted as long as processes pend on `msgsnd`, a special error return surfaces when this occurs. If a process is currently blocked on a `msgsnd` and the message queue is deleted, then a `-1` value is returned with an `errno` value set to `EIDRM`.

One final item to note on `msgsnd` involves the parameters that are modified when the `msgsnd` API call finishes. Table 16.3 lists the entire structure, but the items modified after successful completion of the `msgsnd` API function are listed in Table 16.5.

TABLE 16.5 Structure Updates after Successful `msgsnd` Completion

Parameter	Update
`msg_lspid`	Set to the process ID of the process that called `msgsnd`
`msg_qnum`	Incremented by one
`msg_stime`	Set to the current time

Note that the `msg_stime` *is the time that the message was enqueued and not the time that the* `msgsnd` *API function was called. This can be important if the* `msgsnd` *function blocks (because of a full message queue).*

msgrcv

Now you can focus on the last function in the message queue API. The `msgrcv` API function provides the means to read a message from the queue. The user provides a message buffer (filled in within `msgrcv`) and the message type of interest. The function prototype for `msgrcv` is defined as follows:

```
ssize_t msgrcv( int msgid, struct msgbuf *msgp, size_t msgsz,
                long msgtyp, int msgflg );
```

The arguments passed to `msgrcv` include the `msgid` (message queue identifier received from `msgget`), a reference to a message buffer (`msgp`), the size of the buffer (`msgsz`), the message type of interest (`msgtyp`), and finally a set of flags (`msgflag`). The first three arguments are self-explanatory, so this section concentrates on the latter two: `msgtyp` and `msgflag`.

The `msgtyp` argument (message type) specifies to `msgrcv` the messages to be received. Each message within the queue contains a message type. The `msgtyp` argument to `msgrcv` defines that only those types of messages are sought. If no messages of that type are found, the calling process blocks until a message of the desired type is enqueued. Otherwise, the first message of the given type is returned to the caller. The caller could provide a zero as the `msgtyp`, which tells `msgrcv` to ignore the message type and return the first message on the queue. One exception to the message type request is discussed with `msgflg`.

The `msgflg` argument allows the caller to alter the default behavior of the `msgrcv` API function. As with `msgsnd`, you can instruct `msgrcv` not to block if no messages are waiting on the queue. This is done also with the `IPC_NOWAIT` flag. the previous paragraph discussed the use of `msgtyp` with a zero and nonzero value, but what if you were interested in any flag except a certain one? This can be accomplished by setting `msgtyp` with the undesired message type and setting the flag `MSG_EXCEPT` within `msgflg`. Finally, the use of flag `MSG_NOERROR` instructs `msgrcv` to ignore the size check of the incoming message and the available buffer passed from the user and simply truncate the message if the user buffer isn't large enough. All of the options for `msgtyp` are described in Table 16.6, and options for `msgflg` are shown in Table 16.7.

TABLE 16.6 `msgtyp` Arguments for `msgrcv`

msgtyp	Description
0	Read the first message available on the queue.
>0	If the `msgflg MSG_EXCEPT` is set, read the first message on the queue not equal to the `msgtyp`. Otherwise, if `MSG_EXCEPT` is not set, read the first message on the queue with the defined `msgtyp`.
<0	The first message on the queue that is less than or equal to the absolute value of `msgtyp` is returned.

TABLE 16.7 `msgflg` Arguments for `msgrcv`

Flag	Description
`IPC_NOWAIT`	Return immediately if no messages awaiting are of the given `msgtyp` (no blocking).
`MSG_EXCEPT`	Return first message available other than `msgtyp`.
`MSG_NOERROR`	Truncate the message if user buffer isn't of sufficient size.

When a message is read from the queue, the internal structure representing the queue is automatically updated as shown in Table 16.8.

TABLE 16.8 Structure Updates after Successful `msgsnd` Completion

Parameter	Update
`msg_lrpid`	Set to the process ID of process calling **`msgrcv`**
`msg_qnum`	Decremented by one
`msg_rtime`	Set to the current time

Note that `msg_rtime` *is the time that the message was dequeued and not the time that the* `msgrcv` *API function was called. This can be important if the* `msgrcv` *function blocks (because of an empty message queue).*

Now take a look at some examples to illustrate msgrcv and the use of msgtyp and msgflg options. The most common use of msgrcv is to read the next available message from the queue:

```
ret = msgrcv( msgid, (struct msgbuf *)&buf, sizeof(buf), 0, 0 );
if (ret != -1) {
  printf("Message of type %ld received\n", *(long *)&buf );
}
```

Note that you check specifically for a return value that's not -1. You do this because msgrcv actually returns the number of bytes read. If the return is -1, errno contains the error that occurred.

If you desire not to block on the call, you can do this very simply as follows:

```
ret = msgrcv( msgid, (struct msgbuf *)&buf, sizeof(buf),
                    0, IPC_NOWAIT );
if (ret != -1) {
  printf("Message of type %ld received\n", *(long *)&buf );
} else if (errno == EAGAIN) {
  printf("Message unavailable\n");
}
```

With the presence of an error return from msgrcv and errno set to EAGAIN, it's understood that no messages are available for read. This isn't actually an error, just an indication that no messages are available in the nonblocking scenario.

Message queues permit multiple writers and readers to the same queue. These could be the same process, but very likely each is a different process. Say that you have a process that manages only a certain type of message. You identify this particular message by its message type. In the next example, you see a snippet from a process whose job it is to manage only messages of type 5.

```
ret = msgrcv( msgid, (struct msgbuf *)&buf, sizeof(buf), 5, 0 );
```

Any message sent of type 5 is received by the process executing this code snippet. To manage all other message types (other than 5), you can use the MSG_EXCEPT flag to receive these. Take for example:

```
ret = msgrcv( msgid, (struct msgbuf *)&buf, sizeof(buf),
                5, MSG_EXCEPT );
```

Any message received on the queue other than type 5 is read using this line. If only messages of type 5 are available, this function blocks until a message not of type 5 is enqueued.

One final note on msgrcv is what happens if a process is blocked on a queue that is removed. The removal is permitted to occur, and the process blocked on the queue receives an error return with the errno set to EIDRM (as with blocked msgsnd calls). It's therefore important to fully recognize the error returns that are possible.

User Utilities

GNU/Linux provides the ipcs command to explore IPC assets from the command line. The ipcs utility provides information on message queues as well as semaphores and shared memory segments. This section looks at its use for message queues.

The general form of the ipcs utility for message queues is as follows:

```
# ipcs -q
```

This presents all of the message queues that are visible to the process. You can start by creating a message queue (as was done in Listing 16.1):

```
# ./mqcreate
Created a Message Queue 819200
# ipcs -q

—— Message Queues ——
key        msqid      owner      perms      used-bytes   messages
0x0000006f 819200     mtj        666        0            0
```

You see the newly created queue (key 0x6f, or decimal 111). If you send a message to the message queue (such as was illustrated with Listing 16.4), you see the following:

```
# ./mqsend
Message successfully sent to queue 819200
# ipcs -q

—— Message Queues ——
key        msqid      owner      perms      used-bytes   messages
0x0000006f 819200     mtj        666        96           1
```

You see now that a message is contained on the queue that occupies 96 bytes. You can also take a deeper look at the queue by specifying the message queue ID. This is done with `ipcs` using the `-i` option:

```
# ipcs -q -i 819200
Message Queue msqid=819200
uid=500 gid=500 cuid=500        cgid=500        mode=0666
cbytes=96       qbytes=16384    qnum=1  lspid=22309     lrpid=0
send_time=Sat Mar 27 18:59:34 2004
rcv_time=Not set
change_time=Sat Mar 27 18:58:43 2004
```

You're now able to review the structure representing the message queue (as defined in Table 16.3). The `ipcs` utility can be very useful to view snapshots of message queues for application debugging.

You can also delete queues from the command line using the `ipcrm` command. To delete your previously created message queue, you simply use the `ipcrm` command as follows:

```
$ ipcrm -q 819200
$
```

As with the message queue API functions, you pass the message queue ID as the indicator of the message queue to remove.

Summary

This chapter introduced the message queue API and its application of interprocess communication. It began with a whirlwind tour of the API and then detailed each of the functions, including the behavioral modifiers (`msgflg` arguments). Finally, it reviewed the `ipcs` utility and demonstrated its use as a debugging tool as well as the `ipcrm` command for removing message queues from the command line.

Message Queue APIs

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget( key_t key, int msgflg );
int msgctl( int msgid, int cmd, struct msqid_ds *buf );
int msgsnd( int msgid, structu msgbuf *msgp, size_t msgsz,
int msgflg );
size_t msgrcv( int msgid, struct msgbuf *msgp, size_t msgsz,
long msgtyp, int msgflg );
```

17 Synchronization with Semaphores

In This Chapter

- Introduction to GNU/Linux Semaphores
- Discussion of Binary and Counting Semaphores
- Creating and Configuring Semaphores
- Acquiring and Releasing Semaphores
- Single Semaphores or Semaphore Arrays
- The `ipcs` and `ipcrm` Utilities for Semaphores

Introduction

This chapter explores the topic of semaphores. GNU/Linux provides both binary and counting semaphores using the same POSIX-compliant API function set. It also investigates semaphores in GNU/Linux and their similarities with some of the other interprocess communication (IPC) mechanisms.

Semaphore Theory

First you need to go through a quick review of semaphore theory. A semaphore is nothing more than a variable that is protected. It provides a means to restrict access to a resource that is shared amongst two or more processes. Two operations are permitted, commonly called acquire and release. The acquire operation allows a process to take the semaphore, and if it has already been acquired, then the process

blocks until it's available. If a process has the semaphore, it can release it, which allows other processes to acquire it. The process of releasing a semaphore automatically wakes up the next process awaiting it on the acquire operation. Consider the simple example in Figure 17.1.

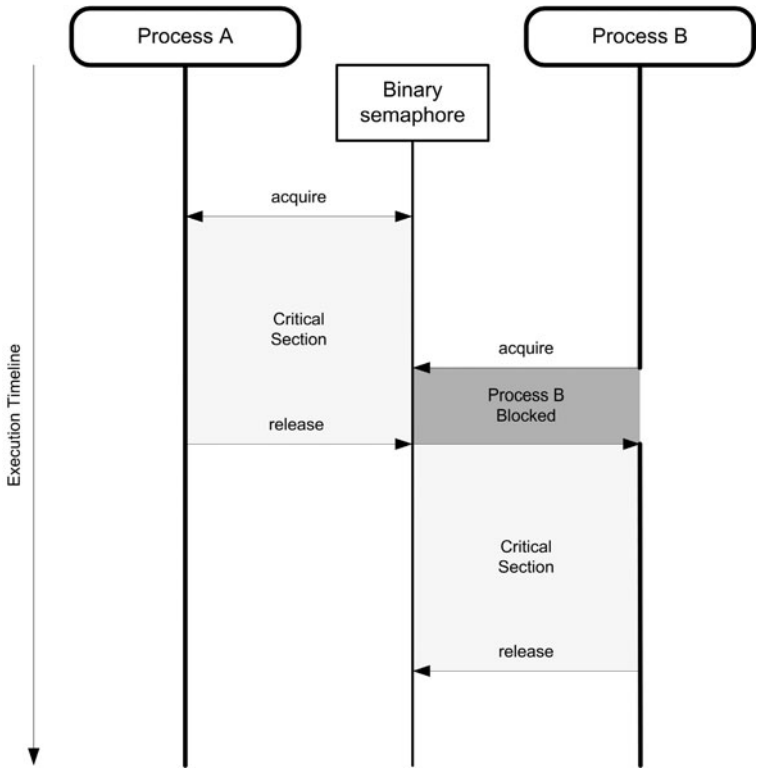

FIGURE 17.1 Simple binary semaphore example with two processes.

As shown in Figure 17.1, two processes are both vying for the single semaphore. Process A performs the acquire first and, therefore, is provided with the semaphore. The period in which the semaphore is owned by the process is commonly called a *critical section*. The critical section can be performed by only one process—hence the need for the coordination provided by the semaphore. While process A has the semaphore, process B is not permitted to perform its critical section.

Note that while process A is in its critical section, process B attempts to acquire the semaphore. As the semaphore has already been acquired by process A, process B is placed into a blocked state. When process A finally releases the semaphore, it is then granted to process B, which is allowed to enter its critical section. process B at a later time releases the semaphore, making it available for acquisition.

Semaphores commonly represent a point of synchronization in a system. For example, a semaphore can represent access to a shared resource. Only when the process has access to the semaphore can it access the shared resources. This ensures that only one process has access to the shared resource at a time, thus providing coordination between two or more users of the resource.

Semaphores were invented by Edsger Dijkstra for the T.H.E. operating system. Originally, the semaphore operations were defined as P and V. The P stands for the Dutch proberen, or to test, and the V for verhogen, or to increment.

Edsger Dijkstra used the train analogy to illustrate the critical section. Imagine two parallel train tracks that for a short duration merge into a single track. The single track is the shared resource and is also the critical section. The semaphore ensures that only one train is permitted on the shared track at a time. Not having the semaphore can have disastrous results on two trains trying to use the shared track at the same time. The effect on software is just as treacherous.

Types of Semaphores

Semaphores come in two basic varieties. The first are *binary semaphores*, as illustrated in Figure 17.1. The binary semaphore represents a single resource; therefore, when one process has acquired it, others are blocked until it is released.

The other style is the *counting semaphore*, which is used to represent shared resources in quantities greater than one. Consider a pool of buffers. A counting semaphore can represent the entire set of buffers by setting its value to the number of buffers available. Each time a process requires a buffer, it acquires the semaphore, which decrements its value. When the semaphore value reaches zero, processes are blocked until the value becomes nonzero. When a semaphore is released, the semaphore value is increased, thus permitting other processes to acquire a semaphore (and associated buffer). This is the one use for a counting semaphore (see Figure 17.2).

In the counting semaphore example, each process requires two resources before being able to perform its desired activities. In this example, the value of the counting semaphore is 3, which means that only one process is permitted to fully operate at a time. Process A acquires its two resources first, which means that process B blocks until process A releases at least one of its resources.

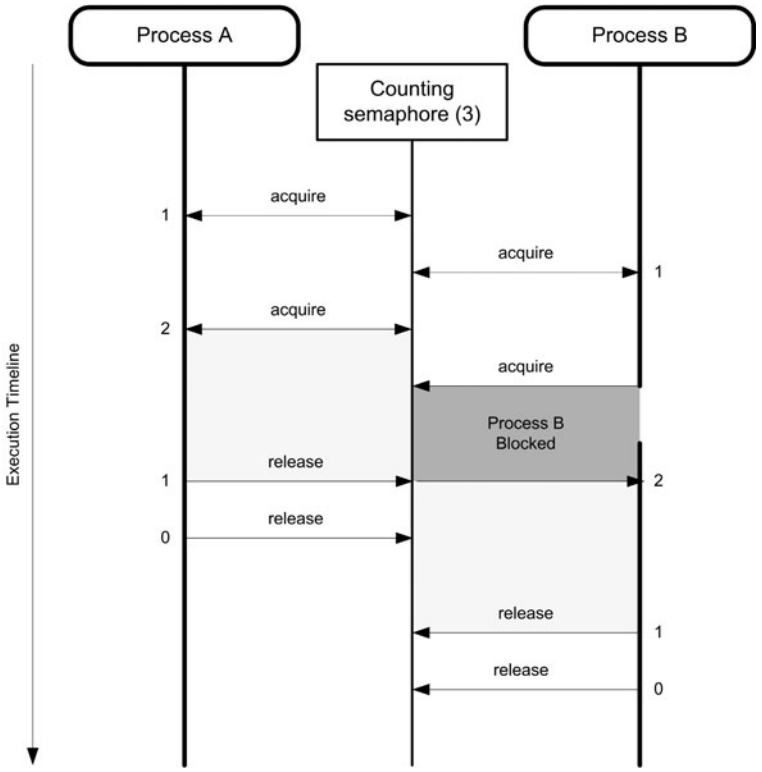

FIGURE 17.2 Counting semaphore example with two processes.

Quick Overview of GNU/Linux Semaphores

This chapter's discussion begins with a whirlwind tour of the GNU/Linux semaphore API. This section looks at code examples illustrating each of the API capabilities such as creating a new semaphore, finding a semaphore, acquiring a semaphore, releasing a semaphore, configuring a semaphore, and removing a semaphore. After you've finished the quick overview, you can dig deeper into the semaphore API.

Semaphores in GNU/Linux are actually semaphore arrays. A single semaphore can represent an array of 64 semaphores. This unique feature of GNU/Linux permits atomic operations over numerous semaphores at the same time. In the early discussions of GNU/Linux semaphores in this chapter, you explore single semaphore uses. In the detailed discussions that follow, you look at the more complex semaphore array examples.

Using the semaphore API requires that the function prototypes and symbols be available to the application. This is done by including the following three header files:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

Creating a Semaphore

To create a semaphore (or get an existing semaphore), you use the `semget` API function. This function takes a semaphore key, a semaphore count, and a set of flags. The count represents the number of semaphores in the set. In this case, you specify the need for one semaphore. The semaphore flags, argument 3 as shown in Listing 17.1, specify that the semaphore is to be created (`IPC_CREAT`). You also specify the read/write permissions to use (in this case `0666` for read/write for the user, group, and system in octal). An important item to consider is that when a semaphore is created, its value is zero. This suits for this example, but this chapter investigates later how to initialize the semaphore's value.

Listing 17.1 demonstrates creating a semaphore. In the following examples, you use the key `MY_SEM_ID` to represent your globally unique semaphore. At line 10, you use the `semget` with your semaphore key, semaphore set count, and command (with read/write permissions).

LISTING 17.1 Creating a Semaphore with `semget` (on the CD-ROM at `./source/ch17/semcreate.c`)

```
1:        #include <stdio.h>
2:        #include <sys/sem.h>
3:        #include "common.h"
4:
5:        int main()
6:        {
7:          int semid;
8:
9:          /* Create the semaphore with the id MY_SEM_ID */
10:         semid = semget( MY_SEM_ID, 1, 0666 | IPC_CREAT );
11:
12:         if (semid >= 0) {
13:
14:           printf( "semcreate: Created a semaphore %d\n", semid );
15:
16:         }
17:
```

```
18:         return 0;
19:       }
```

Upon completion of this simple application, a new globally available semaphore would be available with a key identified by `MY_SEM_ID`. Any process in the system could use this semaphore.

Getting and Releasing a Semaphore

Now take a look at an application that attempts to acquire an existing semaphore and also another application that releases it. Recall that your previously created semaphore (in Listing 17.1) was initialized with a value of zero. This is identical to a binary semaphore already having been acquired.

Listing 17.2 illustrates an application acquiring your semaphore. The GNU/Linux semaphore API is a little more complicated than many semaphore APIs, but it is POSIX compliant and, therefore, important for porting to other UNIX-like operating systems.

LISTING 17.2 Getting a Semaphore with `semop`

```
1:        #include <stdio.h>
2:        #include <sys/sem.h>
3:        #include <stdlib.h>
4:        #include "common.h"
5:
6:        int main()
7:        {
8:          int semid;
9:          struct sembuf sb;
10:
11:         /* Get the semaphore with the id MY_SEM_ID */
12:         semid = semget( MY_SEM_ID, 1, 0 );
13:
14:         if (semid >= 0) {
15:
16:           sb.sem_num = 0;
17:           sb.sem_op = -1;
18:           sb.sem_flg = 0;
19:
20:           printf( "semacq: Attempting to acquire semaphore
              %d\n", semid );
21:
```

```
22:          /* Acquire the semaphore */
23:          if ( semop( semid, &sb, 1 ) == -1 ) {
24:
25:            printf( "semacq: semop failed.\n" );
26:            exit(-1);
27:
28:          }
29:
30:          printf( "semacq: Semaphore acquired %d\n", semid );
31:
32:        }
33:
34:        return 0;
35:      }
```

You begin by identifying the semaphore identifier with `semget` at line 12. If this is successful, you build your semaphore operations structure (identified by the `sembuf` structure). This structure contains the semaphore number, the operation to be applied to the semaphore, and a set of operation flags. Because you have only one semaphore, you use the semaphore number zero to identify it. To acquire the semaphore, you specify an operation of `-1`. This subtracts one from the semaphore, but only if it's greater than zero to begin with. If the semaphore is already zero, the operation (and the process) blocks until the semaphore value is incremented.

With the `sembuf` created (variable `sb`), you use this with the API function `semop` to acquire the semaphore. You specify the semaphore identifier, your `sembuf` structure, and then the number of `sembuf`s that were passed in (in this case, one). This implies that you can provide an array of `sembuf`s, which is investigated later in the chapter. As long as the semaphore operation can finish (semaphore value is nonzero), then it returns with success (a non –1 value). This means that the process performing the `semop` has acquired the semaphore.

Now it's time to look at a release example. This example demonstrates the `semop` API function from the perspective of releasing the semaphore (see Listing 17.3).

In many cases, the release follows the acquire in the same process. This usage allows synchronization between two processes. The first process attempts to acquire the semaphore and then blocks when it's not available. The second process, knowing that another process is sitting blocked on the semaphore, releases it, allowing the process to continue. This provides a lock-step operation between the processes and is practical and useful.

LISTING 17.3 Releasing a Semaphore with `semop` (on the CD-ROM at `./source/ch17/semrel.c`)

```
1:        #include <stdio.h>
2:        #include <sys/sem.h>
3:        #include <stdlib.h>
4:        #include "common.h"
5:
6:        int main()
7:        {
8:          int semid;
9:          struct sembuf sb;
10:
11:         /* Get the semaphore with the id MY_SEM_ID */
12:         semid = semget( MY_SEM_ID, 1, 0 );
13:
14:         if (semid >= 0) {
15:
16:           printf( "semrel: Releasing semaphore %d\n", semid );
17:
18:           sb.sem_num = 0;
19:           sb.sem_op  = 1;
20:           sb.sem_flg = 0;
21:
22:           /* Release the semaphore */
23:           if (semop( semid, &sb, 1 ) == -1) {
24:
25:             printf("semrel: semop failed.\n");
26:             exit(-1);
27:
28:           }
29:
30:           printf( "semrel: Semaphore released %d\n", semid );
31:
32:         }
33:
34:         return 0;
35:       }
```

At line 12 of Listing 17.3, you first identify the semaphore of interest using the `semget` API function. Having your semaphore identifier, you build your `sembuf` structure to release the semaphore at line 23 using the `semop` API function. In this example, your `sem_op` element is 1 (compared to the –1 in Listing 17.2). In this

example, you are releasing the semaphore, which means that you're making it nonzero (and thus available).

It's important to note the symmetry the `sembuf` *uses in Listings 17.2 and 17.3. To acquire the semaphore, you subtract 1 from its value. To release the semaphore, you add 1 to its value. When the semaphore's value is zero, it's unavailable, forcing any processing attempting to acquire it to block. An initial value of 1 for the semaphore defines it as a binary semaphore. If the semaphore value is greater than zero, it can be considered a counting semaphore.*

Now take a look at a sample application of each of the functions discussed thus far. Listing 17.4 illustrates execution of Listing 17.1, `semcreate`, Listing 17.2, `semacq`, and Listing 17.3, `semrel`.

LISTING 17.4 Execution of the Sample Semaphore Applications

```
1:      # ./semcreate
2:      semcreate: Created a semaphore 1376259
3:      # ./semacq &
4:      [1] 12189
5:      semacq: Attempting to acquire semaphore 1376259
6:      # ./semrel
7:      semrel: Releasing semaphore 1376259
8:      semrel: Semaphore released 1376259
9:      # semacq: Semaphore acquired 1376259
10:
11:     [1]+  Done                    ./semacq
12:     #
```

At line 1, you create the semaphore. You emit the identifier associated with this semaphore, `1376259` (which is shown at line 2). Next, at line 3, you perform the `semacq` application, which acquires the semaphore. You run this in the background (identified by the trailing `&` symbol) because this application immediately blocks because the semaphore is unavailable. At line 4, you see the creation of the new subprocess (where `[1]` represents the number of subprocesses and `12189` is its process ID, or pid). The `semacq` application prints out its message, indicating that it's attempting to acquire the semaphore, but then it blocks. You then execute the `semrel` application to release the semaphore (line 6). You see two messages from this application; the first at line 7 indicates that it is about to release the semaphore, and then at line 8, you see that it was successful. Immediately thereafter, you see the `semacq` application acquires the newly released semaphore, given its output at line 9.

Finally, at line 11, you see the `semacq` application subprocess finish. Because it is unblocked (based upon the presence of its desired semaphore), the `semacq`'s main function reached its return, and thus the process finished.

Configuring a Semaphore

While a number of elements can be configured for a semaphore, this section looks specifically at reading and writing the value of the semaphore (the current count).

The first example, Listing 17.5, demonstrates reading the current value of the semaphore. You achieve this using the `semctl` API function.

LISTING 17.5 Retrieving the Current Semaphore Count (on the CD-ROM at `./source/ch17/semcrd.c`)

```
1:      #include <stdio.h>
2:      #include <sys/sem.h>
3:      #include <stdlib.h>
4:      #include "common.h"
5:
6:      int main()
7:      {
8:        int semid, cnt;
9:
10:       /* Get the semaphore with the id MY_SEM_ID */
11:       semid = semget( MY_SEM_ID, 1, 0 );
12:
13:       if (semid >= 0) {
14:
15:         /* Read the current semaphore count */
16:         cnt = semctl( semid, 0, GETVAL );
17:
18:         if (cnt != -1) {
19:
20:           printf("semcrd: current semaphore count %d.\n", cnt);
21:
22:         }
23:
24:       }
25:
26:       return 0;
27:     }
```

Reading the semaphore count is performed at line 16. You specify the semaphore identifier, the index of the semaphore (0), and the command (GETVAL). Note that the semaphore is identified by an index because it can possibly represent an array of semaphores (rather than one). The return value from this command is either –1 for error or the count of the semaphore.

You can configure a semaphore with a count using a similar mechanism (as shown in Listing 17.6).

LISTING 17.6 Setting the Current Semaphore Count

```
1:      #include <stdio.h>
2:      #include <sys/sem.h>
3:      #include <stdlib.h>
4:      #include "common.h"
5:
6:      int main()
7:      {
8:        int semid, ret;
9:
10:       /* Get the semaphore with the id MY_SEM_ID */
11:       semid = semget( MY_SEM_ID, 1, 0 );
12:
13:       if (semid >= 0) {
14:
15:         /* Read the current semaphore count */
16:         ret = semctl( semid, 0, SETVAL, 6 );
17:
18:         if (ret != -1) {
19:
20:           printf( "semcrd: semaphore count updated.\n" );
21:
22:         }
23:
24:       }
25:
26:       return 0;
27:     }
```

As with retrieving the current semaphore value, you can set this value using the semctl API function. The difference here is that along with the semaphore identifier (semid) and semaphore index (0), you specify the set command (SETVAL) and a value. In this example (line 16 of Listing 17.6), you are setting the semaphore value

to 6. Setting the value to 6, as shown here, changes the binary semaphore to a counting semaphore. This means that six semaphore acquires are permitted before an acquiring process blocks.

Removing a Semaphore

Removing a semaphore is also performed through the `semctl` API function. After retrieving the semaphore identifier (line 10 in Listing 17.7), you remove the semaphore using the `semctl` API function and the `IPC_RMID` command (at line 14).

LISTING 17.7 Removing a Semaphore

```
1:      #include <stdio.h>
2:      #include <sys/sem.h>
3:      #include "common.h"
4:
5:      int main()
6:      {
7:        int semid, ret;
8:
9:        /* Get the semaphore with the id MY_SEM_ID */
10:       semid = semget( MY_SEM_ID, 1, 0 );
11:
12:       if (semid >= 0) {
13:
14:         ret = semctl( semid, 0, IPC_RMID);
15:
16:         if (ret != -1) {
17:
18:           printf( "Semaphore %d removed.\n", semid );
19:
20:         }
21:
22:       }
23:
24:       return 0;
25:     }
```

As you can probably see, the semaphore API probably is not the simplest that you've used before.

That's it for the whirlwind tour; next the chapter explores the semaphore API in greater detail and looks at some of its other capabilities.

The Semaphore API

As noted before, the semaphore API handles not only the case of managing a single semaphore, but also groups (or arrays) of semaphores. This section investigates the use of those groups of semaphores. As a quick review, Table 17.1 shows the API functions and describes their uses. The following discussion continues to use the term *semaphore*, but note this can refer instead to a semaphore array.

TABLE 17.1 Semaphore API Functions and Their Uses

API Function	Uses
`semget`	Create a new semaphore.
	Get an existing semaphore.
`semop`	Acquire or release a semaphore.
`semctl`	Get info about a semaphore.
	Set info about a semaphore.
	Remove a semaphore.

The following sections address each of these functions using both simple examples (a single semaphore) and the more complex uses (semaphore arrays).

semget

The `semget` API function serves two fundamental roles. Its first use is in the creation of new semaphores. The second use is identifying an existing semaphore. In both cases, the response from `semget` is a semaphore identifier (a simple integer value representing the semaphore). The prototype for the `semget` API function is defined as follows:

```
int semget( key_t key, int nsems, int semflg );
```

The `key` argument specifies a system-wide identifier that uniquely identifies this semaphore. The key must be nonzero or the special symbol `IPC_PRIVATE`. The `IPC_PRIVATE` variable tells `semget` that no key is provided and to simply make one up. Because no key exists, other processes have no way to know about this semaphore. Therefore, it's a private semaphore for this particular process.

You can create a single semaphore (with an `nsems` value of 1) or multiple semaphores. If you're using `semget` to get an existing semaphore, this value can simply be zero.

Finally, the `semflg` argument allows you to alter the behavior of the `semget` API function. The `semflg` argument can take on three basic forms, depending upon what you desire. In the first form, you want to create a new semaphore. In this case, the `semflg` argument must be the `IPC_CREAT` value OR'd with the permissions (see Table 17.2). The second form also provides for semaphore creation, but with the constraint that if the semaphore already exists, an error is generated. This second form requires the `semflg` argument to be set to `IPC_CREAT | IPC_EXCL` along with the permissions. If the second form is used and the semaphore already exists, the call fails (–1 return) with `errno` set to `EEXIST`. The third form takes a zero for `semflg` and identifies that an existing semaphore is being requested.

TABLE 17.2 Semaphore Permissions for the `semget semflg` Argument

Symbol	Value	Meaning
`S_IRUSR`	0400	User has read permission.
`S_IWUSR`	0200	User has write permission.
`S_IRGRP`	0040	Group has read permission.
`S_IWGRP`	0020	Group has write permission.
`S_IROTH`	0004	Other has read permission.
`S_IWOTH`	0002	Other has write permission.

Now it's time to look at a few examples of `semget`, used in each of the three scenarios defined earlier in this section. In the examples that follow, assume `semid` is an `int` value, and `mySem` is of type `key_t`. In the first example, you create a new semaphore (or access an existing one) of the private type.

```
semid = semget( IPC_PRIVATE, 1, IPC_CREAT | 0666 );
```

After the `semget` call completes, the semaphore identifier is stored in `semid`. Otherwise, if an error occurs, a –1 is returned. Note that in this example (using `IPC_PRIVATE`), `semid` is all you have to identify this semaphore. If `semid` is somehow lost, you have no way to find this semaphore again.

In the next example, you create a semaphore using a system-wide unique key value (`0x222`). You also indicate that if the semaphore already exists, you don't simply get its value, but instead fail the call. Recall that this is provided by the `IPC_EXCL` command, as follows:

```
// Create a new semaphore
semid = semget( 0x222, 1, IPC_CREAT | IPC_EXCL | 0666 );
if ( semid == -1 ) {
  printf( "Semaphore already exists, or error\n" );
} else {
  printf( "Semaphore created (id %d)\n", semid );
}
```

If you don't want to rely on the fact that `0x222` might not be unique in your system, you can use the `ftok` system function. This function provides the means to create a new unique key in the system. It does this by using a known file in the filesystem and an integer number. The file in the filesystem is unique by default (considering its path). Therefore, by using the unique file (and integer), it's an easy task to then create a unique system-wide value. Take a look at an example of the use of `ftok` to create a unique key value. Assume for this example that your file and path are defined as `/home/mtj/semaphores/mysem`.

```
key_t mySem;
int   semid;
// Create a key based upon the defined path and number
myKey = ftok( "/home/mtj/semaphores/mysem", 0 );
semid = semget( myKey, 1, IPC_CREAT | IPC_EXCL | 0666 );
```

Note that each time `ftok` is called with those parameters, the same key is generated (which is why this method works at all!). As long as each process that needs access to the semaphore knows about the file and number, the key can be recalculated and then used to identify the semaphore.

In the examples discussed thus far, you've created a single semaphore. You can create an array of semaphores by simply specifying an `nsems` value greater than one, such as the following:

```
semarrayid = semget( myKey, 10, IPC_CREAT | 0666 );
```

The result is a semaphore array created that consists of 10 semaphores. The return value (`semarrayid`) represents the entire set of semaphores. You get a chance to see how individual semaphores can be addressed in the `semctl` and `semop` discussions later in the chapter.

In this last example of `semget`, you simply get the semaphore identifier of an existing semaphore. In this example, you specify the key value and no command:

```
semid = semget( 0x222, 0, 0 );
if ( semid == -1 ) {
  printf( "Semaphore does not exist...\n" );
}
```

One final note on semaphores is that, just as is the case with message queues, a set of defaults is provided to the semaphore as it's created. The parameters that are defined are shown in Table 17.3. Later on in the discussion of `semctl`, you can see how some of the parameters can be changed.

TABLE 17.3 Semaphore Internal Values

Parameter	Default Value
`sem_perm.cuid`	Effective user ID of the calling process (creator)
`sem_perm.uid`	Effective user ID of the calling process (owner)
`sem_perm.cgid`	Effective group ID of the calling process (creator)
`sem_perm.gid`	Effective group ID of the calling process (owner)
`sem_perm.mode`	Permissions (lower 9 bits of `semflg`)
`sem_nsems`	Set to the value of `nsems`
`sem_otime`	Set to zero (last `semop` time)
`sem_ctime`	Set to the current time (create time)

The process can override some of these parameters. You get to explore this later in the discussion of `semctl`.

semctl

The `semctl` API function provides a number of control operations on semaphores or semaphore arrays. Examples of functionality range from setting the value of the semaphore (as shown in Listing 17.6) to removing a semaphore or semaphore array (see Listing 17.7). You get a chance to see these and other examples in this section.

The function prototype for the `semctl` call is as follows:

```
int semctl( int semid, int semnum, int cmd, ... );
```

The first argument defines the semaphore identifier, the second defines the semaphore number of interest, the third defines the command to be applied, and then potentially another argument (usually defined as a union). The operations that can be performed are shown in Table 17.4.

TABLE 17.4 Operations That Can Be Performed Using `semctl`

Command	Description	Fourth Argument
`GETVAL`	Return the semaphore value.	
`SETVAL`	Set the semaphore value.	`int`
`GETPID`	Return the process ID that last operated on the semaphore (`semop`).	
`GETNCNT`	Return the number of processes awaiting the defined semaphore to increase in value.	`int`
`GETZCNT`	Return the number of processes awaiting the defined semaphore to become zero.	`int`
`GETALL`	Return the value of each semaphore in a semaphore array.	`u_short*`
`SETALL`	Set the value of each semaphore in a semaphore array.	`u_short*`
`IPC_STAT`	Return the effective user, group, and permissions for a semaphore.	`struct semid_ds*`
`IPC_SET`	Set the effective user, group, and permissions for a semaphore.	`struct semid_ds*`
`IPC_RMID`	Remove the semaphore or semaphore array.	

Now it's time to look at some examples of each of these operations in `semctl`, focusing on semaphore array examples where applicable. The first example illustrates the setting of a semaphore value and then returning its value. In this example, you first set the value of the semaphore to `10` (using the command `SETVAL`) and then read it back out using `GETVAL`. Note that the `semnum` argument (argument 2) defines an individual semaphore. Later on, you can take look at the semaphore array case with `GETALL` and `SETALL`.

```
int semid, ret, value;
...
/* Set the semaphore to 10 */
ret = semctl( semid, 0, SETVAL, 10 );
...
/* Read the semaphore value (return value) */
value = semctl( semid, 0, GETVAL );
```

The GETPID command allows you to identify the last process that performed a semop on the semaphore. The process identifier is the return value, and argument 4 is not used in this case.

```
int semid, pid;
...
pid = semctl( semid, 0, GETPID );
```

If no semop has been performed on the semaphore, the return value is zero.

To identify the number of semaphores that are currently awaiting a semaphore to increase in value, you can use the GETNCNT command. You can also identify the number of processes that are awaiting the semaphore value to become zero using GETZCNT. Both of these commands are illustrated in the following for the semaphore numbered zero:

```
int semid, count;
/* How many processes are awaiting this semaphore to increase */
count = semctl( semid, 0, GETNCNT );
/* How many processes are awaiting this semaphore to become zero */
count = semctl( semid, 0, GETZCNT );
```

Now it's time to take a look at an example of some semaphore array operations. Listing 17.8 illustrates both the SETVAL and GETVAL commands with semctl.

In this example, you create a semaphore array of 10 semaphores. The creation of the semaphore array is performed at lines 20–21 using the semget API function. Note that because you're going to create and remove the semaphore array within this same function, you use no key and instead use the IPC_PRIVATE key. The MAX_SEMAPHORES symbol defines the number of semaphores that you are going to create, and finally you specify that you are creating the semaphore array (IPC_CREAT) with the standard permissions.

Next, you initialize the semaphore value array (lines 26–30). While this is not a traditional example, you initialize each semaphore to one plus its semnum (so semaphore zero has a value of one, semaphore one has a value of two, and so on). You do this so that you can inspect the value array later and know what you're looking

at. At line 33, you set the `arg.array` parameter to the address of the array (`sem_array`). Note that you're using the `semun` union, which defines some commonly used types for semaphores. In this case, you use the `unsigned short` field to represent an array of semaphore values.

At line 36, you use the `semctl` API function and the `SETALL` command to set the semaphore values. You provide the semaphore identifier `semnum` as zero (unused in this case), the `SETALL` command, and finally the `semun` union. Upon return of this API function, the semaphore array identified by `semid` has the values as defined in `sem_array`.

Next, you explore the `GETALL` command, which retrieves the array of values for the semaphore array. You first set your `arg.array` to a new array (just to avoid reusing the existing array that has the contents that you are looking for), at line 41. At line 44, you call `semctl` again with the `semid`, zero for `semnum` (unused here, again), the `GETALL` command, and the `semun` union.

To illustrate what you have read, you next loop through the `sem_read_array` and emit each value for each semaphore index within the semaphore array (lines 49–53).

While `GETALL` allows you to retrieve the entire semaphore array in one call, you can perform the same action using the `GETVAL` command, calling `semctl` for each semaphore of the array. This is illustrated at lines 56–62. This also applies to using the `SETVAL` command to mimic the `SETALL` behavior.

Finally, at line 65, you use the `semctl` API function with the `IPC_RMID` command to remove the semaphore array.

LISTING 17.8 Creating and Manipulating Semaphore Arrays (on the CD-ROM at `./source/ch17/semall.c`)

```
1:        #include <stdio.h>
2:        #include <sys/types.h>
3:        #include <sys/sem.h>
4:        #include <errno.h>
5:
6:        #define MAX_SEMAPHORES  10
7:
8:        int main()
9:        {
10:         int i, ret, semid;
11:         unsigned short sem_array[MAX_SEMAPHORES];
12:         unsigned short sem_read_array[MAX_SEMAPHORES];
13:
14:         union semun {
15:           int val;
```

```
16:           struct semid_ds *buf;
17:           unsigned short *array;
18:         } arg;
19:
20:         semid = semget( IPC_PRIVATE, MAX_SEMAPHORES,
21:                         IPC_CREAT | 0666 );
22:
23:         if (semid != -1) {
24:
25:           /* Initialize the sem_array */
26:           for ( i = 0 ; i < MAX_SEMAPHORES ; i++ ) {
27:
28:             sem_array[i] = (unsigned short)(i+1);
29:
30:           }
31:
32:           /* Update the arg union with the sem_array address */
33:           arg.array = sem_array;
34:
35:           /* Set the values of the semaphore-array */
36:           ret = semctl( semid, 0, SETALL, arg );
37:
38:           if (ret == -1) printf("SETALL failed (%d)\n", errno);
39:
40:           /* Update the arg union with another array for read */
41:           arg.array = sem_read_array;
42:
43:           /* Read the values of the semaphore array */
44:           ret = semctl( semid, 0, GETALL, arg );
45:
46:           if (ret == -1) printf("GETALL failed (%d)\n", errno);
47:
48:           /* print the sem_read_array */
49:           for ( i = 0 ; i < MAX_SEMAPHORES ; i++ ) {
50:
51:             printf("Semaphore %d, value %d\n", i,
               sem_read_array[i] );
52:
53:           }
54:
55:           /* Use GETVAL in a similar manner */
56:           for ( i = 0 ; i < MAX_SEMAPHORES ; i++ ) {
57:
58:             ret = semctl( semid, i, GETVAL );
59:
```

```
60:             printf("Semaphore %d, value %d\n", i, ret );
61:
62:           }
63:
64:           /* Delete the semaphore */
65:           ret = semctl( semid, 0, IPC_RMID );
66:
67:         } else {
68:
69:           printf("Could not allocate semaphore (%d)\n", errno);
70:
71:         }
72:
73:         return 0;
74:       }
```

Executing this application (called `semall`) produces the output shown in Listing 17.9. Not surprisingly, the `GETVAL` emits identical output as that shown for the `GETALL`.

LISTING 17.9 Output from the `semall` Application Shown in Listing 17.8

```
# ./semall
Semaphore 0, value 1
Semaphore 1, value 2
Semaphore 2, value 3
Semaphore 3, value 4
Semaphore 4, value 5
Semaphore 5, value 6
Semaphore 6, value 7
Semaphore 7, value 8
Semaphore 8, value 9
Semaphore 9, value 10
Semaphore 0, value 1
Semaphore 1, value 2
Semaphore 2, value 3
Semaphore 3, value 4
Semaphore 4, value 5
Semaphore 5, value 6
Semaphore 6, value 7
Semaphore 7, value 8
Semaphore 8, value 9
Semaphore 9, value 10
#
```

The IPC_STAT command retrieves the current information about a semaphore or semaphore array. The data is retrieved into a structure called semid_ds and contains a variety of parameters. The application that reads this information is shown in Listing 17.10. You read the semaphore information at line 23 using the semctl API function and the IPC_STAT command. The information captured is then emitted at lines 27–49.

LISTING 17.10 Reading Semaphore Information Using IPC_STAT (on the CD-ROM at ./source/ch17/semstat.c)

```
1:        #include <stdio.h>
2:        #include <sys/sem.h>
3:        #include <time.h>
4:        #include "common.h"
5:
6:        int main()
7:        {
8:          int semid, ret;
9:          struct semid_ds sembuf;
10:
11:         union semun {
12:           int val;
13:           struct semid_ds *buf;
14:           unsigned short *array;
15:         } arg;
16:
17:         /* Get the semaphore with the id MY_SEM_ID */
18:         semid = semget( MY_SEM_ID, 1, 0 );
19:
20:         if (semid >= 0) {
21:
22:           arg.buf = &sembuf;
23:           ret = semctl( semid, 0, IPC_STAT, arg );
24:
25:           if (ret != -1) {
26:
27:             if (sembuf.sem_otime) {
28:               printf( "Last semop time %s",
29:                       ctime( &sembuf.sem_otime ) );
30:             }
31:
```

```
32:             printf( "Last change time %s",
33:                      ctime( &sembuf.sem_ctime ) );
34:
35:             printf( "Number of semaphores %ld\n",
36:                      sembuf.sem_nsems );
37:
38:             printf( "Owner's user id %d\n",
39:                      sembuf.sem_perm.uid );
40:             printf( "Owner's group id %d\n",
41:                      sembuf.sem_perm.gid );
42:
43:             printf( "Creator's user id %d\n",
44:                      sembuf.sem_perm.cuid );
45:             printf( "Creator's group id %d\n",
46:                      sembuf.sem_perm.cgid );
47:
48:             printf( "Permissions 0%o\n",
49:                      sembuf.sem_perm.mode );
50:
51:           }
52:
53:         }
54:
55:         return 0;
56:       }
```

Three of the fields shown can be updated through another call to `semctl` using the `IPC_SET` call. The three updateable parameters are the effective user ID (`sem_perm.uid`), the effective group ID (`sem_perm.gid`), and the permissions (`sem_perm.mode`). The following code snippet illustrates modifying the permissions:

```
/* First, read the semaphore information */
arg.buf = &sembuf;
ret = semctl( semid, 0, IPC_STAT, arg );
/* Next, update the permissions */
sembuf.sem_perm.mode = 0644;
/* Finally, update the semaphore information */
ret = semctl( semid, 0, IPC_SET, arg );
```

After the `IPC_SET semctl` has completed, the last change time (`sem_ctime`) is updated to the current time.

Finally, the IPC_RMID command permits you to remove a semaphore or semaphore array. A code snippet demonstrating this process is shown in the following:

```
int semid;
...
semid = semget( the_key, NUM_SEMAPHORES, 0 );
ret = semctl( semid, 0, IPC_RMID );
```

Note that if any processes are currently blocked on the semaphore, they are immediately unblocked with an error return and errno is set to EIDRM.

semop

The semop API function provides the means to acquire and release a semaphore or semaphore array. The basic operations provided by semop are to decrement a semaphore (acquire one or more semaphores) or to increment a semaphore (release one or more semaphores). The API for the semop function is defined as follows:

```
int semop( int semid, struct sembuf *sops, unsigned int nsops );
```

The semop takes three parameters: a semaphore identifier (semid), a sembuf structure, and the number of semaphore operations to be performed (nsops). The semaphore structure defines the semaphore number of interest, the operation to perform, and a flag word that can be used to alter the behavior of the operation. The sembuf structure is shown as follows:

```
struct sembuf {
    unsigned short sem_num;
    short sem_op;
    short sem_flg;
};
```

As you can imagine, the sembuf array can produce very complex semaphore interactions. You can acquire one semaphore and release another in a single semop operation.

Take a look at a simple application that acquires 10 semaphores in one operation. This application is shown in Listing 17.11.

An important difference to notice here is that rather than specify a single sembuf structure (as you did in single semaphore operations), you specify an array of sembufs (line 9). You identify your semaphore array at line 12; note again that you specify the number of semaphores (nsems, or number of semaphores, as argument 2). You build out your sembuf array as acquires (with a sem_op of –1) and also

initialize the `sem_num` field with the semaphore number. This specifies that you want to acquire each of the semaphores in the array. If one or more aren't available, the operation blocks until all semaphores can be acquired.

At line 26, you perform the `semop` API function to acquire the semaphores. Upon acquisition (or error), the `semop` function returns to the application. As long as the return value is not –1, you have successfully acquired the semaphore array. Note that you can specify –2 for each `sem_op`, which requires that two counts of the semaphore are needed for successful acquisition.

LISTING 17.11 Acquiring an Array of Semaphores Using `semop` (on the CD-ROM at `./source/ch17/semaacq.c`)

```
1:      #include <stdio.h>
2:      #include <sys/sem.h>
3:      #include <stdlib.h>
4:      #include "common.h"
5:
6:      int main()
7:      {
8:        int semid, i;
9:        struct sembuf sb[10];
10:
11:       /* Get the semaphore with the id MY_SEM_ID */
12:       semid = semget( MY_SEMARRAY_ID, 10, 0 );
13:
14:       if (semid >= 0) {
15:
16:         for (i = 0 ; i < 10 ; i++) {
17:           sb[i].sem_num = i;
18:           sb[i].sem_op = -1;
19:           sb[i].sem_flg = 0;
20:         }
21:
22:         printf( "semaacq: Attempting to acquire semaphore %d\n",
23:                  semid );
24:
25:         /* Acquire the semaphores */
26:         if (semop( semid, &sb[0], 10 ) == -1) {
27:
28:           printf("semaacq: semop failed.\n");
29:           exit(-1);
30:
31:         }
```

```
32:
33:           printf( "semaacq: Semaphore acquired %d\n", semid );
34:
35:         }
36:
37:         return 0;
38:   }
```

Next, take a look at the semaphore release operation. This includes only the changes from Listing 17.11, as otherwise they are very similar (on the CD-ROM at `./source/ch17/semarel.c`). In fact, the only difference is the `sembuf` initialization:

```
for ( i = 0 ; i < 10 ; i++ ) {
  sb[i].sem_num = i;
  sb[i].sem_op = 1;
  sb[i].sem_flg = 0;
}
```

In this example, you increment the semaphore (release) instead of decrementing it (as was done in Listing 17.11).

The `sem_flg` within the `sembuf` structure permits you to alter the behavior of the `semop` API function. Two flags are possible, as shown in Table 17.5.

TABLE 17.5 Semaphore Flag Options (`sembuf.sem_flg`)

Flag	Purpose
`SEM_UNDO`	Undo the semaphore operation if the process exits.
`IPC_NOWAIT`	Return immediately if the semaphore operation cannot be performed (if the process would block) and return an `errno` of `EAGAIN`.

Another useful operation that can be performed on semaphores is the wait-for-zero operation. In this case, the process is blocked until the semaphore value becomes zero. This operation is performed by simply setting the `sem_op` field to zero, as follows:

```
struct sembuf sb;
...
sb.sem_num = 0;        // semaphore 0
sb.sem_op = 0;        // wait for zero
sb.sem_flg = 0;        // no flags
...
```

As with previous `semop`s, setting `sem_flg` with `IPC_NOWAIT` causes `semop` to return immediately if the operation blocks with an `errno` of `EAGAIN`.

Finally, if the semaphore is removed while a process is blocked on it (via a `semop` operation), the process becomes immediately unblocked and an `errno` value is returned as `EIDRM`.

User Utilities

GNU/Linux provides the `ipcs` command to explore semaphores from the command line. The `ipcs` utility provides information on a variety of resources; this section explores its use for investigating semaphores.

The general form of the `ipcs` utility for semaphores is as follows:

```
# ipcs -s
```

This presents all the semaphores that are visible to the calling process. Take a look at an example where you create a semaphore (as was done in Listing 17.1):

```
# ./semcreate
semcreate: Created a semaphore 1769475
# ipcs -s

—— Semaphore Arrays ——
key        semid      owner      perms      nsems
0x0000006f 1769475    mtj        666        1

#
```

Here, you see your newly created semaphore (`key 0x6f`). You can get extended information about the semaphore using the `-i` option. This allows you to specify a specific semaphore ID, for example:

```
# ipcs -s -i 1769475

Semaphore Array semid=1769475
uid=500  gid=500         cuid=500        cgid=500
mode=0666, access_perms=0666
nsems = 1
otime = Not set
ctime = Fri Apr  9 17:50:01 2004
semnum     value      ncount     zcount     pid
0          0          0          0          0

#
```

Here you see your semaphore in greater detail. You see the owner and creator process and group IDs, permissions, number of semaphores (`nsems`), last `semop` time, last change time, and the details of the semaphore itself (`semnum` through `pid`). The value represents the actual value of the semaphore (zero after creation). If you were to perform the release operation (see Listing 17.3) and then perform this command again, you would then see this:

```
# ./semrel
    semrel: Releasing semaphore 1769475
    semrel: Semaphore released 1769475
    # ipcs -s -i 1769475

    Semaphore Array semid=1769475
    uid=500  gid=500         cuid=500        cgid=500
    mode=0666, access_perms=0666
    nsems = 1
    otime = Fri Apr  9 17:54:44 2004
    ctime = Fri Apr  9 17:50:01 2004
    semnum     value      ncount     zcount     pid
    0          1          0          0          20494

    #
```

Note here that your value has increased (based upon the semaphore release), and other information (such as `otime` and `pid`) has been updated given a semaphore operation having been performed.

You can also delete semaphores from the command line using the `ipcrm` command. To delete your previously created semaphore, you simply use the `ipcrm` command as follows:

```
# ipcrm -s 1769475
[mtj@camus ch17]$ ipcs -s

—— Semaphore Arrays ——
key        semid      owner      perms      nsems

#
```

As with the `semop` and `semctl` API functions, the `ipcrm` command uses the semaphore identifier to specify the semaphore to be removed.

Summary

This chapter introduced the semaphore API and its application of interprocess coordination and synchronization. It began with a whirlwind tour of the API and then followed with a detailed description of each command including examples of each. Finally, the chapter reviewed the `ipcs` and `ipcrm` commands and demonstrated their debugging and semaphore management capabilities.

Semaphore APIs

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget( key_t key, int nsems, int semflg );
int semop( int semid, struct sembuf *sops, unsigned int nsops );
int semctl( int semid, int semnum, int cmd, ... );
```

This page intentionally left blank

18 Shared Memory Programming

In This Chapter

- Introduction to Shared Memory
- Creating and Configuring Shared Memory Segments
- Using and Protecting Shared Memory Segments
- Locking and Unlocking Shared Segments
- Using the `ipcs` and `ipcrm` Utilities

Introduction

Shared memory APIs are the final topic of interprocess communication (IPC) that this book details. Shared memory allows two or more processes to share a chunk of memory (mapped to each of the process's individual address spaces) so that each can communicate with all others. Shared memory goes even further, as you will see in this chapter.

Recall from Chapter 13, "Introduction to Sockets Programming," that the address spaces for parent and child processes are independent. The parent process can create a chunk of memory (such as declaring an array), but after the fork completes, the parent and child see different memory. In GNU/Linux, all processes have unique virtual address spaces, but the shared memory application programming interface (API) permits a process to attach to a common (shared) address segment.

With all this power comes some complexity. For example, when processes share memory segments, they must also provide a means to coordinate access to them.

This is commonly provided via a semaphore (by the developer), which can be contained within the shared memory segment itself. This chapter looks at this specific technique.

If shared memory segments have this disadvantage, why not use an existing IPC mechanism that has built-in coordination, such as message queues? The answer also lies in the simplicity of shared memory. When you use a message queue, one process writes a message to a queue, which involves a copy from the user's address space to the kernel space. When another user reads from the message queue, another copy is performed from the kernel's address space to the new user's address space. The benefit of shared memory is that you minimize copying in its entirety. The segment is shared between the two processes in their own address spaces, so bulk copies of data are not necessary.

Because processes share the memory segment in each of their address spaces, copies are minimized in sharing data. For this reason, shared memory can be the fastest form of IPC available within GNU/Linux.

Quick Overview of Shared Memory

This section takes a quick look at the shared memory APIs. Later, the next section digs into the API further. First, however, this section looks at code snippets to create a shared memory segment, get an identifier for an existing one, configure a segment, attach, and detach, and it also gives some examples of processes using them.

Using the shared memory API requires the function prototypes and symbols to be available to the application. This is done by including the following (at the top of the C source file):

```
#include <sys/ipc.h>
#include <sys/shm.h>
```

Creating a Shared Memory Segment

To create a shared memory segment, you use the `shmget` API function. Using `shmget`, you specify a unique shared memory ID, the size of the segment, and finally a set of flags (see Listing 18.1). The `flags` argument, as you saw with message queues and semaphores, includes both access permissions and a command to create the segment (`IPC_CREAT`).

LISTING 18.1 Creating a Shared Memory Segment with `shmget` (on the CD-ROM at `./source/ch18/shmcreate.c`)

```
 1:    #include <stdio.h>
 2:    #include <sys/shm.h>
 3:    #include "common.h"
 4:
 5:    int main()
 6:    {
 7:      int shmid;
 8:
 9:      /* Create the shared memory segment using MY_SHM_ID */
10:      shmid = shmget( MY_SHM_ID, 4096, 0666 | IPC_CREAT );
11:
12:      if ( shmid >= 0 ) {
13:
14:        printf( "Created a shared memory segment %d\n", shmid );
15:
16:      }
17:
18:      return 0;
19:    }
```

At line 10 in Listing 18.1, you create a new shared memory segment that's 4 KB in size. The size specified must be evenly divisible by the page size of the architecture in question (typically 4 KB). The return value of `shmget` (stored in `shmid`) can be used in subsequent calls to configure or attach to the segment.

To identify the page size on a given system, simply call the `getpagesize` *function. This returns the number of bytes contained within a system page.*

```
#include <unistd.h>
int getpagesize( void );
```

Getting Information on a Shared Memory Segment

You can also get information about a shared memory segment and even set some parameters. The `shmctl` API function provides a number of capabilities. Here, you take a look at retrieving information about a shared memory segment.

In Listing 18.2, you first get the shared memory identifier for the segment using the `shmget` API function. After you have this, you can call `shmctl` to grab the current stats. To `shmctl` you pass the identifier for the shared memory segment, the command to grab stats (`IPC_STAT`), and finally a buffer in which the data is written.

This buffer is a structure of type `shmid_ds`. You will look more at this structure in the "`shmctl`" section later in this chapter. Upon successful return of `shmctl`, identified by a zero return, you emit your data of interest. Here, you emit the size of the shared memory segment (`shm_segsz`) and the number of attaches that have been performed on the segment (`shm_nattch`).

LISTING 18.2 Retrieving Information about a Shared Memory Segment (on the CD-ROM at `./source/ch18/shmszget.c`)

```
#include <stdio.h>
#include <sys/shm.h>
#include <errno.h>
#include "common.h"

int main()
{
  int shmid, ret;
  struct shmid_ds shmds;

  /* Get the shared memory segment using MY_SHM_ID */
  shmid = shmget( MY_SHM_ID, 0, 0 );

  if ( shmid >= 0 ) {

    ret = shmctl( shmid, IPC_STAT, &shmds );

    if (ret == 0) {

      printf( "Size of memory segment is %d\n", shmds.shm_segsz
      );
      printf( "Number of attaches %d\n", (int)shmds.shm_nattch );

    } else {

      printf( "shmctl failed (%d)\n", errno );

    }

  } else {

    printf( "Shared memory segment not found.\n" );

  }
```

```
34:
35:       return 0;
36:     }
```

Attaching and Detaching a Shared Memory Segment

To use your shared memory, you must attach to it. Attaching to a shared memory segment maps the shared memory into your process's memory space. To attach to the segment, you use the `shmat` API function. This returns a pointer to the segment in the process's address space. This address can then be used by the process like any other memory reference. You detach from the memory segment using the `shmdt` API function.

In Listing 18.3, a simple application is shown to attach to and detach from a shared memory segment. You first get the shared memory identifier using `shmget` (at line 12). At line 16, you attach to the segment using `shmat`. You specify your identifier and an address (where you want to place it in your address space) and an options word (`0`). After checking, if this was successful (the return of a nonzero address from `shmat`), you detach from the segment at line 23 using `shmdt`.

LISTING 18.3 Attaching to and Detaching from a Shared Memory Segment (on the CD-ROM at `./source/ch18/shmattch.c`)

```
 1:    #include <stdio.h>
 2:    #include <sys/shm.h>
 3:    #include <errno.h>
 4:    #include "common.h"
 5:
 6:    int main()
 7:    {
 8:      int shmid, ret;
 9:      void *mem;
10:
11:      /* Get the shared memory segment using MY_SHM_ID */
12:      shmid = shmget( MY_SHM_ID, 0, 0 );
13:
14:      if ( shmid >= 0 ) {
15:
16:        mem = shmat( shmid, (const void *)0, 0 );
17:
18:        if ( (int)mem != -1 ) {
19:
20:          printf( "Shared memory was attached in our "
```

```
21:                   "address space at %p\n", mem );
22:
23:          ret = shmdt( mem );
24:
25:          if (ret == 0) {
26:
27:            printf("Successfully detached memory\n");
28:
29:          } else {
30:
31:            printf("Memory detached Failed (%d)\n", errno);
32:
33:          }
34:
35:        } else {
36:
37:          printf( "shmat failed (%d)\n", errno );
38:
39:        }
40:
41:      } else {
42:
43:        printf( "Shared memory segment not found.\n" );
44:
45:      }
46:
47:      return 0;
48:    }
```

Using a Shared Memory Segment

Now take a look at two processes that use a shared memory segment. For brevity, this example passes on the error checking. First you take a look at the write example. In Listing 18.4, you see a short example that uses the `strcpy` standard library function to write to the shared memory segment. Because the segment is just a block of memory, you cast it from a `void` pointer to a character pointer in order to write to it (avoiding compiler warnings) at line 16. It's important to note that a shared memory segment is nothing more than a block of memory, and anything you would expect to do with a memory reference is possible with the shared memory block.

LISTING 18.4 Writing to a Shared Memory Segment (on the CD-ROM at `./source/ch18/shmwrite.c`)

```
#include <stdio.h>
#include <sys/shm.h>
#include <string.h>
#include "common.h"

int main()
{
  int shmid, ret;
  void *mem;

  /* Get the shared memory segment using MY_SHM_ID */
  shmid = shmget( MY_SHM_ID, 0, 0 );

  mem = shmat( shmid, (const void *)0, 0 );

  strcpy( (char *)mem, "This is a test string.\n" );

  ret = shmdt( mem );

  return 0;
}
```

Now take a look at a read example. In Listing 18.5, you see a similar application to Listing 18.4. In this particular case, you read from the block of memory by using the `printf` call. In Listing 18.4 (the write application), you copied a string into the block with the `strcpy` function. Now in Listing 18.5, you emit that same string using `printf`. Note that the first process attaches to the memory, writes the string, and then detaches and exits. The next process attaches and reads from the memory. Any number of processes can read or write to this memory, which is one of the basic problems. Some solutions for this problem are investigated in the section "Using a Shared Memory Segment" later in this chapter.

LISTING 18.5 Reading from a Shared Memory Segment (on the CD-ROM at `./source/ch18/shmread.c`)

```
#include <stdio.h>
#include <sys/shm.h>
#include <string.h>
#include "common.h"

int main()
```

```
 7:    {
 8:      int shmid, ret;
 9:      void *mem;
10:
11:      /* Get the shared memory segment using MY_SHM_ID */
12:      shmid = shmget( MY_SHM_ID, 0, 0 );
13:
14:      mem = shmat( shmid, (const void *)0, 0 );
15:
16:      printf( "%s", (char *)mem );
17:
18:      ret = shmdt( mem );
19:
20:      return 0;
21:    }
```

Removing a Shared Memory Segment

To permanently remove a shared memory segment, you use the `shmctl` API function. You use a special command with `shmctl` called `IPC_RMID` to remove the segment (much as is done with message queues and semaphores). Listing 18.6 illustrates the segment removal.

LISTING 18.6 Removing a Shared Memory Segment (on the CD-ROM at `./source/ch18/shmdel.c`)

```
 1:    #include <stdio.h>
 2:    #include <sys/shm.h>
 3:    #include <errno.h>
 4:    #include "common.h"
 5:
 6:    int main()
 7:    {
 8:      int shmid, ret;
 9:
10:      /* Create the shared memory segment using MY_SHM_ID */
11:      shmid = shmget( MY_SHM_ID, 0, 0 );
12:
13:      if ( shmid >= 0 ) {
14:
15:        ret = shmctl( shmid, IPC_RMID, 0 );
16:
17:        if (ret == 0) {
18:
19:          printf( "Shared memory segment removed\n" );
```

```
20:
21:         } else {
22:
23:           printf( "shmctl failed (%d)\n", errno );
24:
25:         }
26:
27:       } else {
28:
29:         printf( "Shared memory segment not found.\n" );
30:
31:       }
32:
33:       return 0;
34:     }
```

After the shared memory segment identifier is found (line 11), you call `shmctl` with the `IPC_RMID` argument at line 15.

That completes the quick tour of the shared memory API. The next section digs deeper into the APIs and looks at some of the more detailed aspects.

Shared Memory APIs

Now that the quick review is finished, you can start your deeper look into the APIs by looking at Table 18.1, which provides the shared memory API functions, along with their basic uses.

TABLE 18.1 Shared Memory API Functions and Uses

API Function	Uses
`shmget`	Create a new shared memory segment.
	Get the identifier for an existing shared memory segment.
`shmctl`	Get info on a shared memory segment.
	Set certain info on a shared memory segment.
	Remove a shared memory segment.
`shmat`	Attach to a shared memory segment.
`shmdt`	Detach from a shared memory segment.

The next sections address these API functions in detail, identifying each of their uses with sample source.

shmget

The shmget API function (like semget and msqget) is a multirole function. First, it can be used to create a new shared memory segment, and second, it can be used to get the ID of an existing shared memory segment. The result of the shmget API function (in either role) is a shared memory segment identifier that is to be used in all other shared memory functions. The prototype for the shmget function is defined as follows:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget( key_t key, size_t size, int shmflag );
```

The key argument specifies a system-wide identifier that uniquely identifies the shared memory segment. The key must be a nonzero value or the special symbol IPC_PRIVATE. The IPC_PRIVATE argument defines that you are creating a private segment (one that has no system-wide name). No other processes can find this segment, but it can be useful to create segments that are used only within a process or process group (where the return key can be communicated).

The size argument identifies the size of the shared memory segment to create. When you are interested in an existing shared memory segment, you leave this argument as zero (as it's not used by the function in the segment the segment "create" case). As a minimum, the size must be PAGE_SIZE (or 4 KB). The size should also be evenly divisible by PAGE_SIZE, as the segment allocated is in PAGE_SIZE chunks. The maximum size is implementation dependent, but typically is 4 MB.

The shmflag argument permits the specification of two separate parameters. These are a command and an optional set of access permissions. The command portion can take one of three forms. The first is to create a new shared memory segment, where the shmflag is equal to the IPC_CREAT symbol. This returns the identifier for a new segment or an identifier for an existing segment (if it already exists). If you want to create the segment and fail if the segment already exists, then you can use the IPC_CREAT with the IPC_EXCL symbol (second form). If (IPC_CREAT | IPC_EXCL) is used and the shared segment already exists, then an error status is returned, and errno is set to EEXIST. The final form simply requests an existing shared memory segment. In this case, you specify a value of zero for the command argument.

When you are creating a new shared memory segment, each of the read/write access permissions can be used except for the execute permissions. These permissions are shown in Table 18.2.

TABLE 18.2 Shared Memory Segment Permissions for `shmget` `msgflag` Argument

Symbol	Value	Meaning
`S_IRUSR`	0400	User read permission
`S_IWUSR`	0200	User write permission
`S_IRGRP`	0040	Group read permission
`S_IWGRP`	0020	Group write permission
`S_IROTH`	0004	Other read permission
`S_IWOTH`	0002	Other write permission

Now it's time to take a look at a few examples of the `shmget` function to create new shared memory segments or to access existing ones.

In this first example, you create a new private shared memory segment of size 4 KB. Note that because you're using `IPC_PRIVATE`, you are assured of creating a new segment as no unique key is provided. You also specify full read and write permission to all (system, group, and user).

```
shmid = shmget( IPC_PRIVATE, 4096, IPC_CREAT | 0666 );
```

If the `shmget` API function fails, a `–1` is returned (as `shmid`) with the actual error specified in the special `errno` variable (for this particular process).

Now take a look at the creation of a memory segment, with an error return if the segment already exists. In this example, your system-wide identifier (`key`) is `0x123`, and you request a 64 KB segment.

```
shmid = shmget( 0x123, (64 * 1024), (IPC_CREAT | IPC_EXCL | 0666)
);
if (shmid == -1) {
  printf( "shmget failed (%d)\n", errno );
}
```

Here you use the `IPC_CREAT` with `IPC_EXCL` to ensure that the segment doesn't exist. If you get a `-1` return from `shmget`, an error occurred (such as the segment already exists).

Creating system-wide keys with `ftok` was discussed in Chapter 16, "IPC with Message Queues," and Chapter 17, "Synchronization with Semaphores." Please refer to those chapters for a detailed discussion of file-based key creation.

Finally, take a look at a simple example of finding the shared memory identifier for an existing segment.

```
shmid = shmget( 0x123, 0, 0 );
if ( shmid != -1 ) {
  // Found our shared memory segment
}
```

Here you specify only the system-wide key; segment size and flags are both zero (as you are getting the segment, not creating it).

A final point to discuss with shared memory segments creation is the initialization of the shared memory data structure that is contained within the kernel. The shared memory structure is shown in Listing 18.7.

LISTING 18.7 The Shared Memory Structure (`shmid_ds`)

```
struct shmid_ds {
  struct ipc_perm shm_perm    /* Access permissions         */
  int    shm_segsz;           /* Segment size (in bytes)    */
  time_t shm_atime;           /* Last attach time (shmat)   */
  time_t shm_dtime;           /* Last detach time (shmdt)   */
  time_t shm_ctime;           /* Last change time (shmctl)  */
  unsigned short shm_cpid;    /* Pid of segment creator     */
  unsigned short shm_lpid;    /* Pid of last segment user   */
  short  shm_nattch;          /* Number of current attaches */
};
struct ipc_perm {
  key_t __key;
  unsigned short uid;
  unsigned short gid;
  unsigned short cuid;
  unsigned short cgid;
  unsigned short mode;
  unsigned short pad1;
  unsigned short __seq;
  unsigned short pad2;
  unsigned long int __unused1;
  unsigned long int __unused2;
};
```

Upon creation of a new shared memory segment, the `shm_perm` structure is initialized with the key and creator's user ID and group ID. Other initializations are shown in Table 18.3.

TABLE 18.3 Shared Memory Data Structure `init` on Creation

Field	Initialization
`shm_segsz`	Segment size provided to `shmget`
`shm_atime`	0
`shm_dtime`	0
`shm_ctime`	Current time
`shm_cpid`	Calling process's pid
`shm_lpid`	0
`shm_nattch`	0
`shm_perm.cuid`	Creator's process user ID
`shm_perm.gid`	Creator's process group ID

You will return to these elements shortly when you read about the control aspects of shared memory segments.

shmctl

The `shmctl` API function provides three separate functions. The first is to read the current shared memory structure (as defined at Listing 18.7) using the `IPC_STAT` command. The second is to write the shared memory structure using the `IPC_SET` command. Finally, a shared memory segment can be removed using the `IPC_RMID` command. The `shmctl` function prototype is shown as follows:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl( int shmid, int cmd, struct shmid_ds *buf );
```

You begin by removing a shared memory segment. To remove a segment, you first must have the shared memory identifier. You then pass this identifier, along with the command `IPC_RMID`, to `shmctl`, as follows:

```
int shmid, ret;
...
shmid = shmget( SHM_KEY, 0, 0 );
if ( shmid != -1 ) {
  ret = shmctl( shmid, IPC_RMID, 0 );
  if (ret == 0) {
```

```
        // shared memory segment successfully removed.
      }
    }
```

If no processes are currently attached to the shared memory segment, then the segment is removed. If processes are currently attached to the shared memory segment, then the segment is marked for deletion but not yet deleted. This means that only after the last process detaches from the segment is the segment removed. After the segment is marked for deletion, no processes can attach to the segment. Any attempt to attach results in an error return with `errno` set to `EIDRM`.

Internally, after the shared memory segment is removed, its key is changed to `IPC_PRIVATE`. *This disallows any new process from finding the segment.*

Next, take a look at the `IPC_STAT` command that can be used within `shmctl` to gather information about a shared memory segment. In Listing 18.7, you saw a number of parameters that define a shared memory segment. This structure can be read via the `IPC_STAT` command, as shown in Listing 18.8.

LISTING 18.8 Shared Memory Data Structure Elements Accessible Through `shmctl` (on the CD-ROM at `./source/ch18/shmstat.c`)

```
#include <stdio.h>
#include <sys/shm.h>
#include <errno.h>
#include <time.h>
#include "common.h"

int main()
{
  int shmid, ret;
  struct shmid_ds shmds;

  /* Create the shared memory segment using MY_SHM_ID */
  shmid = shmget( MY_SHM_ID, 0, 0 );

  if ( shmid >= 0 ) {

    ret = shmctl( shmid, IPC_STAT, &shmds );

    if (ret == 0) {

      printf( "Size of memory segment is %d\n",
```

```
22:                     shmds.shm_segsz );
23:          printf( "Number of attaches %d\n",
24:                     (int)shmds.shm_nattch );
25:          printf( "Create time %s",
26:                     ctime( &shmds.shm_ctime ) );
27:          if (shmds.shm_atime) {
28:            printf( "Last attach time %s",
29:                       ctime( &shmds.shm_atime ) );
30:          }
31:          if (shmds.shm_dtime) {
32:            printf( "Last detach time %s",
33:                       ctime( &shmds.shm_dtime ) );
34:          }
35:          printf( "Segment creation user %d\n",
36:                     shmds.shm_cpid );
37:          if (shmds.shm_lpid) {
38:            printf( "Last segment user %d\n",
39:                      shmds.shm_lpid );
40:          }
41:          printf( "Access permissions 0%o\n",
42:                     shmds.shm_perm.mode );
43:
44:        } else {
45:
46:          printf( "shmctl failed (%d)\n", errno );
47:
48:        }
49:
50:      } else {
51:
52:        printf( "Shared memory segment not found.\n" );
53:
54:      }
55:
56:      return 0;
57:    }
```

Listing 18.8 is rather self-explanatory. After getting your shared memory identifier at line 13, you use `shmctl` to grab the shared memory structure at line 17. Upon success of `shmctl`, you emit the various accessible data elements using `printf`. Note that at lines 27 and 31, you check that the time values are nonzero. If you find no attaches or detaches, the value is zero, and therefore you have no reason to convert it to a string time.

The final command available with `shmctl` is `IPC_SET`. This permits the caller to update certain elements of the shared memory segment data structure. These elements are shown in Table 18.4.

TABLE 18.4 Shared Memory Data Structure Writeable Elements

Field	Description
`shm_perm.uid`	Owner process effective user ID
`shm_perm.gid`	Owner process effective group ID
`shm_flags`	Access permissions
`shm_ctime`	Takes the current time of `shmctl.IPC_SET` action

The following code snippet illustrates setting new permissions (see Listing 18.9). It's important that the shared memory data structure be read first to get the current set of parameters.

LISTING 18.9 Changing Access Permissions in a Shared Memory Segment (on the CD-ROM at `./source/ch18/shmset.c`)

```
#include <stdio.h>
#include <sys/shm.h>
#include <errno.h>
#include <time.h>
#include "common.h"

int main()
{
  int shmid, ret;
  struct shmid_ds shmds;

  /* Create the shared memory segment using MY_SHM_ID */
  shmid = shmget( MY_SHM_ID, 0, 0 );

  if ( shmid >= 0 ) {

    ret = shmctl( shmid, IPC_STAT, &shmds );

    if (ret == 0) {

      printf("old permissions were 0%o\n",shmds.shm_perm.mode );

```

```
23:           shmds.shm_perm.mode = 0444;
24:
25:           ret = shmctl( shmid, IPC_SET, &shmds );
26:
27:           ret = shmctl( shmid, IPC_STAT, &shmds );
28:
29:           printf("new permissions are 0%o\n", shmds.shm_perm.mode );
30:
31:         } else {
32:
33:           printf( "shmctl failed (%d)\n", errno );
34:
35:         }
36:
37:       } else {
38:
39:         printf( "Shared memory segment not found.\n" );
40:
41:       }
42:
43:       return 0;
44:     }
```

In Listing 18.9, you grab the current data structure for the memory segment at line 17 and then change the mode at line 23. You write this back to the segment's data structure at line 25 using the `IPC_SET` command, and then you read it back out at line 27. Not very exciting, but the key to remember is to read the structure first. Otherwise, the effective user and group IDs are going to be incorrect, leading to anomalous behavior.

One final topic for shared memory control that differs from message queues and semaphores is the ability to lock down segments so that they're not candidates for swapping out of memory. This can be a performance benefit, because rather than the segment being swapped out to the filesystem, it stays in memory and is therefore available to applications without having to swap it back in. Therefore, this mechanism is very useful from a performance standpoint. The `shmctl` API function provides the means both to lock down a segment and also to unlock.

The following examples illustrate the lock and unlock of a shared memory segment:

```
int shmid;
...
shmid = shmget( MY_SHM_ID, 0, 0 );
ret = shmctl( shmid, SHM_LOCK, 0 );
```

```
if ( ret == 0 ) {
  printf( "Shared Memory Segment Locked down.\n" );
}
```

Unlocking the segment is very similar. Rather than specify SHM_LOCK, you instead use the SHM_UNLOCK symbolic, as follows:

```
ret = shmctl( shmid, SHM_UNLOCK, 0 );
```

As before, a zero return indicates success of the shmctl call. Only a superuser can perform this particular command via shmctl.

shmat

After the shared memory segment has been created, a process must attach to it to make it available within its address space. This is provided by the shmat API function. Its prototype is defined as follows:

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat( int shmid, const void *shmaddr, int shmflag );
```

The shmat function takes the shared memory segment identifier (returned by shmget), an address where the process wants to insert this segment in the process's address space (a desired address), and a set of flags. The desired address (shmaddr) is rounded down if the SHM_RND flag is set within shmflags. This option is rarely used because the process needs to have explicit knowledge of the available address regions within the process's address space. This method also is not entirely portable. To have the shmat API function automatically place the region within the process's address space, you pass a (const void *)NULL argument.

You can also specify the SHM_READONLY within shmflags to enforce a read-only policy on the segment for this particular process. This process must first have read permission on the segment. If SHM_READONLY is not specified, it is assumed that the segment is being mapped for both read and write. No write-only flag exists.

The return value from shmat is the address in which the shared memory segment is mapped into this process. A quick example of shmat is shown here:

```
int shmid;
void *myAddr;
/* Get the id for an existing shared memory segment */
shmid = shmget( MY_SHM_SEGMENT, 0, 0 );
/* Map the segment into our space */
myAddr = shmat( shmid, 0, 0 );
```

```
if ((int)myAddr != -1) {
  // Attach failed.
} else {
  // Attach succeeded.
}
```

Upon completion, `myAddr` contains an address in which the segment is attached or `-1`, indicating that the segment failed to be attached. The return address can then be utilized by the process just like any other address.

The local address into which the shared memory segment is mapped might be different for every process that attaches to it. Therefore, no process should assume that because another mapped at a given address, it is available at the same local address.

Upon successful completion of the `shmat` call, the shared memory data structure is updated as follows: the `shm_atime` field is updated with the current time (last attach time), `shm_lpid` is updated with the effective process ID for the calling process, and the `shm_nattch` field is incremented by 1 (the number of processes currently attached to the segment).

When a processes exits, its shared memory segments are automatically detached. Despite this, you should detach from your segments using `shmdt` rather than relying on GNU/Linux to do this for you. Also, when a process forks into a parent and child, the child inherits any shared memory segments that were created previously by the parent.

shmdt

The `shmdt` API function detaches an attached shared memory segment from a process. When a process no longer needs access to the memory, this function frees it and also unmaps the memory mapped into the process's local address space that was occupied by this segment. The function prototype for the `shmdt` function is as follows:

```
#include <sys/types.h>
#include <sys/shm.h>
int shmdt( const void *shmaddr );
```

The caller provides the address that was provided by `shmat` (as its return value). A return value of zero indicates a successful detach of the segment. Consider the following code snippet as a demonstration of the `shmdt` call:

```
int shmid;
void *myAddr;
/* Get the id for an existing shared memory segment */
shmid = shmget( MY_SHM_SEGMENT, 0, 0 );
/* Map the segment into our space */
myAddr = shmat( shmid, 0, 0 );
...
/* Detach (unmap) the segment */
ret = shmdt( myAddr );
if (ret == 0) {
  /* Segment detached */
}
```

Upon successful detach, the shared memory structure is updated as follows: the `shm_dtime` field is updated with the current time (of the `shmdt` call), the `shm_lpid` is updated with the process ID of the process calling `shmdt`, and finally, the `shm_nattach` field is decremented by 1.

The address region mapped by the shared memory segment is unavailable to the process and results in a segment violation if an access is attempted.

If the segment had been previously marked for deletion (via a prior call to `shmctl` with the command of `IPC_RMID`) and the number of current attaches is zero, then the segment is removed.

Using a Shared Memory Segment

Shared memory can be a powerful mechanism for communication and coordination between processes. With this power comes some complexity. Because shared memory is a resource that's available to all processes that attach to it, you must coordinate access to it. One mechanism is to simply add a semaphore to the shared memory segment. If the segment represents multiple contexts, multiple semaphores can be created, each coordinating its respective access to a portion of the segment.

Take a look at a simple example of coordinating access to a shared memory segment. Listing 18.10 illustrates a simple application that provides for creating, using, and removing a shared memory segment. As we have already covered the creation and removal aspects in detail (lines 31–58 for create and lines 137–158 for remove), the use scenario (lines 59–111) is what we focus on here.

This `block` (which represents your shared memory block) is `typdef`'d at lines 11–15. This contains your shared structure (`string`), a `counter` (as the index to your string), and your semaphore to coordinate access. Note that this is loaded into your shared structure at line 48.

The use scenario begins by grabbing the user character passed as the second argument from the command line. This is the character you place into the buffer on each pass. You invoke this process twice with different characters to see each access to the shared structure in a synchronized way. After getting the shared memory key (via `shmget` at line 69), you attach to the segment at line 72. The return value is the address of your shared block, which you cast to your block type (`MY_BLOCK_TYPE`). You then loop through a count of 2500, each iteration acquiring the semaphore, loading your character into the string array of the shared memory segment (your critical section), and then releasing the semaphore.

LISTING 18.10 Shared Memory Example Using Semaphore Coordination (on the CD-ROM at `./source/ch18/shmexpl.c`)

```
 1:    #include <stdio.h>
 2:    #include <sys/shm.h>
 3:    #include <sys/sem.h>
 4:    #include <string.h>
 5:    #include <stdlib.h>
 6:    #include <unistd.h>
 7:    #include "common.h"
 8:
 9:    #define MAX_STRING      5000
10:
11:    typedef struct {
12:      int semID;
13:      int counter;
14:      char string[MAX_STRING+1];
15:    } MY_BLOCK_T;
16:
17:
18:    int main( int argc, char *argv[] )
19:    {
20:      int shmid, ret, i;
21:      MY_BLOCK_T *block;
22:      struct sembuf sb;
23:      char user;
24:
25:      /* Make sure there's a command */
26:      if (argc >= 2) {
27:
28:        /* Create the shared memory segment and init it
29:         * with the semaphore
30:         */
```

```
31:         if (!strncmp( argv[1], "create", 6 )) {
32:
33:           /* Create the shared memory segment and semaphore */
34:
35:           printf("Creating the shared memory segment\n");
36:
37:           /* Create the shared memory segment */
38:           shmid = shmget( MY_SHM_ID,
39:                           sizeof(MY_BLOCK_T), (IPC_CREAT | 0666) );
40:
41:           /* Attach to the segment */
42:           block = (MY_BLOCK_T *)shmat( shmid, (const void *)0, 0 );
43:
44:           /* Initialize our write pointer */
45:           block->counter = 0;
46:
47:           /* Create the semaphore */
48:           block->semID = semget( MY_SEM_ID, 1, (IPC_CREAT | 0666) );
49:
50:           /* Increment the semaphore */
51:           sb.sem_num = 0;
52:           sb.sem_op = 1;
53:           sb.sem_flg = 0;
54:           semop( block->semID, &sb, 1 );
55:
56:           /* Now, detach from the segment */
57:           shmdt( (void *)block );
58:
59:         } else if (!strncmp( argv[1], "use", 3 )) {
60:
61:           /* Use the segment */
62:
63:           /* Must specify also a letter (to write to the buffer) */
64:           if (argc < 3) exit(-1);
65:
66:           user = (char)argv[2][0];
67:
68:           /* Grab the shared memory segment */
69:           shmid = shmget( MY_SHM_ID, 0, 0 );
70:
71:           /* Attach to it */
72:           block = (MY_BLOCK_T *)shmat( shmid, (const void *)0, 0 );
73:
```

```
74:           for (i = 0 ; i < 2500 ; i++) {
75:
76:             /* Give up the CPU temporarily */
77:             sleep(0);
78:
79:             /* Grab the semaphore */
80:             sb.sem_num = 0;
81:             sb.sem_op = -1;
82:             sb.sem_flg = 0;
83:             if ( semop( block->semID, &sb, 1 ) != -1 ) {
84:
85:               /* Write our letter to the segment buffer
86:                * (only if we have the semaphore).  This
87:                * is our critical section.
88:                */
89:               block->string[block->counter++] = user;
90:
91:               /* Release the semaphore */
92:               sb.sem_num = 0;
93:               sb.sem_op = 1;
94:               sb.sem_flg = 0;
95:               if ( semop( block->semID, &sb, 1 ) == -1 ) {
96:
97:                 printf("Failed to release the semaphore\n");
98:
99:               }'
100:
101:            } else {
102:
103:                printf("Failed to acquire the semaphore\n");
104:
105:            }
106:
107:          }
108:
109:          /* We're done, unmap the shared memory segment. */
110:          ret = shmdt( (void *)block );
111:
112:        } else if (!strncmp( argv[1], "read", 6 )) {
113:
114:          /* Here, we'll read the buffer in the shared segment */
115:
116:          shmid = shmget( MY_SHM_ID, 0, 0 );
117:
```

```
118:           if (shmid != -1) {
119:
120:             block = (MY_BLOCK_T *)shmat( shmid, (const void *)0, 0 );
121:
122:             /* Terminate the buffer */
123:             block->string[block->counter+1] = 0;
124:
125:             printf( "%s\n", block->string );
126:
127:             printf("length %d\n", block->counter);
128:
129:             ret = shmdt( (void *)block );
130:
131:           } else {
132:
133:             printf("Unable to read segment.\n");
134:
135:           }
136:
137:         } else if (!strncmp( argv[1], "remove", 6 )) {
138:
139:           shmid = shmget( MY_SHM_ID, 0, 0 );
140:
142:           if (shmid != -1) {
143:
144:             block = (MY_BLOCK_T *)shmat( shmid, (const void *)0, 0 );
145:
146:             /* Remove the semaphore */
147:             ret = semctl( block->semID, 0, IPC_RMID );
148:
149:             /* Remove the shared segment */
150:             ret = shmctl( shmid, IPC_RMID, 0 );
151:
152:             if (ret == 0) {
153:
154:               printf("Successfully removed the segment.\n");
155:
156:             }
157:
158:           }
159:
160:         } else {
161:
```

```
162:           printf( "Unknown command %s\n", argv[1] );
163:
164:         }
165:
166:       }
167:
168:       return 0;
169:     }
```

The key point of Listing 18.10 is that reading or writing from memory in a shared memory segment must be protected by a semaphore. Other structures can be represented in a shared segment, such as a message queue. The queue doesn't require any protection because it's protected internally.

Now take a look at a sample run of the application shown in Listing 18.10. You create your shared memory segment and then execute your use scenarios one after another (quickly). Note that you specify two different characters to differentiate which process had control for that position in the string. After the "use" process is complete, you use the `read` command to emit the string (a snippet is shown here).

```
$ ./shmexpl create
Creating the shared memory segment
$ ./shmexpl use a &
$ ./shmexpl use b &
[1] 18254
[2] 18255
[1]+ Done
[2]+ Done
$ ./shmexpl read
aaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbaaabbb...
length 5000
$ ./shmexpl remove
Successfully removed the segment.
$
```

Note that in some cases, you can see an entire string of `a`'s and then all the `b`'s. It all comes down to executing the use cases quickly enough so that they each compete for the shared resource.

User Utilities

GNU/Linux provides the `ipcs` command to explore IPC assets from the command line (including shared memory segments that are visible to the user). The `ipcs` utility provides information on shared memory segments as well as message queues and semaphores. You get a chance to investigate its use for shared memory segments here.

The general form of the `ipcs` utility for shared memory segments is as follows:

```
$ ipcs -m
```

This presents all of the shared memory segments that are visible to the process. For an example, you can start by creating a shared memory segment (as was shown previously in Listing 18.1):

```
$ ./shmcreate
Created a shared memory segment 163840
[mtj@camus ch18]$ ipcs -m
—— Shared Memory Segments ——
key        shmid     owner      perms      bytes      nattch     status
0x000003e7 163840    mtj        666        4096       0
$
```

You see here a new shared memory segment being available (`0x3e7 = 999`). Its size is 4,096 bytes, and you can see currently no attaches to this segment (`nattch = 0`). If you want to dig into this segment deeper, you can specify this shared memory segment specifically to `ipcs`. This is done with `ipcs` using the `-i` option:

```
$ ipcs -m -i 163840
Shared memory Segment shmid=163840
uid=500 gid=500 cuid=500        cgid=500
mode=0666       access_perms=0666
bytes=4096      lpid=0  cpid=15558      nattch=0
att_time=Not set
det_time=Not set
change_time=Thu May 20 11:44:44 2004
$
```

You now see some more detailed information, including the attach, detach, and change times; last process ID; created process ID; and so on.

Finally, you can remove the shared memory segment using the `ipcrm` command. To remove your previously created shared memory segment, you simply provide the shared memory identifier, as follows:

```
$ ipcrm -m 163840
$
```

Summary

This chapter introduced shared memory in GNU/Linux and the APIs that control its use. It first introduced the shared memory APIs as a quick review and then provided a more detailed view of the APIs. Because shared memory segments can be shared by multiple asynchronous processes, the chapter illustrated the protection of a shared memory segment with a semaphore. Finally, it reviewed the `ipcs` utility and demonstrated its use as a debugging tool, as well as the `ipcrm` utility for removing shared memory segments from the command line.

References

GNU/Linux `shmget`, `shmop` man pages

Shared Memory APIs

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget( key_t key, size_t size, int shmflag );
int shmctl( int shmid, int cmd, struct shmid_ds *buf );
void *shmat( int shmid, const void *shmaddr, int shmflag );
int shmdt( const void *shmaddr );
```

This page intentionally left blank

19 Advanced File Handling

In This Chapter

- Review of File Types and Attributes
- File System Traversal
- File Mapping
- File Event Notification
- Buffering and Syncing

Introduction

This chapter continues where Chapter 11, "File Handling in GNU/Linux," left off and explores some of the more advanced features of file handling. This includes a discussion of file types and their testing, special files, directory traversal, file system mapping, and many other operations.

Testing File Types

You can start with the simple task of determining a file's type. The type is the classification of the file (for example, whether it's a regular file, symbolic link, or one of the many special files). The GNU C library provides a number of functions that provide this classification, but first you need to retrieve status information of the file for the test using the `stat` command.

The `stat` command returns a number of different data elements about a file. This section focuses on the file type, and then some of the other elements are covered in the next section.

With the `stat` information, you use the `st_mode` field in particular to determine the file type. This field is a coding of the file type, but the GNU C library provides a set of macros that can be used to test for a given file type (returns 1 if the file is of the defined type, otherwise 0). These are shown in Table 19.1.

TABLE 19.1 File Test Macros for `stat`

Macro	Description (Returns 1 If the File Is of That Type)
`S_ISREG`	Regular file
`S_ISDIR`	Directory
`S_ISCHR`	Special character file
`S_ISBLK`	Special block file
`S_ISLNK`	Symbolic link
`S_ISFIFO`	Special FIFO file
`S_ISSOCK`	Socket

To use `stat`, you call it with a buffer that represents a structure of the information to be captured. Listing 19.1 provides a simple application that uses the filename provided as the argument and then uses `stat` and the file test macros to determine the file type.

LISTING 19.1 Using the File Test Macros to Determine the File Type (on the CD-ROM at `./source/ch19/ftype.c`)

```
 1:  int main( int argc, char *argv[] )
 2:  {
 3:    struct stat statbuf;
 4:
 5:    /* Check to see if a filename was provided */
 6:    if (argc < 2) {
 7:      printf("specify a file to test\n");
 8:      return 1;
 9:    }
10:
11:    /* Obtain stat information on the file */
12:    if ( stat(argv[1], &statbuf) == -1 ) {
13:      printf("Can't stat file\n");
14:      return 1;
```

```
15:     }
16:
17:     printf("%s ", argv[1]);
18:
19:     /* Determine the file type from the st_mode */
20:     if        (S_ISDIR(statbuf.st_mode)) {
21:       printf("is a directory\n");
22:     } else if (S_ISCHR(statbuf.st_mode)) {
23:       printf("is a character special file\n");
24:     } else if (S_ISBLK(statbuf.st_mode)) {
25:       printf("is a block special file\n");
26:     } else if (S_ISREG(statbuf.st_mode)) {
27:       printf("is a regular file\n");
28:     } else if (S_ISFIFO(statbuf.st_mode)) {
29:       printf("is a FIFO special file\n");
30:     } else if (S_ISSOCK(statbuf.st_mode)) {
31:       printf("is a socket\n");
32:     } else {
33:       printf("is unknown\n");
34:     }
35:
36:     return 0;
37:   }
```

Using this application is demonstrated in the following with a variety of file types:

```
$ ./ftype ..
.. is a directory
$ ./ftype /dev/mem
/dev/mem is a character special file
$ ./ftype /dev/fd0
a block special file
$ ./ftype /etc/resolv.conf
/etc/resolv.conf is a regular file
$
```

The `stat` command provides much more than just the file type through `st_mode`. The next section takes a look at some of the other attributes that are returned through `stat`.

Other `stat` information

The `stat` command provides a wealth of information about a file, including its size, owner, group, last modified time, and more. Listing 19.2 provides a simple program that implements the emission of the previously mentioned data elements, making use of other helper functions to convert the raw data into human-readable strings. In particular, the `strftime` function takes a time structure (standard time format) and converts it into a string. This function offers a large number of options and output formats.

LISTING 19.2 Using the `stat` Command to Capture File Data (on the CD-ROM at `./software/ch19/ftype2.c`)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
#include <langinfo.h>

int main( int argc, char *argv[] )
{
  struct stat    statbuf;
  struct passwd *pwd;
  struct group  *grp;
  struct tm     *tm;
  char           tmstr[257];

  /* Check to see if a filename was provided */
  if (argc < 2) {
    printf("specify a file to test\n");
    return 1;
  }

  /* Obtain stat information on the file */
  if ( stat(argv[1], &statbuf) == -1 ) {
    printf("Can't stat file\n");
    return 1;
  }

  printf("File Size : %-d\n", statbuf.st_size);

  pwd = getpwuid(statbuf.st_uid);
```

```
32:     if (pwd) printf("File Owner: %s\n", pwd->pw_name);
33:
34:     grp = getgrgid(statbuf.st_gid);
35:     if (grp) printf("File Group: %s\n", grp->gr_name);
36:
37:     tm = localtime(&statbuf.st_mtime);
38:     strftime(tmstr, sizeof(tmstr), nl_langinfo(D_T_FMT), tm);
39:     printf("File Date : %s\n", tmstr);
40:
41:     return 0;
42:   }
```

Executing the new `stat` application (called `ftype2`) produces the following for the `passwd` file:

```
$ ./ftype /etc/passwd
File Size : 1337
File Owner: root
File Group: root
File Date : Sun Jan  6 19:44:50 2008
$
```

The same information can be retrieved with the `fstat` command, although this command operates on a file descriptor instead of a filename.

Determining the Current Working Directory

You can learn the name of the current working directory, including the full path, with the `getcwd` command. Another function called `pathconf` enables you to determine the size of buffer necessary to pass to `getcwd`. This process is demonstrated in Listing 19.3. After the maximum path length is known, a buffer is allocated with `malloc` and then passed to `getcwd` to return the current working directory. After this is emitted to `stdout`, the buffer is freed.

LISTING 19.3 Getting the Current Working Directory with `getcwd` (on the CD-ROM at `./software/ch19/gcwd.c`)

```
1:  #include <stdio.h>
2:  #include <unistd.h>
3:  #include <stdlib.h>
4:
5:  int main()
6:  {
```

```
 7:     char *pathname;
 8:     long  maxbufsize;
 9:
10:     maxbufsize = pathconf( ".", _PC_PATH_MAX );
11:     if (maxbufsize == -1) return 1;
12:
13:     pathname = (char *)malloc(maxbufsize);
14:
15:     (void)getcwd( pathname, (size_t)maxbufsize );
16:
17:     printf("%s\n", pathname);
18:
19:     free(pathname);
20:
21:     return 0;
22:   }
```

Enumerating Directories

GNU/Linux provides variety of directory manipulation functions that are useful for enumerating a directory or an entire filesystem. Similar to operating on a file, you must first open the directory and then read its contents. In this case, the contents of the directory are the files themselves (which can be other directories).

To open a directory, you use the `opendir` API function. This returns a directory stream (a reference of type `DIR`). After the stream is opened, it can be read with the `readdir` function. Each call to `readdir` returns a directory entry or `NULL` if the enumeration is complete. When the directory stream is no longer needed, it should be closed with `closedir`.

The following simple application (Listing 19.4) illustrates using the API functions discussed so far (`opendir`, `readdir`, `closedir`). In this example, you enumerate a directory (specified on the command line) and emit the `stdout` only those directory entries that refer to regular files.

LISTING 19.4 Enumerating a Directory with `readdir` (on the CD-ROM at `./software/ch19/dirlist.c`)

```
1:  #include <stdio.h>
2:  #include <sys/types.h>
3:  #include <sys/stat.h>
4:  #include <dirent.h>
5:
6:  int main( int argc, char *argv[] )
```

```
 7:  {
 8:    DIR *dirp;
 9:    struct stat statbuf;
10:    struct dirent *dp;
11:
12:    if (argc < 2) return 1;
13:
14:    if (stat(argv[1], &statbuf) == -1) return 1;
15:
16:    if (!S_ISDIR(statbuf.st_mode)) return 1;
17:
18:    dirp = opendir( argv[1] );
19:
20:    while ((dp = readdir(dirp)) != NULL) {
21:
22:      if (stat(dp->d_name, &statbuf) == -1) continue;
23:      if (!S_ISREG(statbuf.st_mode)) continue;
24:
25:      printf("%s\n", dp->d_name);
26:
27:    }
28:
29:    closedir(dirp);
30:
31:    return 0;
32:  }
```

During enumeration, you can remember a position in the directory stream using `telldir`, and then return to this later with a call to `seekdir`. To return to the beginning of the stream, you use `rewinddir`.

Another operation for walking a directory is called `ftw` (which is an acronym for "file tree walk"). With this API function, you provide a directory to start the walk and then a function that is called for each file or directory found during the walk.

For each file that's encountered during the walk, a callback function is called and passed the filename, the `stat` structure, and a flag word that indicates the type of file (`FTW_D` for a directory, `FTW_F` for a file, `FTW_SL` for a symbolic link, `FTW_NS` for a file that couldn't be `stat`'ed, and `FTW_DNR` for a directory that could not be read).

An example of the use of `ftw` is shown in Listing 19.5. The `fncheck` function is the callback function called for each file encountered. The `main` function takes the argument passed as the starting directory to walk. This is provided to the `ftw` function along with the callback function and the `ndirs` argument. This argument

defines the number of directory streams that can be simultaneously open (better performance results with a larger number streams).

LISTING 19.5 Walking a Directory with `ftw` (on the CD-ROM at `./software/ch19/dwalk.c`)

```
 1:  #include <stdio.h>
 2:  #include <ftw.h>
 3:
 4:  int fncheck( const char *file, const struct stat *sb, int flag )
 5:  {
 6:    if (flag == FTW_F)
 7:      printf("File %s (%d)\n", file, (int)sb->st_size);
 8:    else if (flag == FTW_D)
 9:      printf("Directory %s\n", file);
10:
11:    return 0;
12:  }
13:
14:  int main( int argc, char *argv[] )
15:  {
16:    if (argc < 2) return 1;
17:
18:    ftw( argv[1], fncheck, 1 );
19:
20:    return 0;
21:  }
```

The final enumeration technique that this section explores is called globbing. The `glob` API function allows you to specify a regular expression representing the files to be returned. The results are stored in a glob buffer, which can be enumerated when the function completes. An example of the `glob` API function is provided in Listing 19.6. The first argument is the pattern (in this case files that end `.c` or `.h`). The second argument is one or more flags (for example `GLOB_APPEND` can be specified to append a new glob to a previous invocation). The third argument is an error function that is called when a directory is attempted to be opened but fails. Finally, the results are stored in the glob buffer (which entails internal allocation of memory). For this reason, you have to free this buffer with `globfree` when it's no longer needed.

This simple program (Listing 19.6) results in a printout of files that end in `.c` or `.h` in the current directory.

LISTING 19.6 Demonstrating the `glob` API Function (on the CD-ROM at `./software/ch19/globtest.c`)

```
 1:  #include <stdio.h>
 2:  #include <glob.h>
 3:
 4:  int main()
 5:  {
 6:    glob_t globbuf;
 7:    int    err, i;
 8:
 9:    err = glob( "*.[ch]", 0, NULL, &globbuf );
10:
11:    if (err == 0) {
12:
13:      for (i = 0 ; i < globbuf.gl_pathc ; i++) {
14:
15:        printf("%s\n", globbuf.gl_pathv[i]);
16:
17:      }
18:
19:      globfree( &globbuf );
20:
21:    }
22:
23:    return 0;
24:  }
```

GNU/Linux (the GNU C Library) provides a number of options for enumerating a subdirectory, whether you want to view all files, files based upon a regular expression, or through a callback means for each file.

File Event Notification with `inotify`

One of the new features in the 2.6 kernel provides support for notification of filesystem events. This new mechanism is called `inotify`, which refers to the fact that it provides for notification of inode events. This capability of the kernel has been used in a variety of applications, the most visible being the Beagle search utility. Beagle uses `inotify` to automatically notify of changes to the filesystem to perform indexing. The alternative is to periodically scan the filesystem for changes, which is very inefficient.

What's interesting about `inotify` is that it uses the file-based mechanisms to provide for notification about filesystem events. For example, when monitoring is

requested, events are provided to the user-space application through a file descriptor. The `inotify` API provides system calls that wrap these mechanisms. It's time to take a look at this capability in more detail.

Notification Process

The basic flow of `inotify` is first to provide a monitoring point. This can be a file or a directory. If a directory is provided (which could be an entire filesystem), then all elements of that directory are candidates for event monitoring. After a watchpoint has been defined, you can read from a file descriptor to receive events.

`inotify` Events

An event (whose structure is shown in Listing 19.7) provides an event type (`mask`), the name of the file affected (if applicable), and other elements.

LISTING 19.7 The `inotify_event` Structure

```
struct inotify_event
{
  int wd;               /* Watch descriptor.  */
  uint32_t mask;        /* Watch mask.  */
  uint32_t cookie;      /* Cookie to sync two events.  */
  uint32_t len;         /* Length (with NULs) of name.  */
  char name __flexarr;  /* Name.  */
};
```

The `mask` field shown in Listing 19.7 is the event tag that indicates the type of operation that was performed (and is being notified). The current list of events is provided in Table 19.2.

TABLE 19.2 Events Supported in the Current Version of `inotify`

Event Name	Value	Description
IN_ACCESS	0x00000001	File was accessed.
IN_MODIFY	0x00000002	File was modified.
IN_ATTRIB	0x00000004	Attributes were changed.
IN_CLOSE_WRITE	0x00000008	File was closed (writeable).
IN_CLOSE_NOWRITE	0x00000010	File was closed (non-writeable).
IN_CLOSE	0x00000018	File was closed (both).

→

Event Name	Value	Description
IN_OPEN	0x00000020	File was opened.
IN_MOVED_FROM	0x00000040	File was moved (from "name").
IN_MOVE_TO	0x00000080	File was moved (to "name").
IN_MOVE	0x000000C0	File was moved (either).
IN_CREATE	0x00000100	File was created.
IN_DELETE	0x00000200	File was deleted.
IN_DELETE_SELF	0x00000400	The file watchpoint was deleted.
IN_MOVE_SELF	0x00000800	The file watchpoint was moved.
IN_UNMOUNT	0x00002000	The filesystem containing file was unmounted.
IN_Q_OVERFLOW	0x00004000	Event queue overflow.
IN_ISDIR	0x40000000	Event file refers to a directory.
IN_ALL_EVENTS	0x00000FFF	All events.

Simple `inotify`-Based Application

Now it's time to take a look at a simple application that registers for events on a user-defined file and emits those events in a human-readable fashion. To begin you initialize `inotify` (which creates an instance of `inotify` for your application). This call to `inotify_init` returns a file descriptor that serves as your means to receive events from the kernel. You also register a watchpoint and then monitor the file (covered later). The `main` function is provided in Listing 19.8.

LISTING 19.8 The `main` Function for Your `inotify` Application (on the CD-ROM at `./software/ch19/dirwatch.c`)

```
 1:  #include <stdio.h>
 2:  #include <sys/inotify.h>
 3:  #include <errno.h>
 4:
 5:  int main( int argc, char *argv[] )
 6:  {
 7:    int ifd, err;
 8:
 9:    int register_watchpoint( int fd, char *dir );
10:    int watch( int fd );
11:
```

```
12:     if (argc < 2) return -1;
13:
14:     ifd = inotify_init();
15:
16:     if (ifd < 0) {
17:       printf("can't init inotify (%d)\n", errno);
18:       return -1;
19:     }
20:
21:     err = register_watchpoint( ifd, argv[1] );
22:     if (err < 0) {
23:       printf("can't add watch (%d)\n", errno);
24:       return -1;
25:     }
26:
27:     watch(ifd);
28:
29:     close(ifd);
30:
31:     return 0;
32:   }
```

Note that because events are posted from the kernel through a file descriptor, you can use the traditional mechanisms on that descriptor such as select to identify when an event is available (rather than sitting on the file).

Next, you have a look at the registration function (`register_watchpoint`). This function, shown in Listing 19.9, uses the `inotify_add_watch` API function to register for one or more events. As shown here, you specify the file descriptor (returned from `inotify_init`), the file to monitor (which was grabbed as the command-line argument), and then the event mask. Here you specify all events with the `IN_ALL_EVENTS` symbolic constant.

LISTING 19.9 Registering for an `inotify` Event (on the CD-ROM at `./software/ch19/dirwatch.c`)

```
1:  int register_watchpoint( int fd, char *dir )
2:  {
3:    int err;
4:
5:    err = inotify_add_watch( fd, dir, IN_ALL_EVENTS );
6:
7:    return err;
8:  }
```

Now that you've registered for events, they queue to your file descriptor. The `watch` function is used as your monitoring loop (as called from `main` and shown in Listing 19.10). In this function you declare a buffer for your incoming events (which can be multiple per invocation) and then enter your monitoring loop. You can call the `read` system call to wait for events (as this call blocks until an event is received). When this returns, you do some quick sanity checking and then walk through each event that's contained in the buffer. This event is emitted with a call to `emit_event`. Because events can vary in size (considering a variable length filename), you increment the index by the size of the event and the size of the accompanying file name (if present). The filename is indicated by a nonzero `len` field within the event structure. When all events have been emitted, you start again by sitting on the `read` function.

LISTING 19.10 The Event Monitoring Loop (on the CD-ROM at `./software/ch19/dirwatch.c`)

```
#define MAX_EVENTS      256

#define BUFFER_SIZE     (MAX_EVENTS * sizeof(struct inotify_event))

int watch( int fd )
{
  char ev_buffer[ BUFFER_SIZE ];
  struct inotify_event *ievent;
  int  len, i;

  void emit_event( struct inotify_event *ievent );

  while (1) {

    len = read( fd, ev_buffer, BUFFER_SIZE );

    if (len < 0) {
      if (errno == EINTR) continue;
    }

    i = 0;
    while (i < len) {

      ievent = (struct inotify_event *)&ev_buffer[i];

      emit_event( ievent );

```

```
28:         i += sizeof(struct inotify_event) + ievent->len;
29:
30:       }
31:
32:     }
33:
34:     return 0;
35:   }
```

The final function, `emit_event` (shown in Listing 19.11), parses the event structure and emits the information to `stdout`. First, you check to see if a string is provided, and if so, you emit this first. This name refers to the filename that was affected (directory or regular file). You then test each of the event tags to see which is present and then emit this to `stdout`.

LISTING 19.11 Parsing and Emitting the `event` Structure (on the CD-ROM at `./software/ch19/dirwatch.c`)

```
 1:  void emit_event( struct inotify_event *ievent )
 2:  {
 3:    if (ievent->len) printf("[%s] ", ievent->name);
 4:
 5:    if ( ievent->mask & IN_ACCESS ) printf("Accessed\n");
 6:    if ( ievent->mask & IN_MODIFY ) printf("Modified\n");
 7:    if ( ievent->mask & IN_ATTRIB ) printf("Attributes Changed\n");
 8:    if ( ievent->mask & IN_CLOSE_WRITE ) printf("Closed
       (writeable)\n");
 9:    if ( ievent->mask & IN_CLOSE_NOWRITE ) printf("Closed
       (unwriteable)\n");
10:    if ( ievent->mask & IN_OPEN ) printf("Opened\n");
11:    if ( ievent->mask & IN_MOVED_FROM ) printf("File moved from\n");
12:    if ( ievent->mask & IN_MOVED_TO ) printf("File moved to\n");
13:    if ( ievent->mask & IN_CREATE ) printf("Subfile created\n");
14:    if ( ievent->mask & IN_DELETE ) printf("Subfile deleted\n");
15:    if ( ievent->mask & IN_DELETE_SELF ) printf("Self deleted\n");
16:    if ( ievent->mask & IN_MOVE_SELF ) printf("Self moved\n");
17:
18:    return;
19:  }
```

Now take a look at an example use of this application on a simple directory (see Listing 19.12). Because this is performed in two different shells, the following example shows one shell as flush and the other shell as indented. On the left is the shell activity and on the right (indented) is the output of your `inotify` application.

You begin by creating the test directory (called `itest`) and then in another shell starting your `inotify` application (called `dirwatch`), specifying the directory to be monitored. You then perform a number of file operations and emit the resulting output by `dirwatch`. In the end, the directory itself is removed, which is detected and reported. At this point, no further monitoring can take place (so ideally the application should exit).

LISTING 19.12 Sample Output from the `inotify` Application

```
$ mkdir itest

    $ ./dirwatch /home/mtj/itest

$ touch itest/newfile

    [newfile] Subfile created
    [newfile] Opened
    [newfile] Attributes Changed
    [newfile] Closed (writeable)

$ ls itest
newfile

    Opened
    Closed (unwriteable)

$ mv itest/newfile itest/newerfile

    [newfile] File moved from
    [newerfile] File moved to

$ rm itest/newerfile

    [newerfile] Subfile deleted

$ rmdir itest

    Self deleted
```

With an API that is clean and simple to understand, the `inotify` event system is a great way to monitor filesystem events and to keep track of what's going on in the filesystem.

Removing Files from the Filesystem

To remove a file from a filesystem, you have a number of API functions available, but one is recommended because of its ability to work on either files or directories. The `remove` function removes a file, whether it is a regular file or directory. If the file to be removed is a regular file, then `remove` acts like `unlink`, but if the file is a directory, then `remove` acts like `rmdir`.

In Listing 19.13 you create a file in the filesystem using the `mkstemp` function. This returns a unique filename given a template provided by the caller (as shown, the `X` characters are returned with a unique signature). After this file is created, it is promptly closed and then deleted with the `remove` function.

LISTING 19.13 Creating and Removing a File with `mkstemp` and `remove` (on the CD-ROM at `./software/ch19/rmtest.c`)

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:
 4:  int main()
 5:  {
 6:    FILE *fp;
 7:    char filename[L_tmpnam+1] = "fileXXXXXX";
 8:    int err;
 9:
10:    err = mkstemp( filename );
11:
 2:    fp = fopen( filename, "w" );
13:    if (fp != NULL) {
14:
15:      fclose(fp);
16:
17:      err = remove( filename );
18:
19:      if (!err) printf("%s removed\n", filename);
20:
21:    }
22:
23:    return 0;
24:  }
```

Synchronizing Data

Older UNIX system users no doubt recall the `sync` command for synchronizing data to the disk. The `sync` command forces changed blocks that are buffered in the

kernel out to the disk. Normally, changed blocks are cached in the buffer cache within the kernel that makes them faster to retrieve if they're needed again. Periodically, these blocks are written out to the disk. This process also works for reads, where read data is cached within the buffer cache (making it faster for retrieval). A kernel process controls when read data is removed from the cache or when write data is pushed to the disk.

The GNU C Library provides three API functions for synchronizing data, depending upon your particular need: `sync`, `fsync`, and `fdatasync`. In most systems, `fsync` and `fdatasync` are equivalent.

The `sync` function (which can also be invoked at the command line) causes all pending writes to be scheduled for write on their intended mediums. This means that the write to the device might not be completed when the call returns, but is scheduled for write. Its prototype is as follows:

```
void sync( void );
```

If you're really just interested in a particular file descriptor, you can use the `fsync` API function. This function has the added bonus that it does not return until the data is on its intended medium. So if you're more interested in making sure that the data gets to the external device than you are in just scheduling it for write, `fsync` is your call. Its prototype is as follows:

```
int fsync( int filedes );
```

Note that `fsync` might return an error if an issue occurred (such as EIO, if a failure occurred while reading or writing to the device).

Summary

In addition to providing a rich and expressive file system interface, GNU/Linux provides a number of advanced features for file testing, directory traversal, and even event notification of filesystem changes. This chapter provided a quick summary of some of the more interesting filesystem features.

Advanced File Handling APIs

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

int stat( const char *restrict path,
struct stat *restrict buf );
```

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


#include <pwd.h>

struct passwd *getpwuid( uid_t uid );


#include <grp.h>

struct group *getgrgid( gid_t gid );


#include <time.h>
#include <langinfo.h>

size_t strftime( char *restric s, size_t maxsize,
                 const char *restrict format,
                 const struct tm *restrict timeptr );

char *nl_langinfo( nl_item item );


#include <unistd.h>

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

char *getcwd( char *buf, size_t size );


#include <dirent.h>

DIR *opendir( const char *dirname );

struct dirent *readdir( DIR *dirp );

int closedir( DIR *dirp );

long telldir( DIR *dirp );

void seekdir( DIR *dirp, long loc );

void rewinddir( DIR *dirp );


#include <ftw.h>
```

```
int ftw( const char *path, int (*fn)(const char *,
        const struct stat *ptr, int flag),
             int ndirs );


#include <glob.h>

int glob( const char *restrict pattern, int flags,
           int (*errfunc)(const char *epath, int errno),
           glob_t *restrict pglob );

void globfree( glob_t *pglob );


#include <sys/inotify.h>

int inotify_init( void );

int inotify_add_watch( int fd, const char *name,
uint32_t mask );

int inotify_rm_watch( int fd, uint32_t wd );


#include <stdio.h>

int remove( const char *path );

int mkstemp( char *template );


#include <unistd.h>

int unlink( const char *path );

int rmdir( const char *path );


#include <unistd.h>

void sync( void );

int fsync( int filedes );

int fdatasync( int filedes );
```

This page intentionally left blank

20 Other Application Development Topics

In This Chapter

- Parsing Command-Line Options with `getopt` and `getopt_long`
- Time Conversion Functions
- Gathering System-Level Information with `sysinfo`
- Mapping Physical Memory with `mmap`
- Locking and Unlocking Memory Pages for Performance
- Linux Error Reporting

Introduction

So far, a large number of topics relating to some of the more useful GNU/Linux service APIs have been discussed. This chapter now looks at a number of miscellaneous core APIs that complete the exploration of GNU/Linux application development. This includes the `getopt` function to parse command-line options, time and time conversion functions, physical memory mapping functions, and memory locking for high-performance applications.

The C language provides the means to pass command-line arguments into a program as it begins execution. The C `main` *function can accept two arguments,* `argv` *and* `argc`*. The* `argc` *argument defines the number of arguments that were passed in, whereas* `argv` *is a character pointer array (vector), containing an element per argument. For example,* `argv[0]` *is a character pointer to the first argument (the program name), and* `argv[argc-1]` *points to the last argument.*

Parsing Command-Line Options with `getopt` and `getopt_long`

The `getopt` function provides a simplified API for extracting command-line arguments and their options from the command line (which conforms to the POSIX standard for format and argument passing style). Most arguments take the following form:

```
<application> -f <f-arg>
```

where `-f` is a command-line option and `<f-arg>` is the option for `-f`. Function `getopt` can also handle much more complex argument arrangements, as you see in this section.

The function prototype for the `getopt` function is provided as:

```
#include <unistd.h>
int getopt( int argc, char * const argv[], const char *optstring );
extern char *optarg;

extern int optopt, optind;
```

The `getopt` function takes three arguments; the first two are the `argc` and `argv` arguments that are received through `main`. The third argument, `optstring`, represents the options specification. This consists of the options that you accept for the application. The option string has a special form. You define the characters that you accept as your options, and for each option that has an argument, you follow it with a `:`. Consider the following sample option string:

```
"abc:d"
```

This parses options such as `-a`, `-b`, `-d`, and also `-c <arg>`. You could also provide a double colon, such as `"abc::d"`, which tells `getopt` that `c` uses an optional argument.

The `getopt` function returns an `int` that represents the character option. With this, three external variables are also provided as part of the `getopt` API. These are `optarg`, `optopt`, and `optind`. The `optarg` variable points to an option argument and is used to extract the option when one is expected. The `optopt` variable specifies the option that is currently being processed. The return value of `getopt` represents the variable. When `getopt` is finished parsing (returns `-1`), the `optind` variable represents the index of those arguments on the command line that were not parsed. For example, if a set of arguments is provided on the command line without any `"-"` option designators, then these arguments can be retrieved via the `optind` argument (you will see an example of this shortly).

The application developer must ensure that all options required for the application are specified. The `getopt` *function provides the parsing aspect of command-line arguments, but the application must determine whether the options specified are accurate.*

Now take a look at an example of `getopt` that demonstrates the features that have been touched upon (see Listing 20.1). At line 8 you call `getopt` to get the next option. Note that you call it until you get a `-1` return, and it iterates through all of the options that are available. If `getopt` returns `-1`, you exit the loop (to line 36).

At line 10, you start at the `switch` construct to test the returns. If the `'h'` character is returned (line 12–14) you handle the `Help` option. At line 16, you handle the verbose option, which has an argument. Because you expect an integer argument after `-v`, you grab it using the `optarg` variable, passing it to `atoi` to convert it to an integer (line 17).

At line 20, you grab the `-f` argument (representing the filename). Because you are looking for a string argument, you can use `optarg` directly (line 21). At line 29, you test for any unrecognized options for which `getopt` returns `'?'`. You emit the actual option found with `optopt` (line 29).

Finally, at lines 38–42, you emit any options found that were not parsed using the `optind` variable. The `getopt` internally moves the nonoption arguments to the end of the `argv` argument list. Therefore, you can walk from `optind` to `argc` to find these.

LISTING 20.1 Example Use of `getopt` (on the CD-ROM at `./source/ch20/opttest.c`)

```
 1:    #include <unistd.h>
 2:    #include <stdio.h>
 3:
 4:    int main( int argc, char *argv[] )
 5:    {
 6:      int c;
 7:
 8:      while ( (c = getopt( argc, argv, "hv:f:d" ) ) != -1 ) {
 9:
10:        switch( c ) {
11:
12:          case 'h':
13:            printf( "Help menu.\n" );
14:            break;
15:
16:          case 'v':
17:            printf( "Verbose level = %d\n", atoi(optarg) );
```

```
18:             break;
19:
20:           case 'f':
21:             printf( "Filename is = %s\n", optarg );
22:             break;
23:
24:           case 'd':
25:             printf( "Debug mode\n" );
26:             break;
27:
28:           case '?':
29:             printf("Unrecognized option encountered -%c\n", optopt);
30:
31:           default:
32:             exit(-1);
33:
34:         }
35:
36:       }
37:
38:       for ( c = optind ; c < argc ; c++ ) {
39:
40:         printf( "Non option %s\n", argv[c] );
41:
42:       }
43:
44:
45:       /*
46:        *  Option parsing complete...
47:        */
48:
49:       return 0;
50:     }
```

Many new applications support not only short option arguments (such as `-a`) but also longer options (such as `–command=start`). The `getopt_long` function provides the application developer with the ability to parse both types of option arguments. The `getopt_long` function has the prototype:

```
#include <getopt.h<
int getopt_long( int argc, char * const argv[],
                    const char *optsring,
                    const struct option *longopts, int *longindex
);
```

The first three arguments (`argc`, `argv`, and `opstring`) mirror the `getopt` function. What differs for `getopt_long` are the final two arguments: `longopts` and `longindex`. The `longopts` argument is a structure that defines the set of long arguments you want to be parsed. This structure is defined as follows:

```
struct option {
  const char *name;
  int has_arg;
  int *flag;
  int val;
};
```

where `name` is the name of the long option (such as `command`) and `has_arg` represents the argument that might follow the option (`0` for no argument, `1` for a required option, and `2` for an optional argument). The `flag` reference determines how the return value is provided. If `flag` is not `NULL`, the return value is provided by the fourth argument, `val`; otherwise, the return value is returned by `getopt_long`.

Now take a look at an example of `getopt_long`. In this example, the application accepts the following arguments:

```
–start
–stop
–command <command>
```

As you will see, the `getopt_long` function is a perfect example of the use of a data structure to simplify the job of coding (see Listing 20.2).

The first item to note is that for the `getopt_long` function, you must include `getopt.h` (rather than `unistd.h`, as was done for `getopt`). The option data structure is defined at lines 4–9. At line 5, you define the element for the `–start` option. You define the name `start`, specify that it has no options, and then define the character that `getopt_long` returns after this option is found (`'s'`). The `–stop` option is defined similarly (but returns `'t'` on recognition). The `–command` option identifies a required argument to follow (as defined by `required_argument`) and returns `'c'` when found on the command line.

At lines 16 and 17, you see the call to `getopt_long`, which specifies the option string (`'stc:'`) and the options structure (`longopts`). The return value, like function `getopt`, is `-1` for no further options or a single character (as defined in the options structure).

When `–start` is encountered, an `'s'` is returned and handled at lines 21–23. The `–stop` option is found, a `'t'` is returned and handled at lines 25–27. When `–command` is parsed, `getopt_long` returns `'c'`, and you emit the command option at

line 30 using the `optarg` variable. Finally, you identify unrecognized options at lines 33–36 when `'?'` is returned from `getopt_long` (or an unknown option).

LISTING 20.2 Simple Example of `getopt_long` to Parse Command-Line Options (on the CD-ROM at `./source/ch20/optlong.c`)

```
 1:    #include <stdio.h<
 2:    #include <getopt.h<
 3:
 4:    static struct option longopts[] = {
 5:      { "start",   no_argument,        NULL,  's' },
 6:      { "stop",    no_argument,        NULL,  't' },
 7:      { "command", required_argument,  NULL,  'c' },
 8:      { NULL,      0,                  NULL,   0 }
 9:    };
10:
11:
12:    int main( int argc, char *argv[] )
13:    {
14:      int c;
15:
16:      while ( (c = getopt_long( argc, argv, "stc:",
17:                longopts, NULL)) != -1 ) {
18:
19:        switch( c ) {
20:
21:          case 's':
22:            printf( "Start!\n" );
23:            break;
24:
25:          case 't':
26:            printf( "Stop!\n" );
27:            break;
28:
29:          case 'c':
30:            printf( "Command %s!\n", optarg );
31:            break;
32:
33:          case '?':
34:          default:
35:            printf( "Unknown option\n");
36:            break;
37:
```

```
38:         }
39:
40:       }
41:
42:       return 0;
43:     }
```

Any application that requires command-line configurability can benefit from `getopt` or `getopt_long`.

Time API

GNU/Linux provides a wide variety of functions to deal with time (as in time of day). Time is commonly represented by the `tm` structure, which is identified as:

```
struct tm {
    int  tm_sec;   /* seconds          (0..59)           */
    int  tm_min;   /* minutes          (0..59)           */
    int  tm_hour;  /* hours            (0..23)           */
    int  tm_mday;  /* day of the month (1..31)           */
    int  tm_mon;   /* month            (1..12)           */
    int  tm_year;  /* year             (200x)            */
    int  tm_wday;  /* day of the week  (0..6, 0 = Monday */
    int  tm_yday;  /* day in the year  (1..366)          */
    int  tm_isdst; /* daylight savings time (0, 1, -1)   */
};
```

A simplified representation is defined as the `time_t` structure, which simply represents the time in seconds (since the epoch 00:00:00 UTC, January 1, 1970). The time functions that this section reviews are these:

```
#include <time.h<
time_t time( time_t *t );
struct tm *localtime( const time_t *timep );
struct tm *gmtime( const time_t *timep );
char *asctime( const struct tm *tm );
char *ctime( const time_t *timepDay  );
time_t mktime( struct tm *tm );
```

Grabbing the current time can be done with the `time` function, as follows:

```
time_t currentTime;
currentTime = time( NULL );
```

where NULL is passed to return the local time from the time function. The time can also be loaded into a variable by passing a time_t reference to the function:

```
time_t currentTime;
(void)time( &currentTime );
```

With this time stored, you can now convert it into the tm structure using the localtime function. Putting it together with time, you get:

```
time_t currentTime;
struct tm *tm_time;
currentTime = time( NULL );
tm_time = localtime( &currentTime );
printf( "%02d:%02d:%02d\n",
 tm_time-<tm_hour, tm_time-<tm_min, tm_time-<tm_sec );
```

Converting time to an ASCII string is easily provided using the asctime or ctime function. The ctime function takes a time_t reference, whereas asctime takes a tm structure, as follows:

```
time_t currentTime;
struct tm *tm_time;
currentTime = time( NULL );
printf("%s\n", ctime( &currentTime ) );
tm_time = localtime( &currentTime );
printf("%s\n", asctime( tm_time ) );
```

The gmtime function breaks down a time_t variable into a tm structure, but in Coordinated Universal Time (UTC). This is the same as GMT (Greenwich Mean Time). The gmtime function is illustrated as follows:

```
tm_time = gmtime( &currentTime );
```

Finally, the mktime function converts the tm structure into the time_t format. It is demonstrated as follows:

```
tm_time = gmtime( &currentTime );
```

The entire set of functions is illustrated in the simple application shown in Listing 20.3.

LISTING 20.3 Demonstration of Time Conversion Functions (on the CD-ROM at `./source/ch20/time.c`)

```
 1:    #include <time.h<
 2:    #include <stdio.h<
 3:
 4:    int main()
 5:    {
 6:      time_t currentTime;
 7:      struct tm *tm_time;
 8:
 9:      currentTime = time( NULL );
10:      tm_time = localtime( &currentTime );
11:
12:      printf( "from localtime %02d:%02d:%02d\n",
13:              tm_time-<tm_hour, tm_time-<tm_min, tm_time-<tm_sec );
14:
15:      printf( "from ctime %s\n", ctime( &currentTime ) );
16:
17:      printf( "from asctime/localtime %s\n", asctime( tm_time ) );
18:
19:      tm_time = gmtime( &currentTime );
20:
21:      printf( "from gmtime %02d:%02d:%02d\n",
22:              tm_time-<tm_hour, tm_time-<tm_min, tm_time-<tm_sec );
23:
24:      printf( "from asctime/gmtime %s\n", asctime( tm_time ) );
25:
26:      currentTime = mktime( tm_time );
27:
28:      printf( "from ctime/mktime %s\n", ctime( &currentTime ) );
29:
30:      return 0;
31:    }
```

Executing this application yields the following result:

```
$ ./time

from localtime 22:53:02
from ctime Tue Jun  1 22:53:02 2004

from asctime/localtime Tue Jun  1 22:53:02 2004
```

```
from gmtime 04:53:02
from asctime/gmtime Wed Jun  2 04:53:02 2004

from ctime/mktime Wed Jun  2 05:53:02 2004

$
```

Gathering System Information with `sysinfo`

The `sysinfo` command allows an application to gather high-level information about a system, some of it very useful. The API for the `sysinfo` command is as follows:

```
int sysinfo( struct sysinfo *info );
```

The `sysinfo` command returns zero on success and fills the `sysinfo` structure as defined by Table 20.1. Note that all sizes are provided in the units defined by `mem_unit`.

TABLE 20.1 Elements and Meaning for `struct sysinfo`

Element	Description
`uptime`	The current uptime of this system in seconds
`loads[0]`	System load average for 1 minute
`loads[1]`	System load average for 5 minutes
`loads[2]`	System load average for 15 minutes
`totalram`	Total usable main memory
`freeram`	Available main memory
`sharedram`	Amount of memory that's shared
`bufferram`	Amount of memory used by buffers
`totalswap`	Total swap space
`freeswap`	Free swap space
`procs`	Number of currently active processes
`totalhigh`	Total amount of high memory
`freehigh`	Free amount of high memory
`mem_unit`	Memory unit size in bytes

Gathering the system information is very simple, as illustrated in Listing 20.4. Note that `uptime` has been further decomposed to provide a more meaningful representation (line 16–25).

LISTING 20.4 Sample Use of `sysinfo` Function (on the CD-ROM at `./source/ch20/sysinfo.c`)

```
 1:    #include <sys/sysinfo.h<
 2:    #include <stdio.h<
 3:
 4:    int main()
 5:    {
 6:      struct sysinfo info;
 7:      int ret;
 8:      int days, hours, minutes, seconds;
 9:
10:      ret = sysinfo( &info );
11:
12:      if (ret == 0) {
13:
14:        printf( "Uptime is %ld\n", info.uptime );
15:
16:        days = info.uptime / (24 * 60 * 60);
17:        info.uptime -= (days * (24 * 60 * 60));
18:        hours = info.uptime / (60 * 60);
19:        info.uptime -= (hours * (60 * 60));
20:        minutes = info.uptime / 60;
21:        info.uptime -= (minutes * 60);
22:        seconds = info.uptime;
23:
24:        printf( "Uptime %d Days %d Hours %d Minutes %d Seconds\n",
25:                days, hours, minutes, seconds );
26:        printf( "One minute load average %ld\n", info.loads[0] );
27:        printf( "Five minute load average %ld\n", info.loads[1] );
28:        printf( "Fifteen minute load average %ld\n", info.loads[2] );
29:        printf( "Total Ram Available %ld\n", info.totalram );
30:        printf( "Free Ram Available %ld\n", info.freeram );
31:        printf( "Shared Ram Available %ld\n", info.sharedram );
32:        printf( "Buffer Ram Available %ld\n", info.bufferram );
33:        printf( "Total Swap Size %ld\n", info.totalswap );
34:        printf( "Available Swap Size %ld\n", info.freeswap );
35:        printf( "Processes running: %d\n", info.procs );
36:        printf( "Total high memory size %ld\n", info.totalhigh );
37:        printf( "Available high memory %ld\n", info.freehigh );
```

```
38:         printf( "Memory Unit size %d\n", info.mem_unit );
39:       }
40:
41:       return 0;
42:     }
```

Much of this information can also be gathered from the `/proc` filesystem, but that is more difficult because of the parsing that's necessary. Some information is provided in `/proc/uptime`, `/proc/meminfo`, and `/proc/loadavg`.

The `proc` *filesystem is a virtual filesystem that contains runtime information about the state of the operating system. The* `proc` *filesystem can be used to inquire about various features by "*`cat`*"ing files in the* `proc` *filesystem. For example, you can identify all processes in the system (*`/proc/#`*), information about the CPU (*`/proc/cpuinfo`*), the devices found on the PCI buses (*`/proc/pci`*), the kernel modules currently loaded (*`/proc/modules`*), and much more runtime information.*

Mapping Memory with `mmap`

While not completely related to shared memory, the `mmap` API function provides the means to map file contents into user program space. The prototype function for `mmap` (and `munmap` to unmap the memory) is defined as follows:

```
#include <sys/mman.h>
void *mmap( void *start, size_t length, int prot, int flags,
                int fd, off_t offset );
int munmap( void *start, size_t length );
```

The `mmap` function takes in a file byte offset (`offset`) and tries to map it to the address defined by the caller (`start`) from the file descriptor `fd` (you will look at what this means shortly). Commonly, the `start` address is defined as `NULL`, allowing `mmap` to map it to whatever local address it chooses. This address (local mapping) is returned by the `mmap` function. The length of the region is defined by `length`. The caller defines the desired memory protection through the `prot` argument, which can be `PROT_EXEC` (region can be executed), `PROT_READ` (region can be read), `PROT_WRITE` (region can be written), or `PROT_NONE` (pages can't be accessed). Combinations of protections can be defined. Finally, the type of mapped object is defined by the `flags` argument. This can be `MAP_FIXED` (fail if the `start` address can't be used for the local mapping), `MAP_SHARED` (share this mapping with other processes), or `MAP_PRIVATE` (create a private copy-on-write mapping). The caller must specify `MAP_SHARED` or `MAP_PRIVATE`.

The `offset` *and* `start` *arguments must be on page boundaries. The* `length` *argument should be a multiple of the page size.*

GNU/Linux also provides some other nonstandard flags that are defined in Table 20.2.

TABLE 20.2 Nonstandard Flags for `mmap`

Flag	Use
`MAP_NORESERVE`	Used with `MAP_PRIVATE` to instruct the kernel not to reserve swap space for this region.
`MAP_GROWSDOWN`	Used by the kernel for stack allocation.
`MAP_ANONYMOUS`	This region is not backed by a file (`fd` and offset arguments of `mmap` are therefore ignored).

The `munmap` function simply unmaps the memory mapped by `mmap`. To `munmap`, the address returned by `mmap` is provided along with the length (which was also specified to `mmap`).

Take a look at an example of `mmap` and `munmap`. In Listing 20.5, you find an application that maps physical memory and makes it available for read. This application first creates a file descriptor of the file `/dev/mem`, which represents the physical memory available.

It's important to note that `/dev/mem` should be used only for read operations. Writing can be dangerous to the point of crashing your system. Finally, it's also a large security hole; therefore, its use should be avoided (though it does require root access).

The sample application allows the base address and length to be defined on the command line (lines 15–32). At line 34, you open `/dev/mem` for read. The file `/dev/mem` permits access to physical memory space (for which you use `mmap` to map physical memory into the process's memory space). At lines 38–40, you use `mmap` to map the requested region into the process's address space. The return value is the address from which the memory can be accessed. You then perform a loop to read and `printf` the addresses and their contents (lines 45–57). You finally clean up by unmapping the memory with `munmap` (line 59) and then closing the `/dev/mem` file descriptor at line 67.

LISTING 20.5 Mapping Physical Memory with `mmap` (on the CD-ROM at `./source/ch20/phymap.c`)

```
 1:    #include <stdio.h>
 2:    #include <unistd.h>
 3:    #include <fcntl.h>
 4:    #include <stdlib.h>
 5:    #include <errno.h>
 6:    #include <sys/mman.h>
 7:
 8:    int main( int argc, char *argv[] )
 9:    {
10:      int fd;
11:      unsigned char *addr, *waddr, count;
12:      int length;
13:      off_t offset;
14:
15:      if (argc < 3) {
16:
17:        printf( "Usage is phymap <address< <length<\n");
18:        exit(-1);
19:
20:      }
21:
22:      if (argv[1][1] == 'x') {
23:        sscanf( argv[1], "0x%x", &offset );
24:      } else {
25:        sscanf( argv[1], "%d", &offset );
26:      }
27:
28:      if (argv[2][1] == 'x') {
29:        sscanf( argv[2], "0x%x", &length );
30:      } else {
31:        sscanf( argv[2], "%d", &length );
32:      }
33:
34:      fd = open( "/dev/mem", O_RDONLY );
35:
36:      if (fd != -1) {
37:
38:        addr = (unsigned char *)mmap( NULL, length,
39:                                      PROT_READ, MAP_SHARED,
40:                                      fd, offset );
41:
```

```
42:
43:         if (addr != NULL) {
44:
45:           waddr = addr;
46:
47:           for ( count = 0 ; count < length ; count++ ) {
48:
49:             if ( (count % 16) == 0 ) {
50:
51:               printf( "\n%8p : ", waddr );
52:
53:             }
54:
55:             printf( "%02x ", *waddr++ );
56:
57:           }
58:
59:           munmap( addr, length );
60:
61:         } else {
62:
63:           printf("Unable to map memory.\n");
64:
65:         }
66:
67:         close(fd);
68:         printf("\n");
69:
70:       }
71:
72:       return 0;
73:     }
```

Using this application (call it `phymap`) is illustrated in the following example. Here you peek at the address `0x000c1000` for 64 bytes.

```
# ./phymap 0x000c1000 64
addr = 0x40017000 (Success)

0x40017000 : 56 57 a0 49 04 e8 c0 fe 32 ed 41 c1 e9 03 32 e4
0x40017010 : e8 0d 67 2e 8b b5 e6 08 2e 8a 44 0c d1 e8 f7 e1
0x40017020 : 5f 5e 07 5a 59 c3 33 c9 a4 fe c1 8a 2c 0a ed 75
0x40017030 : f7 fe c1 a4 32 ed c3 50 53 32 e4 b0 11 b3 80 e8
#
```

The address shown here is the local address that's been mapped in your address space. While it's different than your requested `0x000c1000`, the `0x40017000` represents the same region for our process.

Locking and Unlocking Memory

In this section, you take a look at an additional set of functions that are very useful to high-performance applications. The memory-locking functions permit a process to lock some or all of its storage that it is never swapped out of memory. The result is greater performance for the application, because it never has to suffer paging penalties in a busy system, but this does require the proper permissions.

The functions of interest for locking and unlocking memory are as follows:

```
#include <sys/mman.h>
int mlock( const void *addr, size_t len );
int munlock( const void *addr, size_t len );
int mlockall( int flags );
int munlockall( void );
```

The `addr` and `len` arguments must be on page boundaries. Now take a look at the lock/unlock pairs of functions in detail.

The `mlock` function takes an address (`addr`) for which the memory pages that represent the region are to be locked. The `len` argument defines the size of the region to lock (meaning that one or more pages might be locked by the operation). A return value of zero means that the pages are locked. When the application is finished with the memory, a call to `munlock` makes the pages available for swapping. A return of zero represents success of the unlock system call.

The following code example illustrates locking a page of memory and then unlocking it (see Listing 20.6). The buffer here is a locally created array of characters.

LISTING 20.6 Locking and Unlocking a Memory Page (on the CD-ROM at `./source/ch20/lock.c`)

```
1:    #include <stdio.h>
2:    #include <sys/mman.h>
3:
4:    char data[4096];
5:
6:    int main()
7:    {
8:      int ret;
9:
```

```
10:       ret = mlock( &data, 1024 );
11:
12:       printf("mlock ret = %d\n", ret);
13:
14:       ret = munlock( &data, 1024 );
15:
16:       printf("munlock ret = %d\n", ret);
17:
18:       return 0;
19:     }
```

At line 10, you call `mlock` with a reference to the global buffer (`data`) and its length. You unlock this page by calling `mlock` with your buffer and size (identically to the call to `mlock`). The entire page containing the buffer is locked, in the event it falls under the bounds of a page (or two pages if the address spans two)

Child processes do not inherit memory locks (created by `mlock` *or* `mlockall`*). Therefore, if a region of memory is created and subprocesses are also created to operate upon it, each child process should perform its own* `mlock`*.*

The `mlockall` API function locks all memory (disables paging) for a process's entire memory space. This includes not only the code and data for the process, but also its stack, shared libraries, shared memory, and other memory mapped files.

The `mlockall` function takes a single argument, which represents the scope of the lock to be provided. The user can specify `MCL_CURRENT`, which locks all pages that are currently mapped into the address space of the process, or `MCL_FUTURE`, which locks all pages that will be mapped into the address space of the process in the future. For example:

```
/* Lock currently mapped pages */
mlockall( MCL_CURRENT);

/* Lock all future pages */
mlockall( MCL_FUTURE );
```

You can also define that all current and future pages are locked into memory by performing a bitwise OR to put the flags together, as follows:

```
mlockall( MCL_CURRENT | MCL_FUTURE );
```

If insufficient memory is available to lock the current set of pages, an `ENOMEM` error is returned. If `MCL_FUTURE` is used, and insufficient memory exists to lock a

growing process stack, the kernel throws a SIGSEGV (segment violation) signal for the process, causing it to terminate.

The munlockall system call reenables paging for the pages mapped for the calling process. It takes no arguments and returns zero on success:

```
/* Unlock the pages for this process */
munlockall();
```

The munlockall system call should be called after the process has completed its real-time processing.

Linux Error Reporting

In Chapter 11, "File Handling in GNU/Linux," you first encountered the error-reporting mechanism of Linux called the errno variable. This variable is the last error that occurred for a given process (or thread, where each thread has its own errno variable). If another error occurs, the errno variable is overwritten. For this reason, after an application detects an error (such as a non-zero return from a system call), the errno variable should be checked and the error handled.

When using the errno variable, you must first make this variable available to your application by including the errno.h header file. Also defined here (or in a file included by this header file) are the various error code symbolic constants.

Each system call presents a different set of errors to the user. Every system call that returns one or more errors presents those (and the condition that produced them) in the respective man pages.

Now take a look at an example of using the errno variable in a simple application (see Listing 20.7). In this example, you include the necessary header file at line 2. Lines 4–8 are set up to check the number of input arguments and then open the first string argument as a file. Line 10 is the target of the error processing (opening a file). If the file returns a NULL (checked at line 12), then you know an error occurred. The errno variable is checked at lines 14–28 (specifically, two known errors are checked, EACCES and EISDIR) and a string is emitted to notify the user of the error.

LISTING 20.7 Demonstrating the Usage of errno

```
1:  #include <stdio.h>
2:  #include <errno.h>
3:
4:  int main( int argc, char *argv[] )
5:  {
```

```
 6:    FILE *fp;
 7:
 8:    if (argc < 2) return 1;
 9:
10:    fp = fopen( argv[1], "w" );
11:
12:    if (fp == NULL) {
13:
14:      switch(errno) {
15:
16:        case EACCES:
17:          printf("Can't access file.\n");
18:          break;
19:
20:        case EISDIR:
21:          printf("Can't operate on directory.\n");
22:          break;
23:
24:        default:
25:          printf("another error occurred.\n");
26:          break;
27:
28:      }
29:
30:      return 1;
31:
32:    }
33:
34:    fclose(fp);
35:
36:    return 0;
37:  }
```

In most cases, everything shown in Listing 20.7 is not necessary, but instead all that is required is just an indication of the error that occurred (particulary if you have no way to recover from the error). In this case, emitting the error code is the desired result. In these cases, you can use a simple line such as:

```
printf("Error occurred: %d\n", errno);
```

But with around 130 unique error codes, what is even better is a string representing the error type. The GNU C Library provides a few functions exactly for this purpose. The prototypes for these functions can be found in `string.h`.

The first is the sterror function, which takes an error code and emits a descriptive string. Its prototype is as follows:

```
char *strerror( int errnum );
char *strerror_r( int errnum, char *buf, size_t n );
```

So instead of simply printing out the error code, you can emit the string associated with the error code, as follows:

```
printf("%s\n", strerror(errno));
```

If the error is EACCES, then the following string is emitted:

```
Permission denied
```

If you're working in a threaded application, strerror can't be used because it statically allocates the string for the process. For each thread, you want to provide your own buffer instead. This is why the function prototype for strerror_r accepts the error code, buffer, and buffer size. The function also returns a pointer to the user-provided buffer (acting identically to the strerror function).

Finally, you have the perror function. This one is particularly useful because it simply emits the error message to stderr. Its prototype is as follows:

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

The function takes a message that is prepended to the output (typically the function that caused the errno to be set is passed in). Similar to the last example, perror can be used as follows:

```
perror("fopen");
```

For the EACCES error, this results in:

```
fopen: Permission denied
```

The one thing to remember about errno is that its state can be set in every system call. Therefore, if you want to maintain knowledge of a given error code, you need to store it before calling other system calls (which alter it). Further, only consider the errno variable valid after a system call returns an error.

Summary

This chapter covered a number of system calls that are useful in the development of application and tools software. These included the `getopt` and `getopt_long` calls for parsing command-line arguments, a variety of time and time conversion functions, the `sysinfo` call to gather high-level system information, the `mmap` system call to map files, two pairs of page-locking functions to help build high-performance applications (by avoiding page swapping penalties), and finally routines for emitting error codes and descriptive strings.

API Summary

```
#include <unistd.h>
int getopt( int argc, char * const argv[], const char *optstring );
extern char *optarg;
extern int optopt, optind;

#include <getopt.h>
int getopt_long( int argc, char * const argv[],
                    const char *optsring,
                    const struct option *longopts, int *longindex );
extern char *optarg
extern int optopt, optind;

#include <time.h>
time_t time( time_t *t );
struct tm *localtime( const time_t *timep );
struct tm *gmtime( const time_t *timep );
char *asctime( const struct tm *tm );
char *ctime( const time_t *timepDay  );
time_t mktime( struct tm *tm );


#include <sys/mman.h>
void *mmap( void *start, size_t length, int prot, int flags,
               int fd, off_t offset );
int munmap( void *start, size_t length );
int mlock( const void *addr, size_t len );
int munlock( const void *addr, size_t len );
int mlockall( int flags );
int munlockall( void );
```

```
#include <errno.h>
#include <string.h>

    char *strerror( int errnum );
    char *strerror_r( int errnum, char *buf, size_t n );

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

Part IV GNU/Linux Shells and Scripting

This part of the book looks at the topic of scripting languages. While scripting languages are historically tied to shells, scripting is a much larger topic. This part covers Bash scripting, domain-specific languages such as `sed` and `awk`, and finally building interpreters with `flex` and `bison`.

Chapter 21: Standard GNU/Linux Commands

GNU/Linux includes a large set of commands that aid in software development. We'll look at a variety of the most commonly used commands and cover them in a tutorial fashion, including numerous examples.

Chapter 22: Bourne-Again Shell (Bash)

The `bash` shell is the de facto standard shell for GNU/Linux. Shell programming is very beneficial to understand because it permits developers to code repetitive tasks quickly. Shell programming can be slower than compiling to the native instruction set, but efficiency isn't always the most important aspect of development. Compiling source to a native image takes time. The great advantage to scripting is that you can execute your script immediately. This makes scripting languages perfect for prototyping. Many scripting languages also allow the script to be performed interactively, which makes it much easier for them to be understood.

Chapter 23: Editing with sed

The sed utility (stream editor) is a noninteractive text editing utility that is quite useful in performing global editing of files. sed can be used on a single file or many files, and although it is cryptic, it is generally a useful tool that solves a number of text editing problems. This chapter introduces sed and describes its use in a number of simple and complex examples.

Chapter 24: Text Processing with awk

While some consider the awk utility an advancement of sed, it is in its own right a high-level procedural programming language. awk was designed for text pattern processing on files, but it has evolved into a compact language that is useful in a number of areas. awk is very convenient in prototyping because it is interpreted, and programs can be developed and tested quickly. The awk programming language is useful and can be very beneficial to GNU/Linux developers.

Chapter 25: Parser Generation with flex and bison

The development of parsers was traditionally limited to those with extensive experience in parsing and compiler theory. With the introduction of lex and yacc (and the GNU replacements, flex and bison), building parsers is much simplified as the difficult work is done for you. The flex tool is a lexical analyzer generator that creates programs for tokenization of input files. The bison tool, working in concert with the lexer, generates a grammar parser to validate the correctness of input tokens. Each of these tools takes an input file defining the token structure and grammar and produces C source to perform tasks. This makes the development of parsers simple and very maintainable. In this chapter, flex and bison are introduced through multiple examples of tokenization and grammar parsing.

Chapter 26: Scripting with Ruby

Development of applications on GNU/Linux isn't restricted to high-level languages. Applications can instead be written in interpreted languages such as Ruby. Ruby is one of the newer object-oriented scripting languages from which you can develop applications. This chapter introduces Ruby, its syntax, and its useful and unique features.

Chapter 27: Scripting with Python

Continuing from the previous chapter, this chapter explores the Python language for GNU/Linux development. Python is one of the older object-oriented scripting languages that is powerful and very popular. Python is introduced from the perspective of syntax and unique features.

Chapter 28: GNU/Linux Administration Basics

Development on GNU/Linux implies a certain amount of administration. Whether it's upgrading the Linux kernel or installing new software package (from source, or through a package management utility), you face administration tasks that all developers need to understand. This chapter introduces you to some of the more fundamental administration tasks.

This page intentionally left blank

21 Standard GNU/Linux Commands

In This Chapter

- Standard In, Out, and Error
- Invoking Shell Scripts
- Redirection
- Discussion of Important GNU/Linux Commands

Introduction

This chapter looks at some of the basics of standard GNU/Linux shells in addition to the important commands that are used frequently.

Redirection

First take a look at a basic topic in GNU/Linux shell use, that of input or output redirection.

The concept of redirection is simply that of redirecting your input or output to something other than the default. For example, the standard output of most commands is redirected to the shell window. You can redirect the output of a command to a file using the > output redirection symbol. For example:

```
ls -la
```

generates a file listing and emits the results to the shell window. You can instead redirect this to a file as follows:

```
ls -la > ls-out.txt
```

Now what would have been emitted to the shell window is now present in the output file `ls-out.txt`.

Rather than accept input from the keyboard, you can accept input from another source. For example, the command `cat` simply emits its standard input (or files named on its command line) to its standard out. You can redirect the contents of a file to `cat` using the < input redirection symbol.

```
cat < ls-out.txt
```

You can also build more complicated redirection structures. For example, using the `|` (pipe) symbol, you can chain a number of generators and filters together. For example, consider the following command sequence:

```
find . -name '*.[ch]' -print | xargs grep "mtj" | more
```

This is actually three different commands that stream their output from left to right. The first command searches the subdirectory tree (from the current directory) looking for all files that fit a certain pattern. The pattern defined is `'*.[ch]'`, which means all files that end in `.c` or `.h`. These are passed to the next command, `xargs`, which is a special command to read from standard input and pass to the embedded command. In this case, it's `grep`. The `grep` command is a text search utility that searches all files passed to it from the previous stage for the string term `mtj`. For files that pass the search criteria, the lines that contain the search term are emitted to the next stage, the `more` command. The `more` command simply ensures that the user is able to see all output before it scrolls by. When a screen full of output is present, the user must type `return` for `more` to continue and potentially present a new screen's worth of data.

Standard In/Out/Error

For each application, three special file descriptors are automatically created. These are called standard input, standard output, and standard error (see Figure 21.1).

This chapter refers to these by their shortened names for brevity. The `stdin` descriptor is commonly the keyboard. Descriptors `stdout` and `stderr` are the terminal or window attached to the shell (`stdout` for program results, `stderr` for program errors, warnings, and status). The output descriptors are split to provide greater flexibility for emitting information to the user. While `stdout` and `stderr` share the same default output device, they can be split as desired by the developer.

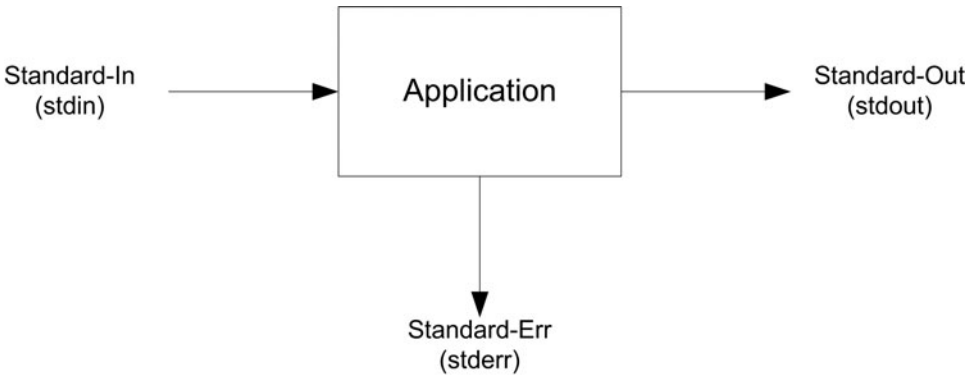

FIGURE 21.1 Program input and output.

Recall from the previous discussion in the chapter that you can redirect `stdout` to a file as follows:

```
prog > out.txt
```

where the output of `prog` is redirected to the file `out.txt`. Note that if you want to append our output to `out.txt` rather than replace the file altogether, you use the double redirect, as follows:

```
prog >> out.txt
```

You can redirect only the error output as follows:

```
prog 2> error-out.txt
```

Note that you are using a constant number here to represent `stderr`. The file descriptors that are defined for the three standard I/O descriptors are shown in Table 21.1.

TABLE 21.1 File Descriptors for Standard I/O Descriptors

Descriptor	Description
0	Standard input (`stdin`)
1	Standard output (`stdout`)
2	Standard error (`stderr`)

If instead you want to redirect both the `stdout` and `stderr` to a file (`out.txt`), you can do the following:

```
prog 1> out.txt 2>&1
```

For the opposite scenario, you can redirect the `stdout` to the `stderr` descriptor as follows:

```
prog 1>&2
```

You can also redirect output to unique files. For example, if you want your `stdout` to go to `out.txt` and `stderr` to go to `err.txt`, you can do the following:

```
prog 1>out.txt 2>err.txt
```

To verify that descriptor routing is working the way you expect, the script in Listing 21.1 can be used to test.

LISTING 21.1 Descriptor Routing Test Script (on CD-ROM at `./source/ch21/redirtest.sh`)

```
1:    #!/bin/bash
2:    echo "stdout test" >&1
3:    echo "stderr test" >&2
```

Finally, consider another example that demonstrates ordering of redirection.

```
prog 2>&1 1>out.txt
```

In this example, we redirect `stderr` to `stdout` and then redirect `stdout` (not including `stderr`) to the file `out.txt`. This has the effect of consolidating both the `stdout` and `stderr` to the file `out.txt`.

Environment Variables

An environment variable is a named object that contains information for use by the shell and other applications. A number of standard environment variables exist, such as the `PWD` variable, but you can create your own for your applications (or change existing variables). You can inspect the `PWD` variable by echoing its contents with the `echo` command:

```
$ echo $PWD
PWD=/home/mtj
$
```

A process can be viewed as an environment and inherits the environment variables from its parent (such as the shell, which is also a process). A process or script might also have local variables.

The `PWD` environment (or shell) variable identifies your current (present) working directory. You can create your own using the `declare` or `export` `bash` built-in command.

A script can make environment variables available to child processes, but only by exporting them. A script cannot export back to the parent process. Take a look at a couple of examples. The `declare` built-in command can be used to declare variables with specific attributes. The `export` built-in command creates a variable and marks it to be passed to child processes in the environment. These commands are illustrated as follows:

```
$ declare -x myvar="Hello"
$ echo $myvar
Hello
$ export myothervar="Hi"
$ echo $myothervar
Hi
$
```

A number of other useful environment variable commands exist. For example, if no argument is provided to `export`, then it emits all of the variables available to the environment (which can also be done with the `declare` and `set` commands).

Script Invocation

When you invoke a command or script, the command or script must be in your binaries path (`PATH` environment variable) in order for it to be found. You can view your path by echoing the `PATH` environment variable as follows:

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/mtj/bin
```

If you have created a script that does not exist in the path defined (for example, in your current working directory), then you have to invoke it as follows:

```
./script.sh
```

The `./` tells the interpreter that the shell script you are invoking is located in the current directory. Otherwise, you get an error message telling you that the script cannot be found.

Two special directory files exist that are important in Linux development. The `.` *file represents the current directory, whereas the* `..` *file represents the parent directory. For example, if you provide the command* `cd .`*, you have no visible change because you have changed the current directory to the current directory. More interesting, the* `cd ..` *command changes the current directory to the parent directory. These special files can be seen by viewing the current subdirectory with* `ls -la`*.*

If this is undesirable, you can easily update your `PATH` environment variable to add your current working subdirectory as follows:

```
$ export PATH=$PATH:./
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/mtj/bin:./
```

The script `script.sh` can now be invoked directly without needing to prefix `./`.

File links are special files that provide a reference to another file. Two types of links exist: *hard* links and *soft* links (otherwise known as symbolic links). A hard link is a new entry in the directory file that points to an existing file. The hard link is indistinguishable from the original file. The problem with hard links is that they must reference files within the same filesystem. Soft links are regular files themselves and simply contain a pointer to the actual file. Soft links can be absolute (point to a file with a full path) or relative (the path being relative from the current location). Soft links can be moved while retaining their linkage to the original file. Note that the special files `.` and `..` are in fact hard links to the absolute directories.

The `ln` *command can be used to construct symbolic links.*

Basic GNU/Linux Commands

Now that you have some basic understanding of redirection and the standard I/O descriptors (`stdin`, `stdout`, and `stderr`), it's time to explore the more useful of the GNU/Linux commands. This section takes an interactive approach for investigating these commands, compared to simply telling you what the command does and all the available options for it. The commands are in no particular order, and therefore you can explore each command independently of any other if desired.

Every command in GNU/Linux is itself a process. The shell manipulates the `stdin`, `stdout`, and `stderr` for commands that are executed in it.

tar

The GNU `tar` command (named for Tape ARchive) is a useful and versatile archiving and compressing utility. Targets for `tar` can be files or directories, where the directories are recursed to gather the full contents of the directory tree. Now it's time to take a look at uses of the `tar` utility, investigating a variety of the options as you go.

You can create a new archive using `tar` as follows:

```
tar cf mytar.tar mydir/
```

You specify two options for creating an archive of directory `mydir`. The `c` option instructs `tar` to create a new archive with the name (identified via the `f` option) `mytar.tar`. The final arguments are the list of files and/or directories to archive (for which all files in `mydir` are included in the archive).

Now say you want to take your `tar` file (also called a *tarball*) and re-create the subdirectory and its contents (otherwise known as the extract option). You could do this (in another subdirectory) by simply typing the following:

```
tar xf mytar.tar
```

If you want to know the contents of your tarball without having to unarchive the contents, you can do this:

```
tar tf mytar.tar
```

which lists all files with their directory paths intact.

By adding the `v`, or verbose option, you can view the operation of the `tar` utility as it works. For both creating and extracting archives, you use the verbose option as follows:

```
tar cvf mytar.tar mydir/
tar xvf mytar.tar
```

One of the most important aspects of the `tar` utility is the automatic compression of the tarball. This is performed using the `z` option and works symmetrically for both creation and extraction of archives, as follows:

```
tar czf mytar.tgz mydir/
tar xzf mytar.tgz
```

A compressed tarball can take more time to create and extract, but this can still be beneficial, especially if you intend to transfer the set of files over the Internet.

cut

The GNU/Linux cut utility can quickly cut elements of each line in a file using one of two types of specification. The user can define the desired data in terms of fields in the file or based upon numbered sequences of characters.

This section first takes a look at the basic format of the cut utility and then looks at some examples of how it can be used. As has been discussed, the cut utility can operate in two modes. In the first mode, cut extracts based upon field specifications using a delimiting character:

```
cut -f[spec] -d[delimiter] file
```

In the second mode, cut extracts based upon character position specifications:

```
cut -c[spec] file
```

The spec argument is a list of comma-separated ranges. A range can be represented (base 1) as shown in Table 21.2.

TABLE 21.2 Range Specs for the cut Utility

Range	Meaning
n	nth character (-c) or field (-f)
n-	nth character (-c) or field (-f) from the end of the line
n-m	From nth to mth character (-c) or field (-f)
-m	From first to mth character (-c) or field (-f)

Now it's time to look at some examples of the cut utility. You can explore the field-based cut first, using the sample file in Listing 21.2.

LISTING 21.2 Sample File (passwd) for Field-Based Cutting (on the CD-ROM at ./source/ch21/passwd)

```
bob:x:500:500::/home/bob:/bin/bash
sally:x:501:501::/home/sally:/bin/sh
jim:x:502:502::/home/jim:/bin/tcsh
dirk:x:503:503::/home/dirk:/bin/ash
```

You can experiment with `cut` interactively, looking not only at the command but also exactly what `cut` produces given the field specification. First, say you want to cut and emit the first field (the name). This is a simple case for `cut`, demonstrated as follows:

```
$ cut -f1 -d: passwd
bob
sally
jim
dirk
$
```

In this example, you specify to cut field 1 (`-f1`) using the colon character as the delimiter (`-d:`). If you want to know the home directory rather than the user name, you simply update the field to point to this element of `passwd` (field 6), as follows:

```
$ cut -f6 -d: passwd
/home/bob
/home/sally
/home/jim
/home/dirk
$
```

You can also extract multiple fields, such as the user name (field 1) and the preferred shell (field 7):

```
$ cut -f1,7 -d: passwd
bob:/bin/bash
sally:/bin/bash
jim:/bin/tcsh
dirk:/bin/ash
$
```

Ordering is important to the `cut` utility. For example, if you had specified `-f7,1` instead in the previous example, the result would have been the same (ordering of fields in the original file is retained).

Now take a look at some examples of character position specifications. In these examples, you pipe your input from another command. The `ls` command lists the contents of a directory, for example:

```
$ ls -la
total 20
drwxrwxr-x    2 mtj      mtj          4096 Feb 17 20:08 .
drwxr-xr-x    6 mtj      mtj          4096 Feb 15 20:47 ..
-rw-rw-r—     1 mtj      mtj          6229 Feb 16 17:59 ch12.txt
-rw-r—r—      1 mtj      mtj           145 Feb 17 20:02 passwd
```

If you are interested only in the file size (which includes the date and name of the file), you can do the following:

```
$ ls -la | cut -c38-
 4096 Feb 17 20:08 .
 4096 Feb 15 20:47 ..
 6229 Feb 16 17:59 ch12.txt
  145 Feb 17 20:02 passwd
$
```

If you are interested only in the size of the file and the file's name, you can use the following:

```
$ ls -la | cut -c38-42,57-
 4096.
 4096..
 6229ch12.txt
  145passwd
$
```

Note in this example that the two `cut` regions include no space between them. This is because `cut` simply segments the regions of the file and emits no spaces between those regions. If you are interested in a space, you can update the command as follows (taking a space from the data itself):

```
$ ls -la | cut -c38-43,57-
 4096 .
 4096 ..
 6229 ch12.txt
  145 passwd
$
```

The `cut` utility is very simple, but it's also a very useful utility with quite a bit of flexibility. The `cut` utility isn't the only game in town; later on in the book you will take a look at `sed` and `awk` and their capabilities for text filtering and processing.

paste

The paste command takes data from one or more files and binds them together into a new stream (with default emission to stdout). Consider the files shown in Listings 21.3 and 21.4.

LISTING 21.3 The Fruits File (on the CD-ROM at ./source/ch21/fruits.txt)

```
Apple
Orange
Banana
Papaya
```

LISTING 21.4 The Tools File (on the CD-ROM at ./source/ch21/tools.txt)

```
Hammer
Pencil
Drill
Level
```

Using the paste utility, you can bind these files together as demonstrated in the following:

```
$ paste fruits.txt tools.txt
Apple   Hammer
Orange  Pencil
Banana  Drill
Papaya  Level
$
```

If you want some delimiter other than tabs between your consecutive elements, you can specify a new one using the -d option. For example, you can use a : character instead, as follows:

```
$ paste -d: fruits.txt tools.txt
Apple:Hammer
Orange:Pencil
Banana:Drill
Papaya:Level
$
```

Rather than pair consecutive elements in a vertical fashion, you can instead pair them horizontally using the `-s` option:

```
$    paste -s fruits.txt tools.txt
Apple   Orange  Banana  Papaya
Hammer  Pencil  Drill   Level
$
```

Note that you can specify more than two files if desired.

One final example can help illustrate the `paste` utility. Recall from the earlier discussion of `cut` that it wasn't possible to alter the order of fields pulled from a file. The following short script provides the utility of listing the filename and then the size of the file (see Listing 21.5).

LISTING 21.5 Simple Reversed `ls` Utility Using `cut` and `paste` (on the CD-ROM at `./source/ch21/newls.sh`)

```
#!/bin/bash
ls -l | cut -c38-42 > /tmp/filesize.txt
ls -l | cut -c57- > /tmp/filename.txt
paste /tmp/filename.txt /tmp/filesize.txt
```

In this example, you first cut the file sizes from the `ls -l` command and store the result to `/tmp/filesize.txt`. You grab the filenames next and store them to `/tmp/filename.txt`. Next, using the `paste` command, you merge the results back together, reversing the order. Executing this script on a small directory results in the following:

```
$ ./newls.sh
fruits.txt         27
newls.sh          133
tools.txt          26
$
```

Note that the use of the `/tmp` directory in Listing 21.5 is useful for temporary files. Temporary files can be written to `/tmp` because they are not persistent across system boots. In some cases, files are removed from `/tmp` as part of a daily or weekly cleanup process.

sort

The `sort` utility is useful for sorting a data file in some defined order. In the simplest case, where the key for `sort` is the beginning of the file, you specify the file. Take for example the `tools.txt` file shown in Listing 21.4. You can easily sort this as follows:

```
$ sort tools.txt
Drill
Hammer
Level
Pencil
$
```

You can reverse sort this file by simply adding the `-r` (reverse) option to the command line.

You can also sort based upon a key defined by the user. Consider the sample text file shown in Listing 21.6. This file contains five lines with three columns, with none of the columns being presorted.

LISTING 21.6 Sample File for Sorting (on the CD-ROM at `./source/ch21/table.txt`)

```
5 11 eee
4  9 ddd
3 21 aaa
2 24 bbb
1  7 ccc
```

To specify a column to sort, you use the `-k` (or key) option. The key represents the column for which you desire the file to be sorted. You can specify more than one key by simply separating them with commas. To sort based upon the first column, you can specify the key as column 1 (or not specify it at all, because it is the default):

```
$ sort -k 1 table.txt
1  7 ccc
2 24 bbb
3 21 aaa
4  9 ddd
5 11 eee
$
```

To sort the second column, another option is required to perform a numeric sort. The space character that comes before the single-digit numbers is significant and therefore precludes a numeric sort. Therefore, you use the `-n` option to force a numeric sort. For example:

```
$ sort -k 2 -n table.txt
1  7 ccc
4  9 ddd
5 11 eee
3 21 aaa
2 24 bbb
$
```

One other useful option allows the specification of a new delimiter, in the event spaces or tabs are not used. The `-t` option allows you to use a different delimiter, such as `-t:` to specify the colon character as the field separator.

find

The `find` utility is a powerful but complex utility that permits searching the file-system for files based upon given criteria. Rather than walk through the plethora of options available for `find`, this section demonstrates some of the more useful patterns.

To find all files that end in `.c` and `.h` in the current subdirectory, the following command can be used:

```
find . -name '*.[ch]' -print
```

The `.` specifies that you want to start at the current subdirectory. The `-name` argument refers to what you are searching for, in this case any file (`'*'`) that ends in either `c` or `h`. Finally, you specify `-print` to emit the search results to standard-out.

For each of the search results, you can execute a command using `-exec`. This permits you to invoke a command on each file that was found based upon the search. For example, if you want to change the file permissions of all of the files found to read-only, you can do the following:

```
find . -name '*.[ch]' -exec chmod 444 {} \;
```

You can also restrict the type of files that you'll look at using the `type` modifier. For example, if you want to look only at regular files (a common occurrence when searching for source files), you can do the following:

```
find . -name '*.[ch]' -type f -print
```

The `f` argument to `-type` represents regular files. You can also look specifically for directories, symbolic links, or special devices. Table 21.3 provides the type modifiers that are available.

TABLE 21.3 Type Modifiers Available to `find` (`-type`)

Modifier	Description
`b`	Block device
`c`	Character device
`d`	Directory
`p`	Pipe (named FIFO)
`f`	Regular file
`l`	Symbolic link
`s`	Socket

One final useful `find` use is identifying files within a directory that have been changed within a certain period of time. The following command (using `mtime`) identifies the files that have been modified in a given time range (multiples of 24 hours). The following command identifies files that have been modified in the last day.

```
find  -name '*' -type f -mtime -1 -print
```

To find files modified in the last week, you can update the `mtime` argument to `-7`. The `atime` argument can be used to identify recently accessed files. The `ctime` argument identifies files whose status was changed recently.

wc

The `wc` utility is very useful to count the number of characters, words, or lines within a file.

The following samples illustrate the possibilities of the `wc` utility.

```
wc -m file.txt        # Count characters in file.txt
wc -w file.txt        # Count words in file.txt
wc -l file.txt        # Count lines in file.txt
```

All three counts can be emitted by accumulating the arguments, as follows:

```
wc -l -w -m file.txt
```

Regardless of the order of the flags, the order of count emission is always lines, then words, and then characters.

grep

The `grep` command permits searching one or more files for a given pattern. The format of the `grep` command can be complex, but the simpler examples are quite useful. This section looks at a few simple examples and discusses what they achieve.

In its simplest form, you can search a single file for a given search string, as follows:

```
grep "the" file.txt
```

The result of this example is each line emitted to `stdout` that contains the word `the`. Rather than specify a single file, you can use the wildcard to check multiple files, as follows:

```
grep "the" *.txt
```

In this case, all files in the current subdirectory that end in `.txt` are checked for the search string.

When the wildcard is used, the filename from which the search string is found is emitted before each line that's emitted to `stdout`.

If the line location of the file where the search string is found is important, the `-n` option can be used.

```
grep -n "the" *.txt
```

Each time the search string is found, the filename and line number where the string was found are emitted before the line. If you are interested only in whether the particular string is found in a file, you can use the `-l` option to simply emit filenames, rather than the lines from which the string is found:

```
grep -l "the" *.txt
```

When you're searching specifically for words within a file, the `-w` option can be helpful in restricting the search to whole words only:

```
grep -w "he" *.txt
```

This option is very useful. When searching for `he`, for example, you also find occurrences of the word `the`. The `-w` option restricts the search only to words that match, so `the` and `there` do not result in a match to `he`.

head and tail

The `head` and `tail` commands can be used to view a portion of a file. As the names imply, `head` allows you to view the upper portion of a file whereas `tail` allows you to view the end of the file.

The number of lines to be emitted can be specified with the `–n` option, the default being 10 lines. Requesting the first two lines of a file with `head` is accomplished as follows:

```
head –n 2 file.txt
```

Similarly, requesting the last two lines of a file using `tail` uses a similar syntax:

```
tail –n 2 file.txt
```

These commands can be joined together with a pipe operation to extract specific slices of a file. For example, the following commands can be executed to view the second to last line of the file:

```
tail –n 2 file.txt | head –n 1
```

The `tail` command is commonly used to monitor files that are actively being written. For example, log files can be monitored as they grow with the `–f` option (which stands for *follow*):

```
tail –f log.txt
```

In addition to operating on the basis of lines, `head` and `tail` can also be used with bytes. Simply specify the `–c` option to indicate the number of bytes.

od

The `od` utility was once called "octal-dump," but is now a general utility that emits files in various representations. The default output remains octal, but files can be emitted in other formats such as characters or hexadecimal. Consider this sample file that contains two short lines of text (emitted with `cat`).

```
$ cat test.txt
This is a
test file.
$
```

With `od`, you can see this in hex format with the `–x` option as follows:

```
$ od -x test.txt
0000000 6854 7369 6920 2073 2061 740a 7365 2074
0000020 6966 656c 0a2e
0000026
$
```

The character option to `od` shows the character contents of the file, but also special characters (such as newlines).

```
$ od -c test.txt
0000000   T   h   i   s       i   s       a      \n   t   e   s   t
0000020   f   i   l   e   .  \n
0000026
$
```

The `od` utility is very useful for file inspection. In addition to octal, hex, and character dumps, it also supports standard types like shorts, floats, and ints. You can also specify the size of the type objects. For example `–t x1` specifies to emit hex bytes (where `x4` represents 4-byte hex values).

Summary

This chapter explored some of the basics of commanding in the GNU/Linux shell. The standard input and output descriptors were investigated (`stdin`, `stdout`, and `stderr`), along with redirection and command pipelining. The chapter also looked at ways of invoking shell scripts, including the addition of the current working directory to the default search path. Finally, you saw some of the more useful GNU/Linux commands, including `tar` (file archives), `find` (file search), `grep` (string search), and some other useful text processing commands (`cut`, `paste`, and `sort`).

22 Bourne-Again Shell (Bash)

In This Chapter

- An Introduction to bash Scripting
- Scripting versus Interactive Shell Use
- User Variables and Environmental Variables
- Arithmetic in bash
- Tests, Conditionals, and Loops in bash
- Script Input and Output
- Dissecting of Useful Scripts

Introduction

This chapter explores script programming in the Bourne-Again SHell, otherwise known as Bash. Bash is the de facto standard command shell and shell scripting language on Linux and other UNIX systems. This chapter investigates some of the major language features of Bash, including variables, arithmetic, control structures, input and output, and function creation. This follows the flow of all other scripting chapters in this book, allowing you to easily understand the similarities and differences of each covered language.

Preliminaries

Before jumping in to scripting with bash, you first need to look at some preliminaries that are necessary to run bash scripts. You can tell which shell you're currently using by interrogating the SHELL environment variable:

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

The `echo` command is used to print to the screen. You print the contents of the `SHELL` variable, accessing the variable by preceding it with the `$` symbol. The result is the shell that you're currently operating on, in this case, `bash`. Technically, it printed out the location of the shell you're using (the `bash` interpreter is found within the `/bin` subdirectory).

If you happened not to be using `bash`, you could simply invoke `bash` to start that interpreter:

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

In this case, you were using another command shell (C-shell here). You invoke `bash` to start this interpreter for further commanding.

Sample Script

You can now write a simple script as a first step to `bash` scripting. The source for the script is shown in Listing 22.1.

LISTING 22.1 First `bash` Script (on the CD-ROM at `./source/ch22/first.sh`)

```
#!/bin/bash
echo "Welcome to $SHELL scripting."
exit
```

When invoked, this script simply emits the line "Welcome to /bin/bash scripting." and then exits. If you try to enter this script (named `first.sh`) and execute it as `./first.sh`, you notice that it doesn't work. You probably see something like this:

```
$ ./first.sh
-bash: ./first.sh: Permission denied.
$
```

The problem here is that the script is not executable. You must first change the attributes of the file to tell GNU/Linux that it can be executed. This is done using the `chmod` command, illustrated as follows:

```
$ chmod +x first.sh
$ ./first.sh
Welcome to /bin/bash scripting.
$
```

You use `chmod` to set the `execute` attribute of the file `first.sh`, telling GNU/Linux that it can be executed. After trying to execute again, you see the expected results.

The question you could ask yourself now is, even though you have told GNU/Linux that the file can be executed, how does it know that the file is a `bash` script? The answer is the "shebang" line in the script. Notice that the first line of the script starts with `#!` (also called *shebang*) followed by the path and interpreter (`/bin/bash`). This defines that `bash` is the shell to be used to interpret this file. If the file had contained a Perl script, it would have begun `#! /bin/perl`. You can also add comments to the script simply by preceding them with a `#` character.

Because the interpreter is also the shell, you can perform this command interactively, as follows:

```
$ echo "Welcome to $SHELL scripting."
Welcome to /bin/bash scripting.
$
```

Now that you have some basics under your belt, it's time to dig into `bash` and work through some more useful scripts.

bash Scripting

This chapter takes a look at the `bash` scripting language. The following sections identify the necessary language constructs for application development, including variables, operations on variables, conditionals, looping, and functions. These sections also demonstrate a number of sample applications to illustrate scripting principles.

Variables

Any worthwhile language permits the creation of variables. In `bash`, variables are untyped, which means that all variables are in essence strings. This doesn't mean you can't do arithmetic on bash variables, which is explored shortly. You can create a variable and then inspect it very easily as follows:

```
$ x=12
$ echo $x
12
$
```

In this example, you create a variable `x` and bind the value `12` to it. You then echo the variable `x` and find your value. Note that the lack of space between the variable name, the equals, and the value is relevant. There can be no spaces; otherwise, an error occurs. Note also that to reference a variable, you precede it with the dollar sign. This variable is scoped (exists) for the life of the shell from which this sequence was performed. Had this sequence of commands been performed within a script (such as `./test.sh`), the variable `x` would not exist after the script was completed.

As `bash` doesn't type its variables, you can create a string variable similarly:

```
$ b="my string"
$ echo $b
my string
$
```

Note that single quotes also would have worked here. An interesting exception is the use of the *backtick*. Consider the following:

```
$ c='echo $b'
$ echo $c
my string
$
```

The backticks have the effect of evaluating the contents within the backticks, and in this case the result is assigned to the variable `c`. When you emit variable `c`, you find the value of the original variable `b`.

The `bash` interpreter defines a set of environment variables that effectively define the environment. These variables exist when the `bash` shell is started (though others can be created using the `export` command). Consider the script in Listing 22.2, which makes use of special environment variables to identify the environment of the script.

LISTING 22.2 Sample Script Illustrating Standard Environmental Variables (on the CD-ROM at `./source/ch22/env.sh`)

```
1:    #!/bin/bash
2:
3:    echo "Welcome to host $HOSTNAME running $OSTYPE."
```

```
4:    echo "You are user $USER and your home directory is $HOME."
5:    echo "The version of bash running on this system is $BASH_VERSION."
6:    sleep 1
7:    echo "This script has been running for $SECONDS second(s)."
8:    exit
```

Upon executing this script, you see the following on a sample GNU/Linux system:

```
$ ./env.sh
Welcome to host camus running linux-gnu.
You are user mtj and your home directory is /home/mtj.
The version of bash running on this system is 2.05b.0(1)-release.
This script has been running for 1 second(s).
$
```

Note that you added a useless sleep call to the script (to stall it for one second) so that you could see that the `SECONDS` variable was working. Also note that the `SECONDS` variable can be emitted from the command line, but this value represents the number of seconds that the shell has been running.

`bash` provides a variety of other special variables. Some of the more important ones are shown in Table 22.1. These can be echoed to understand their formats.

TABLE 22.1 Useful Environment Variables

Variable	Description
`$PATH`	Default path to binaries
`$PWD`	Current working directory
`$OLDPWD`	Last working directory
`$PPID`	Process ID of the interpreter (or script)
`$#`	Number of arguments
`$0, $1, $2, ...`	Arguments
`$*`	All arguments as a single word

One final word about variables in `bash` and then you can move on to some real programming. You can declare variables in `bash`, providing some form of typing. For example, you can declare a constant variable (cannot be changed after definition) or declare an integer or even a variable whose scope extends beyond the script. Examples of these variables are shown interactively in the following:

```
$ x=1
$ declare -r x
$ x=2
-bash: x: readonly variable
$ echo $x
1
$ y=2
$ declare -i y
$ echo $y
2
$ persist='$PWD'
$ declare -x persist
$ export | grep persist
declare -x persist="/home/mtj"
```

The last item might require a little more discussion. In this example, you create a variable `persist` and assign the current working subdirectory to it. You declare for exporting outside of the environment, which means if it had been done in a script, the variable would remain after the script had completed. This can be useful to allow scripts to alter the environment or to modify the environment for child processes.

Simple Arithmetic

You can perform simple arithmetic on variables, but you find differences from normal assignments that were just reviewed. Consider the source in Listing 22.3.

LISTING 22.3 Simple Script Illustrating Arithmetic with Variables (on the CD-ROM at `./source/ch22/arith.sh`)

```
 1:    #!/bin/bash
 2:
 3:    x=10
 4:    y=5
 5:
 6:    let sum=$x+$y
 7:    diff=$(($x - $y))
 8:    let mul=$x*$y
 9:    let div=$x/$y
10:    let mod=$x%$y
11:    let exp=$x**$y
12:
```

```
13:     echo "$x + $y = $sum"
14:     echo "$x - $y = $diff"
15:     echo "$x * $y = $mul"
16:     echo "$x / $y = $div"
17:     echo "$x ** $y = $exp"
18:     exit
```

At lines 3 and 4, you create and assign values to two local variables, called `x` and `y`. You then illustrate simple math evaluations using two different forms. The first form uses the `let` command to assign the evaluated expression to a new variable. No spaces are permitted in this form. The second example uses the `$(( <expr> ))` form. Note in this case that spaces are permitted, potentially making the expression much easier to read.

Bitwise Operators

Standard bitwise operators are also available in `bash`. These include bitwise left shift (`<<`), bitwise right shift (`>>`), bitwise AND (`&`), bitwise OR (`|`), bitwise negate (`~`), bitwise NOT (`!`), and bitwise XOR (`^`). The following interactive session illustrates these operators:

```
$ a=4
$ let b="$a<<1"
$ echo $b
8
$ b=8
$ c=4
$ echo $(($c|$d))
12
$ echo $((0xc^0x3))
15
$
```

Logical Operators

Traditional logical operators can also be found within `bash`. These include the logical AND (`&&`) and logical OR (`||`). The following interactive session illustrates these operators:

```
$ echo $((2 && 0))
0
$ echo $((4 && 1))
1
$ echo $((3 || 0))
1
$ echo $((0 || 0))
0
```

The next section investigates how these can be used in conditionals for decision points.

Conditional Structures

`bash` provides the typical set of conditional constructs. This section explores each of these constructs and also investigates some of the other available conditional expressions that can be used.

Conditionals

This section looks at conditionals. The `if/then` construct provides a decision point after evaluating a test construct. The test construct returns a value as its result. The result of the test construct is zero for true (test succeeds and subsequent commands are executed) or nonzero for false (test fails and `else` section, if available, is executed). Take a look at a simple example to illustrate (see Listing 22.4).

LISTING 22.4 Simple Script Illustrating Basic `if/then/else` Construct (on the CD-ROM at `./source/ch22/cond.sh`)

```
1:    #!/bin/bash
2:    a=1
3:    b=2
4:    if [[ $a -eq $b ]]
5:    then
6:      echo "equal"
7:    else
8:      echo "unequal"
9:    fi
```

Note that the result of the test construct is the inverse of what you would expect. This is because the exit status of a command is `0` *for success/normal and* `!= 0` *to indicate an error.*

After creating two variables, you test them for equality using the `-eq` comparison operator. If the test construct is true, you perform the commands contained in the `then` block. Otherwise, if an `else` block is present, this is executed (the test construct was false). `else-if` chains can also be constructed, as shown in Listing 22.5.

LISTING 22.5 Simple Script Illustrating the `if/then/elif/then/fi` Construct (on the CD-ROM at `./source/ch22/cond2.sh`)

```
#!/bin/bash
x=5
y=8
if [[ $x -lt $y ]]
then
  echo "$x < $y"
elif [[ $x -gt $y ]]
then
  echo "$x > $y"
elif [[ $x -eq $y ]]
then
  echo "$x == $y"
fi
```

In this example, you test the integers using the `-lt` operator (less than), `-gt` (greater than), and finally `-eq` (equality). Other operators are shown in Table 22.2.

Test constructs can also utilize strings such as illustrated in Listing 22.6. In this example, you also look at two forms of the `if/then/fi` construct that provide identical functionality.

TABLE 22.2 Integer Comparison Operators

Operator	Description
`-eq`	Is equal to
`-ne`	Is not equal to
`-gt`	Is greater than
`-ge`	Is greater than or equal to
`-lt`	Is less than
`-le`	Is less than or equal to

LISTING 22.6 Simple Script Illustrating the `if/then/fi` Construct (on the CD-ROM at `./source/ch22/cond3.sh`)

```
1:      #!/bin/bash
2:      str="ernie"
3:      if [[ $str = "Ernie" ]]
4:      then
5:        echo "It's Ernie"
6:      fi
7:
8:
9:      if [[ "$str" == "Ernie" ]]; then echo "It's Ernie"; fi
```

After creating a string variable at line 2, you test it against a constant string at line 3. The = operator tests for string equality, as does the == operator. At line 9, you look at a visibly different form of the `if/then/fi` construct. As it's represented on one line, semicolons separate the individual commands.

In Listing 22.5, you saw the use of the string equality operators. In Table 22.3, we see some of the other string comparison operators.

TABLE 22.3 String Comparison Operators

Operator	Description
=	Is equal to
==	Is equal to
!=	Is not equal to
<	Is alphabetically less than
>	Is alphabetically greater than
-z	Is null
-n	Is not null

As a final look at test constructs, take a look some of the more useful file test operators. Consider the script shown in Listing 22.7. In this script, you emit some information about a file (based upon its attributes). You use the file test operators to determine the attributes of the file.

Listing 22.7 Determine File Attributes Using File Test Operators (on the CD-ROM at `./source/ch22/fileatt.sh`)

```
 1:    #!/bin/sh
 2:    thefile="test.sh"
 3:
 4:    if [ -e $thefile ]
 5:    then
 6:      echo "File Exists"
 7:
 8:      if [ -f $thefile ]
 9:      then
10:        echo "regular file"
11:      elif [ -d $thefile ]
12:      then
13:        echo "directory"
14:      elif [ -h $thefile ]
15:      then
16:        echo "symbolic link"
17:      fi
18:
19:    else
20:      echo "File not present"
21:    fi
22:
23:    exit
```

The first thing to notice in Listing 22.7 is the embedded `if/then/fi` construct. After you identify that the file exists at line 4 using the `-e` operator (returns true of the file exists), you continue to test the attributes of the file. At line 8, you check to see whether you're dealing with a regular file (`-f`), in other words a real file as compared to a directory, a symbolic link, and so forth. The file tests continue with a directory test at line 11 and finally a symbolic link test at line 14. Lines 19–21 close out the initial existence test by emitting whether the file was actually present.

A large number of file test operators is provided by `bash`. Some of the more useful operators are shown in Table 22.4.

TABLE 22.4 File Test Operators

Operator	Description
`-e`	Test for file existence
`-f`	Test for regular file
`-s`	Test for file with nonzero size
`-d`	Test for directory
`-h`	Test for symbolic link
`-r`	Test for file read permission
`-w`	Test for file write permission
`-x`	Test for file execute permission

For the file test operators shown in Table 22.4, a single file argument is provided for each of the tests. Two other useful file test operators compare the dates of two files, illustrated as follows:

```
if [ $file1 -nt $file2 ]
then
  echo "$file is newer than $file2"
elif [ $file1 -ot $file2 ]
then
  echo "$file1 is older than $file2"
fi
```

The file test operator `-nt` tests whether the first file is *newer than* the second file, while `-ot` tests whether the first file is *older than* the second file.

If you are more interested in the reverse of a test, for example, whether a file is not a directory, then the `!` operator can be used. The following code snippet illustrates this use:

```
if [ ! -d $file1 ]
then
  echo "File is not a directory"
fi
```

One special case to note is when you have a single command to perform based upon the success of a given test construct. Consider the following:

```
[ -r myfile.txt ] && echo "the file is readable."
```

If the test succeeds (the file `myfile.txt` is readable), then the command that follows is executed. The logical AND operator between the test and command ensures that only if the initial test construct is true is the command that follows performed. If the test construct is false, the rest of the line is ignored.

This has been a quick introduction to some of the `bash` test operators. The "Resources" section at the end of this chapter provides more information to investigate further.

`case` Construct

Take a look at another conditional structure that provides some advantages over standard `if` conditionals when testing a large number of items. The `case` command permits a sequence of test constructs utilizing integers or strings. Consider the example shown in Listing 22.8.

LISTING 22.8 Simple Example of the `case/esac` Construct (on the CD-ROM at `./source/ch22/case.sh`)

```
 1:    #!/bin/bash
 2:    var=2
 3:
 4:    case "$var" in
 5:      0) echo "The value is 0" ;;
 6:      1) echo The value is 1 ;;
 7:      2) echo The value is 2 ;;
 8:      *) echo The value is not 0, 1, or 2
 9:    esac
10:
11:    exit
```

The `case` construct shown in Listing 22.8 illustrates testing an integer among three values. At line 4, you set up the `case` construct using `$var`. At line 5, you perform the test against 0, and if it succeeds, the commands that follow are executed. Lines 6 and 7 test against values 1 and 2. Finally at line 8, the default `*` simply says that if all previous tests failed, this line is executed. At line 9, the `case` structure is closed. Note that the command list within the tests ends with `;;`. This indicates to the interpreter that the commands are finished, and either another `case` test or the closure of the `case` construct is coming. Note that the ordering of `case` tests is important. Consider if line 8 had been the first test instead of the last. In this case, the default would always succeed, which isn't what is desired.

You can also test ranges within the test construct. Consider the script shown in Listing 22.9 that tests against the ranges `0-5` and `6-9`. The special form `[0-5]` defines a range of values between 0 and 5 inclusive.

LISTING 22.9 Simple Example of the `case/esac` Construct with Ranges (on the CD-ROM at `./source/ch22/case2.sh`)

```
1:    #!/bin/bash
2:    var=2
3:
4:    case $var in
5:      [0-5] ) echo The value is between 0 and 5 ;;
6:      [6-9] ) echo The value is between 6 and 9 ;;
7:      *) echo It's something else...
8:    esac
9:
10:   exit
```

The `case` construct can be used to test characters as well. The script shown in Listing 22.10 illustrates character tests. Also shown is the concatenation of ranges, here `[a-zA-z]` tests for all alphabetic characters, both lower- and uppercase.

LISTING 22.10 Another Example of the `case/esac` Construct Illustrating Ranges (on the CD-ROM at `./source/ch22/case3.sh`)

```
1:    #!/bin/bash
2:
3:    char=f
4:
5:    case $char in
6:      [a-zA-z] ) echo An upper or lower case character ;;
7:      [0-9]    ) echo A number ;;
8:      *        ) echo Something else ;;
9:    esac
10:
11:   exit
```

Finally, strings can also be tested with the `case` construct. A simple example is shown in Listing 22.11. In this example, a string is checked against four possibilities. Note that at line 7, the test construct is made up of two different tests. If the name is `Marc` or `Tim`, then the test is satisfied. You use the logical OR operator in this case, which is legal within the `case` test construct.

LISTING 22.11 Simple String Example of the `case/esac` Construct (on the CD-ROM at `./source/ch22/case4.sh`)

```
#!/bin/bash

name=Tim

case $name in
  Dan        ) echo It's Dan. ;;
  Marc | Tim ) echo It's me. ;;
  Ronald     ) echo It's Ronald. ;;
  *          ) echo I don't know you. ;;
esac

exit
```

This has been the tip of the iceberg as far as `case` test constructs go—many other types of conditionals are possible. The "Resources" section at the end of this chapter provides other sources that dig deeper into this area.

Looping Structures

Now take a look at how looping constructs are performed within `bash`. This section looks at the two most commonly used constructs: the `while` loop and the `for` loop.

`while` Loops

The `while` loop simply performs the commands within the `while` loop as long as the conditional expression is true. First take a look at a simple example that counts from 1 to 5 (shown in Listing 22.12).

LISTING 22.12 Simple `while` Loop Example (on the CD-ROM at `./source/ch22/loop.sh`)

```
#!/bin/bash

var=1

while [ $var -le 5 ]
do
  echo var is $var
  let var=$var+1
```

```
 9:     done
10:
11:     exit
```

In this example, you define the looping conditional at line 5 (`var <= 5`). While this condition is true, you print out the value and increment `var`. After the condition is false, you fall through the loop to `done` (at line 9) and ultimately exit the script.

Loops can also be nested. The sample script in Listing 22.13 illustrates this. In this example, you generate a multiplication table of sorts using two variables. Lines 4 to 18 define the `outer` loop, whereas lines 8 to 14 define the `inner`. The only difference, as in other high-level languages, is that the `inner` loop is indented to show the structure of the code.

LISTING 22.13 Nested `while` Loop Example (on the CD-ROM at `./source/ch22/loop2.sh`)

```
 1:     #!/bin/bash
 2:     outer=0
 3:
 4:     while [ $outer -lt 5 ] ; do
 5:
 6:       inner=0
 7:
 8:       while [ $inner -lt 3 ] ; do
 9:
10:         echo $outer * $inner = $(($outer * $inner))
11:
12:         inner=$(expr $inner + 1)
13:
14:       done
15:
16:       let outer=$outer+1
17:
18:     done
19:
20:     exit
```

Another interesting item to note about the script in Listing 22.13 is the arithmetic expressions used. In the `outer` loop you find the use of `let` to assign `outer` to itself plus one. The `inner` loop uses the `expr` command, which is an expression evaluator.

for Loops

The for/in/do/done construct in bash enables you to loop through a range of variables. This differs from the classical for loop that is available in high-level languages such as C, but bash's perspective is very useful and offers some capabilities not found in C. Listing 22.14 provides a very simple for loop.

LISTING 22.14 Simple for Loop Example (on the CD-ROM at ./source/ch22/forloop.sh)

```
 1:    #!/bin/bash
 2:
 3:    echo Counting from 1 to 5
 4:
 5:    for val in 1 2 3 4 5
 6:    do
 7:      echo -n $val
 8:    done
 9:    echo
10:
11:    exit
```

The result of this script is:

```
# ./test.sh
Counting from 1 to 5
1 2 3 4 5
#
```

Of course, you can emulate the C for loop mechanism very simply as shown in Listing 22.15.

Listing 22.15 Simple for Loop Example Using C-Like Construct (on the CD-ROM at ./source/ch22/forloop2.sh)

```
 1:    #!/bin/bash
 2:
 3:    for ((var=1 ; var <= 5 ; var++))
 4:    do
 5:      echo -n $var
 6:    done
 7:    echo
 8:
 9:    exit
```

This code in Listing 22.15 is identical to the original `for-in` loop shown in Listing 22.14.

You can also use strings within the looping range, as illustrated in Listing 22.16. This script simply iterates through the string's provided range.

LISTING 22.16 Another `for` Loop Illustrating String Ranges (on the CD-ROM at `./source/ch22/forloop3.sh`)

```
1:    #!/bin/bash
2:
3:    echo -n The first four planets are
4:    for planet in mercury venus earth mars ; do
5:      echo -n  $planet
6:    done
7:    echo .
8:
9:    exit
```

Where `bash` shines over traditional high-level languages is in the capability to replace ranges with results of commands. Take a look at a more complicated example of the `for-in` loop that uses the replacement symbol `*`, which indicates the files in the current subdirectory (see Listing 22.17).

LISTING 22.17 Listing the User Subdirectories on the System Using Wildcard Replacement (on the CD-ROM at `./source/ch22/forloop4.sh`)

```
1:    #!/bin/bash
2:
3:    # Save the current directory
4:    curwd=$PWD
5:
6:    # Change the current directory to /home
7:    cd /home
8:
9:    echo -n Users on the system are:
10:
11:   # Loop through each file (via the wildcard)
12:   for user in *; do
13:     echo -n  $user
14:   done
15:   echo
16:
```

```
17:     # Return to the previous directory
18:     cd $curwd
19:
20:     exit
```

Input and Output

You've seen some examples already of output using the `echo` command. This command simply emits the provided string to the display. You also saw suppression of the newline character using the `-n` option. The `echo` command also provides the means to emit data through a binary interface. For example, to emit horizontal tabs, the `\t` option can be used, as follows:

```
echo -e \t\t\t\tIndented text.
```

Some of the other options that exist are shown in Table 22.5. To enable interpretation of these strings, the `-e` option must be specified before the string.

TABLE 22.5 Special Sequences in Echoed Strings

Sequence	Interpretation
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\\`	Backslash
`\NNN`	ASCII code of octal value

You can accept input from the user using the `read` command. The `read` command provides a number of options, a few of which are investigated here. First, take a look at the basic form of a `read` command via the interactive `bash` shell:

```
# read var
test string
# echo $var
test string
# read -s var
```

```
# echo $var
silent input
```

In the first form, you read a string from the user and store it into variable `var`. In the second form of `read` you specify the `-s` flag. The `-s` flag represents silent input, which means that characters that are entered in response to a `read` are not echoed back to the screen. In this case, you typed `silent input`, and you see this after the variable is echoed back.

Some of the other options that exist for the `read` command are shown in Table 22.6

TABLE 22.6 Other Options for the `read` Command

Option	Description
`-a`	Input is assigned into an array, starting with index 0
`-d`	Character to use to terminate input (rather than newline)
`-n`	Maximum number of characters to read
`-p`	Prompt string displayed to prompt user for input
`-s`	Silent mode (don't echo input characters)
`-t`	Timeout in seconds for read
`-u`	File descriptor to read from rather than terminal

Functions

Bash allows you to break scripts up into more manageable pieces by creating functions. Functions can be very simple, such as:

```
function <name> () {
    sequence of command
}
```

As in C, the function must be declared before it can be called. Now take a look at a simple example of a function that sums together two numbers that are passed in from the caller (see Listing 22.18).

LISTING 22.18 Creating a Function That Utilizes Parameters (on the CD-ROM at `./source/ch22/func.sh`)

```
 1:    #!/bin/bash
 2:
 3:    function sum ()
 4:    {
 5:
 6:      echo $(( $1 + $2 ))
 7:
 8:    }
 9:
10:    sum 5 10
```

In this example, you declare a new function called `sum` (lines 3–8), which emits the sum of the two parameters passed to it. Recall that `$1` represents the first parameter and `$2` represents the second. So what does `$0` represent? Just as in C, the first argument from the perspective of a main program is the name of the program itself that was called. In this case, the name is the script file itself. What happens if the caller doesn't provide all of the necessary parameters (passes only one parameter instead of two)? This would be a good time for some error checking, so you can update the script as shown in Listing 22.19.

LISTING 22.19 Adding Error Checking to Our Previous Function (on the CD-ROM at `./source/ch22/func2.sh`)

```
 1:    #!/bin/bash
 2:
 3:    function sum ()
 4:    {
 5:
 6:      if [ $# -ne 2 ] ; then
 7:        echo usage is sum <param1> <param2>
 8:        exit
 9:      fi
10:
11:      echo $(( $1 + $2 ))
12:
13:    }
14:
15:    sum 5 10
```

Note in this version (updated from Listing 22.18) that error checking is now performed in lines 6–9. You test the special variable `$#`, which represents the number of parameters passed to the constant 2. Because you're expecting two arguments to be passed to us, you echo the use and exit if two parameters are not present.

The parameter variables are dependent upon context. So, in Listing 22.20, the `$1` *parameter at line 15 is different from the* `$1` *present at line 6.*

You can also return values from functions. You use the `return` command to actually return the value from the function, and then you use the special variable `$?` to access this value from the caller. See the example shown in Listing 22.20.

LISTING 22.20 Adding Function `return` to Our Previous `sum` Function (on the CD-ROM at `./source/ch22/func3.sh`)

```
 1:    #!/bin/bash
 2:
 3:    function sum ()
 4:    {
 5:
 6:      if [ $# -ne 2 ] ; then
 7:        echo usage is sum <param1> <param2>
 8:        exit
 9:      fi
10:
11:      return $(( $1 + $2 ))
12:
13:    }
14:
15:    sum 5 10
16:    ret=$?
17:
18:    echo $ret
```

In this version, rather than echo the result of the summation, you return it to the caller at line 11. At line 16, you grab the result of the function using the special `$?` variable. This variable represents the exit status of the last function called.

Sample Scripts

Now that you've covered some of the basic elements of scripting, from variables to conditional and looping structures, it's time to look at some sample scripts that actually provide some useful functionality.

Simple Directory Archive Script

The goal of the first script is to provide a subdirectory archive tool. The single parameter for the tool is a subdirectory that is archived using the `tar` utility, with the resulting archive file stored in the current working subdirectory. The script source can be found in Listing 22.21.

LISTING 22.21 Directory Archive Script (on the CD-ROM at `./source/ch22/archive.sh`)

```
 1:    #!/bin/bash
 2:
 3:    # First, do some error checking
 4:    if [ $# -ne 1 ] ; then
 5:      echo Usage is ./archive.sh <directory-name>
 6:      exit -1
 7:    fi
 8:
 9:    if [ ! -e $1 ] ; then
10:      echo Directory does not exist
11:      exit -1
12:    fi
13:
14:    if [ ! -d $1 ] ; then
15:      echo Target must be a directory.
16:      exit -1
17:    fi
18:
19:    # Remove the existing archive
20:    archive=$1.tgz
21:
22:    if [ -f $archive ] ; then
23:      rm -f $archive
24:    fi
25:
26:    # Archive the directory
27:    tar czf $archive $1
28:
29:    exit
```

As is probably apparent right away, the script in Listing 22.21 is mostly error checking. You first ensure that there's a single argument to the script at lines 4–7. You then check that the target provided actually exists at lines 9–12, and that it's a

directory at lines 14–17. At line 20, you create the archive file by appending the extension `.tgz` to the end. If the archive exists, you remove it at lines 22–24. Finally, you call the `tar` utility to create the archive at line 27. You specify three arguments to `tar`: `c` to create the archive, `z` to filter using `gzip`, and `f` to specify the filename for the archive.

Files Updated/Created Today Script

The goal of this script is to recursively search a directory to print any files that have been updated today. This is a relatively simple task that is also simple to express in `bash`.

The following sample script illustrates some other concepts not yet covered. See Listing 22.22 for the full script. This script is made up of three parts. The first part (lines 61–63) invokes the script based upon the user's call. The second part (lines 48–59) does some basic error checking and then starts the recursive process by interrogating the current subdirectory. Finally, the last part is the recursive function that looks at all files within a given subdirectory. Upon finding a new directory, the `recurse()` function is called again to dig down further into the tree.

When you call the `fut.sh` script, two functions are declared (`recurse()` and `main()`). The script ultimately ends up at line 61, where the main function is called using the first argument passed to the script as the argument passed to `main()`.

In function `main()` (line 48), you begin by storing the current date in the format `YYYY-MM-DD`. This is performed using the `date` command, specifying the desired format in double quotes. You store this result into a variable called `today`. Note that `today` isn't local to `main()`; it can be used in other functions afterward.

You can declare a variable as local to a function. This is useful if you want to store information in a function for recursive uses. To declare a local variable, you simply insert the `local` *keyword before the variable.*

The `main()` function continues by storing the argument (the directory to `recurse`) in variable `checkdir` (line 52). You then test `checkdir` to see if it's empty (has zero length) at line 54. If it is empty, you store `.` to `checkdir`, which represents the current subdirectory. This entire test is done so that if the user passes no arguments, you simply use the current subdirectory as the argument default. Finally, you call the `recurse()` function with the `checkdir` variable (line 58).

The bulk of the script is found in function `recurse()`. This function recursively digs into the directory to find any files that have changed today. The first thing you do is `cd` into the subdirectory passed to you (line 15). Note that when `.` is passed,

you cd into the current directory (in other words, no change takes place). You then iterate through the files in the subdirectory (line 18).

The first thing to check for a given file is to see if it's another directory (line 21). If it is, you simply call the recurse() function (recursively) to dig into this subdirectory (line 22). Otherwise, you check to see if the file is a regular file (line 25). If it is, you perform an ls command on the file, gathering a long time format (line 27). The time style format of this ls command (long-iso) happens to match the format that you gathered in main() to represent the date for today.

At line 29, you search the ls line (longfile) using the date stored in today. This is done through the grep command. You pipe the contents of longfile to grep to search for the today string. If the string is found in longfile, the line simply results; otherwise, a blank line results. At line 31, you check to see if the check variable (the result of the grep) is a nonzero length string. If so, you emit the current file and continue the process at line 18 to get the next file in the directory.

After you have exhausted the file list from line 18, you exit the loop at line 39. You check to see if the directory passed to you was not . . If not, you cd up one directory (because you cd'd down one directory at line 15). If the directory was identified as . (the current directory), you avoid cd'ing up one level.

LISTING 22.22 Files Updated/Created Today Script (on the CD-ROM at ./source/ch22/fut.sh)

```
 1:    #!/bin/bash
 2:    #
 3:    # fut.sh
 4:    #
 5:    #  Find files created/updated today.
 6:    #
 7:    #  Usage is:
 8:    #
 9:    #    fut.sh <dir>
10:    #
11:
12:    function recurse()
13:    {
14:      # 'cd' down into the named directory
15:      cd $1
16:
17:      # Iterate through all of the files
18:      for file in * ; do
19:
20:        # If the file is a directory, recurse
21:        if [ -d $file ] ; then
```

```
22:           recurse $file
23:         fi
24:
25:         if [ -f $file ] ; then
26:
27:           longfile='ls -l —time-style=long-iso $file'
28:
29:           check='echo $longfile | grep $today'
30:
31:           if [ -n $check ] ; then
32:
33:             echo $PWD/$file
34:
35:           fi
36:
37:         fi
38:
39:       done
40:
41:       if [ $1 != . ] ; then
42:         cd ..
43:       fi
44:
45:     }
46:
47:
48:     function main()
49:     {
50:       today='date +%Y-%m-%d'
51:
52:       checkdir=$1
53:
54:       if [ -z $checkdir ] ; then
55:         checkdir=.
56:       fi
57:
58:       recurse $checkdir
59:     }
60:
61:     main $1
62:
63:     exit
```

Scripting Language Alternatives

bash is one option of scripting, but GNU/Linux offers a number of shells and scripting language alternatives. In addition to bash, GNU/Linux offers csh/tcsh (Berkeley C-shell), ksh (the Korn shell), zsh (Z-shell), and others. For higher level scripting, you have the choice of numerous languages, such as Python, Ruby, Perl, Tcl, Scsh, and others. So many languages, so little time . . .

Summary

This chapter took a quick tour of the bash scripting shell. It explored variables in bash, including special variables that describe the environment. The basics of scripting were introduced, along with a demonstration of simple numerical methods in bash. Fundamental concepts in bash were also reviewed, including tests, conditionals, and a number of looping constructs. Methods for input and output in scripts were also discussed, in addition to bash's function specification constructs. Finally, a number of useful scripts were discussed and dissected to illustrate bash scripting.

Resources

"Advanced Bash-Scripting Guide" at http://www.tldp.org/LDP/abs/html/.
"Bash Reference Manual" at http://www.gnu.org/software/bash/manual/bashref.html.
"Bash FAQ" at ftp://ftp.cwru.edu/pub/bash/FAQ.

This page intentionally left blank

23 Editing with sed

In This Chapter

- Quick Introduction to sed
- Discussion of sed Spaces
- Typical sed Command-Line Options
- sed and Regular Expressions
- sed Numerical and Pattern Ranges
- Most Useful sed Commands

Introduction

The sed utility is a very useful utility, but it can also be one of the most cryptic. sed is a stream editor that alters text that flows through it using a number of types of transformations. sed does not alter the original file provided to it but instead provides the transformed text to stdout. sed is one of the oldest tools in UNIX, written in the early 1970s by Lee McMahon. sed operations are stream operations (as the name implies) where simple scripts provide filtering and transformation of text. Consider Figure 23.1, which shows a simple example of sed use with a graphical illustration.

Now it's time to pick apart this sed command and look at it a little further to understand how sed works for this case (the substitute command, s).

The full sed command (for the case shown in Figure 23.1) is illustrated in Figure 23.2.

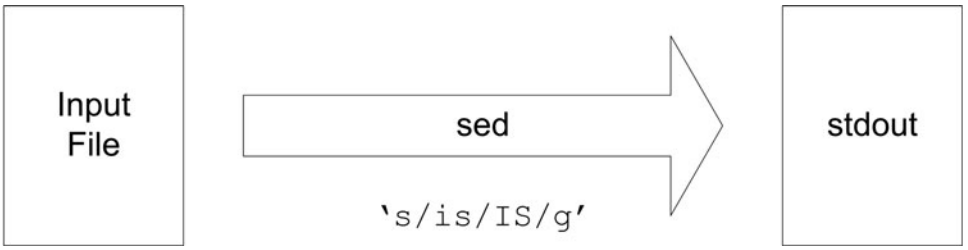

FIGURE 23.1 The `sed` model as a text filter.

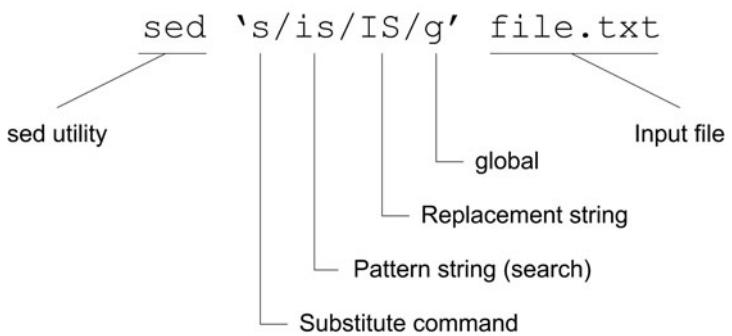

FIGURE 23.2 Anatomy of a simple `sed` invocation.

You invoke `sed` using the `sed` command and then provide a script to use on the input stream (defined by the third parameter, `file.txt`). The `sed` script represents a substitute transformation where a pattern is searched in the input stream (pattern string) and, when found, is replaced by the replacement string. Note that the command (`s`), pattern search string, and replacement string are all delimited by the `/` character. This character can be used for search and replacement; you will take a look at how this is done shortly.

You end the `sed` script with a `g` to indicate that you want to perform the search and replace over the entire stream. If `g` is not provided, only the first occurrence is replaced. You can instead replace `g` with a number, which indicates the particular occurrence to change. For example, ending with `/2` indicates to change only the second occurrence.

Anatomy of a Simple Script

To complete your introduction, you can take look at this script in action. The file shown in Listing 23.1 illustrates the first simple script.

LISTING 23.1 Sample File for sed Illustration (on the CD-ROM at ./source/ch23/file.txt)

```
1:    This is a sample text string which is going to be used.
2:
3:    This is one more string that can be used.
4:
5:    Finally, this is the last string in the test set.
```

Now using the previous sed script on Listing 23.1, you see the following:

```
# sed 's/is/IS/g' file.txt
ThIS IS a sample text string which IS going to be used.

ThIS IS one more string that can be used.

Finally, thIS IS the last string in the test set.

#
```

Each occurrence of is is replaced with IS over the entire file. Consider now what happens if you omit the final g:

```
# sed 's/is/IS/' file.txt
ThIS is a sample text string which is going to be used.

ThIS is one more string that can be used.

Finally, thIS is the last string in the test set.

#
```

Note that instead of replacing each occurrence in the file, it replaced only the first occurrence in each line.

You can create two transforms by using the -e option (which is implied in the earlier example). Consider this example:

```
# sed -e 's/is/IS/g' -e 's/IS/is/g' file.txt
This is a sample text string which is going to be used.

This is one more string that can be used.

Finally, this is the last string in the test set.

#
```

In this case, you convert `is` to `IS`, but then you follow this with the reverse transform. The result is the same as the original file. The key here is the use of the `-e` (script or expression) to provide multiple transformations on the same input text.

`sed` *makes only one pass over the input file and is therefore very efficient.* `sed` *operates by reading a single line from* `stdin`*, executing the series of editing commands on it (potentially a series of commands), and then writing the output to* `stdout`.

Now that you have a quick introduction to `sed`, you can look at some of `sed`'s other capabilities. The `sed` utility can be quite broad and complex. Therefore, the focus in this chapter is on the more useful aspects of `sed`.

sed Spaces (Buffers)

Within `sed`, you find a number of spaces (or buffers) on which `sed` commands operate. Each input line is first copied to the pattern space. The pattern space is the temporary holding space on which `sed` commands are performed. After the provided `sed` scripts have all been performed on the pattern space, the line is copied to the output. Also available in `sed` is the hold space. Within `sed`, you can copy the contents of the pattern space to the hold space and retrieve them at some later time. The hold space is just a temporary buffer, but it can be useful to remember certain lines.

Typical sed Command-Line Options

Rather than specify scripts on the command line, `sed` scripts can also be specified in a file. This can be useful when complex `sed` scripts are needed. To invoke a scripted file with `sed`, simply use the `-f` option to specify the script file.

You can suppress automatic emission of `sed` output by using the `-n` flag. When you look at the print command later in this chapter, you can see where this flag can be useful.

When dealing with very large files, you can achieve better intermediate performance with the `-u` flag. In this unbuffered mode, `sed` loads smaller amounts of data from the input files and flushes data to the output more often. This means better visual performance (you see results more often), but it might represent worse overall performance.

Regular Expressions

A key aspect of sed is its use of regular expressions. A regular expression is a pattern that can match text strings. Regular expressions are a formal language from which very complex patterns can be expressed. Take a look at a few examples in Table 23.1 (using the sed delimiter for completeness):

TABLE 23.1 Pattern Matching Examples with Regular Expressions

`/dog/`	Matches any occurrence of `dog`
`/[a-z]/`	Matches a single character `a` through `z`
`/[a-zA-Z]/`	Matches single characters `a` through `z` and `A` through `Z`
`/[0-9]/`	Matches all single digits (`0` through `9`)
`/0[ab]1/`	Matches `0a1` and `0b1`
`/Z*/`	Matches zero or more occurrences of `Z` (", `Z`, `ZZ`, `ZZZ`, and so on)
`/Z?/`	Matches zero or one instance of `Z` (" `Z`)
`/[^0-9]/`	Matches any single character other than digits
`/t.m/`	Matches any occurrence of `t` separated by one character followed by `m`, such as `tim`, `tom`, and so on

These patterns illustrate a number of special symbols used within regular expressions. For example, the `[]` indicates a range of characters. The `-` symbol is used to define the range. The `*` character indicates that a character might repeat zero or more times. The `?` specifies that one or zero instances of the character is used for the match. The `^` character indicates the characters that are NOT used for the match. Finally, the `.` character matches any character.

Two other regular expressions (called *anchors*) that you explore with sed in this chapter include `^` (different context than mentioned in the previous paragraph) to match at the beginning of the line and `$` to match at the end of the line (see Table 23.2).

TABLE 23.2 Beginning and End Pattern Matching with Anchors

`/^The/`	Matches `"The"` at the beginning of a line
`/end.$/`	Matches `"end."` at the end of a line
`/^T.*\.$/`	Matches lines that start with `T` and end with `.`

Ranges and Occurrences

The lines to be processed by `sed` can be restricted using ranges. A line range can consist of a single-line definition or a range of lines. For example (using the substitute command for illustration):

```
5s/this/that/
```

replaces occurrences of `this` with `that` on line 5. You can specify that only the first five lines are operated upon, such as follows:

```
1,5s/this/that/
```

where `1,5` represents the range of lines one through five. If instead you want the range of line five to the end of the file, this can be provided using the end of file symbol `$`:

```
5,$s/this/that/
```

When you provide no range, it applies to all lines. One final useful modifier is the `!` command. This tells `sed` not to apply to the given line range. Take a look at this with the prior example:

```
5,$!s/this/that/
```

This tells `sed` to apply the substitution to all lines excluding line five through the end of the file.

Rather than specify line ranges, you can also define ranges based upon patterns. Consider the following example:

```
'/^The/s/this/that/'
```

This substitutes `that` for `this` for any line that begins with `The`. You can also perform substitutions between pattern ranges, as follows:

```
'/start/,/end/s/index/idx/g'
```

This replaces `index` with `idx` after a line containing `start` is found and ends the replacement after a line containing `end` is found.

Essential sed Commands

Now that you have some basic details of sed under your belt, you can take a look at some of the most useful commands. The next sections dig in a little further to these commands to explore some of their other uses.

Substitute (s)

The substitute command, as you have explored already, provides a simple search and replacement over the input stream. As this chapter has already investigated this command, this section covers some of the other aspects not yet touched upon. The format of the substitute is as follows:

```
[address1[,address2]]s/pattern/replacement/[n|g]
```

An optional address or address range can be specified before the substitute command. A single address indicates that the substitution is restricted to the particular line or search expression. Two addresses or patterns (separated by a comma) indicate a range for the substitution. The flags (shown at the end of the command specification) are n, representing a number, and g, representing global. If a number is specified, the substitution is performed only on that occurrence. For example:

```
sed '/this/that/' file.txt
```

replaces the first occurrence (of each line) contained in the input file, file.txt. You can achieve the same thing with the script following:

```
sed '/this/that/1' file.txt
```

If you want the second occurrence of this to be replaced with that, you simply specify n as 2. The global flag specifies that all occurrences on all inputs lines are to be replaced.

Delete (d)

The delete command simply deletes the lines that match your defined restriction. For example, if you want to remove the first five lines of a file, you can use the following sed command:

```
sed '1,5d' file.txt
```

You can tell `sed` to delete all but the first five lines using the delete command and reversing the restriction with `!`:

```
sed '1,5!d' file.txt
```

Print (p)

The print command can be thought of as the reverse of delete. One difference is that you must specify the `-n` flag to avoid double-printing the output. To reproduce the earlier delete example of emitting only the first five lines, you use the following:

```
sed -n '1,5p' file.txt
```

You can also instruct `sed` to emit only those lines that contain a particular search string, such as those containing a `:` at the beginning of the line:

```
sed -n '/^:/p' file.txt
```

Appending (a), Inserting (i), and Changing (c) Lines

You can also append, insert, or change entire lines based upon range or pattern. To insert a blank line after a line is found containing `start`, you can perform the following append command:

```
sed '/start/a\ ' file.txt
```

You can insert a blank line at the end of the file using the insert command:

```
sed '$i\' file.txt
```

You can change a line entirely as follows. This script looks for the word `secret` and replaces the entire line with `DELETED`:

```
sed '/secret/c\DELETED' file4.txt
```

Finally, you can perform all three commands using the `{}` symbols to group the commands. The following script emits a start line before the line matching the keyword, a stop line after the matching line, and then `DELETED` for the line itself (on the CD-ROM at `./source/ch23/multi.sed`).

```
sed '/secret/{
i\
—start
```

```
a\
—end
c\
DELETED
}' file.txt
```

The grouping shown here can apply to other commands, but with some restrictions. For example, when the delete command is used, all commands after it in the grouping are ignored because the delete command terminates the editing session.

Quit (q)

The quit command, as the name implies, simply ends the `sed` editing session. This command can be quite useful. Consider the following example:

```
sed '10q' file.txt
```

This command emits the first 10 lines of the `file.txt`, and when the 10th line is reached, the `sed` session is ended. This particular command emulates another useful GNU/Linux command called `head` (which emits the head of a file).

Transformation (y)

You can transform text using the `y` command. This provides for replacing one character for another. You provide two sets of characters: the first refers to the search set and the second the replacement set. For example, if you encounter an `A`, you replace with an `a`, and so on.

To convert all uppercase letters to lowercase, you can use the following:

```
sed 'y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/' file
```

The pattern and replacement strings represent a one-to-one correspondence for which the transformation takes place. For this reason, each string must be of equal length.

Line Numbering (=)

The `=` command is used to emit the current line number. This can be used to emit the line numbers that match a given search string or to simply emit the number of lines in the input file.

For example, if you are interested in which lines contain a given search pattern, you can accomplish this with the following `sed` script:

```
sed -n '/This/=' file.txt
```

This script results in line numbers emitted (one per line) for each line that contains the word `This`.

Holding the Pattern Space (h)

Using the hold buffer, you can store away the pattern space to the hold buffer and then operate on the pattern buffer directly. The following example illustrates emitting both the altered line and the unaltered line. Also illustrated here are comments within `sed` scripts (all characters after the `#` symbol).

```
sed '{
# store the pattern space to the hold buffer
h
# perform the substitution on the pattern space
s/is/IS/
# append the unaltered line to the pattern space
G
}' file.txt
```

This script emits the altered line and follows it immediately with the unaltered line (the line not having been altered by the substitute command).

Summary

While this chapter has only scratched the surface of text processing with `sed`, you've nevertheless seen some of the power provided by this little utility. The chapter took a quick tour of regular expressions and their use in `sed`, as well as the use of numerical and pattern-based ranges to restrict `sed`'s processing. You then reviewed some of the most used `sed` commands, including substitution, printing, deleting, and transforming. The chapter ended with a quick discussion of the use of the hold buffer, which gives `sed` the capability of memory.

Some Useful sed One-Liners

```
# Emit the first 10 lines of a file (such as 'head')
sed 10q file.txt
# Emit a double-spaced version of the file
sed 'G' file.txt
# Emit number of lines in input file
sed -n '$=' file.txt
# Emit the last line of a file
sed '$!d' file.txt
# Emit all lines greater than 30 characters in length
sed -n '/^.\{30\}/p' file.txt
# Emit all non-blank lines
sed '/^$/d' file.txt
# Remove all blank lines at the top of a file
sed '/./,$!d' file.txt
```

Resources

"On the Early History and Impact of Unix Tools to Build the Tools for a New Millennium," from Netizens: *An Anthology*, by Ronda and Michael Hauben, June 12, 1996. Found at http://www.columbia.edu/~rh120/ch001j.c11.

"Sed . . . The Stream Editor" at http://www.cornerstonemag.com/sed/.

This page intentionally left blank

24 Text Processing with awk

In This Chapter

- An Introduction to awk
- Simple awk Scripting
- Complex Applications in awk
- Conditional and Looping Constructs in awk
- awk Built-In Functions

Introduction

The awk programming language is a scripting language that can be used for very simple single-line applications to large applications. awk is a general-purpose language, but it excels at its original task of text processing. This chapter takes a tour of the awk programming language, illustrating by numerous examples how it can be used by the developer. awk takes over where sed left off, but each has its own individual strengths.

Short History

The first version of awk was released in 1977 by Alfred Aho, Peter Weinberger, and Brian Kernighan (its name is the combination of the first letter of the authors' last names). Its first integration was into version 7 AT&T UNIX (as all three creators worked for Bell Labs at the time). awk has gone through a variety of changes in its life. Its syntax and notation borrow both from shell scripting languages and also C.

awk Structure

The higher- level structure of awk programs is conceptually very simple (see Figure 24.1).

FIGURE 24.1 Structure of an awk program.

The structure of an awk program can be split into three sections. The BEGIN section is performed before the first line is read from the input file(s), and the END section is performed after the last line is processed from the input file(s). Between the optional BEGIN and END sections is the awk pattern/action section. For each input file specified, each of the patterns is compared in order, and if a pattern matches, then its associated action is performed.

This chapter splits its discussion of awk into two sections. In the first, you take a look at simple awk programs that can be specified on the command line, and then you look at building more complex awk programs for scripting.

Command-Line awk

You can start by taking a look at some simple awk scripts that demonstrate its behavior. You use the following data in the file missiles.txt (which contains data about Cold War nuclear delivery platforms). The data (Listing 24.1) is delimited with : and contains five fields (missile name, length, weight, range, and speed).

LISTING 24.1 Missile Data for awk Scripts (This information can also be viewed at the Strategic Air Command Web site at http://www.strategic-air-command.com and on the CD-ROM at ./source/ch24/missiles.txt)

```
Thor:65:109330:1725:10250
Snark:67:48147:6325:650
```

```
Jupiter:55:110000:1976:9022
Atlas:75:260000:6300:17500
Titan:98:221500:6300:15000
Minuteman III:56:65000:6300:15000
Peacekeeper:71:195000:6000:15000
```

You can emit lines in much the same way that `sed` did, but include a search expression as the pattern and the print command as the action:

```
# awk '/Thor/{print}' missiles.txt
Thor:65:109330:1725:10250
#
```

Without a pattern, you simply emit the entire file:

```
# awk '{print}' missiles.txt
Thor:65:109330:1725:10250
Snark:67:48147:6325:650
Jupiter:55:110000:1976:9022
Atlas:75:260000:6300:17500
Titan:98:221500:6300:15000
Minuteman III:56:65000:6300:15000
Peacekeeper:71:195000:6000:15000
#
```

Rather than emit the entire line, you can emit selected fields instead. `awk` automatically splits the line (otherwise known as a *record*) into the fields delimited by the colon. So if you want to emit the missile and range for the Thor missile, you can do this as follows:

```
# awk -F: '/Thor/{print $1 ":" $5}' missiles.txt
Thor:10250
#
```

Note that you specify the delimiter as `:` using the `-F` command-line option. Each field is parsed to a `$` variable. The first field is defined as `$1`, the second as `$2`, and so on. The entire record is defined as `$0`.

You can add additional text to make your output more reasonable by simply including more text for the print command (the command is actually one line):

```
# awk -F: '/Thor/{print "Missile " $1 " has a range of " $5 "
miles"}' missiles.txt
Missile Thor has a range of 10250 miles
#
```

Arithmetic expressions are also possible on the data. Consider this example, which emits those missiles that have a range of 12,000 miles or more:

```
# awk -F: '$5 > 12000 { print $1 }' missiles.txt
Atlas
Titan
Minuteman III
Peacekeeper
#
```

In this example, your pattern is the test of `$5` (the range field) being greater than 12,000. When this test pattern is satisfied, your action is to emit the first field (the name of the missile).

`awk` provides a number of built-in variables that can be useful. For example, if you want to know the number of records in the file, you can use the optional `END` section with the `NR` built-in variable (number of the record):

```
# awk 'END { print NR }' missiles.txt
7
#
```

Note here that you emit `NR` at the end, so it's the total number of records that are in the file. If you emit `NR` at each line, it's the number of that given line, such as follows:

```
# awk -F: '{ print NR, $0 }' missiles.txt
1 Thor:65:109330:1725:10250
2 Snark:67:48147:6325:650
3 Jupiter:55:110000:1976:9022
4 Atlas:75:260000:6300:17500
5 Titan:98:221500:6300:15000
6 Minuteman III:56:65000:5300:15000
7 Peacekeeper:71:195000:6000:15000
#
```

Given the range and speed data, you can calculate how long it takes to reach its maximum target (roughly calculated as range over speed). This is provided as follows:

```
# awk -F: '{ printf "%15s %3.2f\n", $1, $4/$5}' missiles.txt
           Thor 0.17
          Snark 9.73
        Jupiter 0.22
          Atlas 0.36
          Titan 0.42
  Minuteman III 0.35
    Peacekeeper 0.40
#
```

This example demonstrates simple arithmetic (`$4/$5` to compute the time to target) but also the use of `printf` within `awk`. Rather than simply print the results (as you've done in previous examples), you use the `printf` command to provide a more structured output. You specify size and alignment for the string (missile name) and also the format of the time-to-target result. From this data, you can see that the Snark has the longest time of flight (it's also the slowest of the missiles shown here), and Thor the least.

Now take a look at one final example in this command-line section that demonstrates a bit more of the arithmetic properties of `awk`. Say that you have one of each of these missiles, and you want to know their combined weight. This is easily calculated, using each of the three `awk` sections, as follows:

```
# awk 'BEGIN {FS=":"} {wt += $3} END {print wt}' missiles.txt
1008977
#
```

Now take a look at each of the three sections to see what's going on. In the first section (`BEGIN`), you specify the field separator (using the built-in `FS` variable). You can also specify this on the command line (with the `-F` option), but the use this example demonstrates makes it part of the script and is therefore less error prone. For each record that you find, you sum the weight field (field 3). Note that you did not initialize the `wt` variable, as `awk` automatically initializes it to zero when it's created. After you have processed the last record, the `END` section is performed where you simply print the weight total (just a tad over 1,000,000 pounds or 457,664.27 kilograms).

You've looked at some of `awk`'s built-in variables so far (such as `FS` and `NR`). These and other built-in variables are available for use. A list of some of the most useful is shown in Table 24.1.

TABLE 24.1 awk's Built-In Variables

Variable	Description
NR	Input record number
NF	Number of fields in the current record
FS	Field separator (default space and tab)
OFS	Output field separator (default space)
RS	Input record separator (default newline)
ORS	Output record separator (default newline)
FILENAME	Current input filename

Scripted awk

Now it's time dig in further and explore some of awk's other capabilities. This section moves beyond the simple single-line scripts and looks at how applications can be developed in awk.

Scripting applications in awk allows you to build bigger and more complex applications. You can start with an update to the previous application (printing the total weight of all missiles). This example sums all of the numeric data and adds headers and trailers to the data (see Listing 24.2).

LISTING 24.2 Expanding the Summing Application (on the CD-ROM at ./source/ch21/tabulate.awk)

```
BEGIN {

  FS = ":"
  printf "\n           Name    Length     Range"
  printf "     Speed    Weight\n"

}

{
  printf "%15s  %8d  %8d  %8d  %8d\n", $1, $2, $4, $5, $3

  len += $2
  wt  += $3
  rng += $4
```

```
15:       spd += $5
16:
17:     }
18:
19:     END {
20:
21:       printf "\n          Totals    ———   ———"
22:       printf "  ———   ———\n"
23:       printf "                  %8d  %8d  %8d  %8d\n\n",
24:              len, rng, spd, wt
25:
26:     }
```

Listing 24.2 illustrates the three awk sections. You define the field separator and emit a header line at lines 3–5 within the BEGIN section. Next, for each record in the input file, you emit the fields of the record in an order different than that of the original itself (note line 10). Lines 10–15 are performed for each record (because no pattern exists here, only an action that defaults to each record). Lines 12–15 simply sum each of the fields for which you desire an accumulation. You keep track of the total lengths (len), total weight (wt), total range (rng), and finally total speed (spd). The END section (which is performed after the last record is processed) emits the totals. Note the use of printf here to better control the format of the output.

You invoke this script (called tabulate.awk) as follows (with sample results shown given the input file from Listing 24.1):

```
# awk -f tabulate.awk missiles.txt
            Name    Length     Range     Speed    Weight
            Thor        65      1725     10250    109330
           Snark        67      6325       650     48147
         Jupiter        55      1976      9022    110000
           Atlas        75      6300     17500    260000
           Titan        98      6300     15000    221500
  Minuteman III         56      5300     15000     65000
     Peacekeeper        71      6000     15000    195000
          Totals      ———      ———      ———      ———
                       487     33926     82422   1008977
#
```

Granted, the data is meaningless, but it illustrates how you can process the input data and format the output data.

Now it's time to update the application to find the extremes of the data. In this example, you store the missile that is the longest, heaviest, has the longest range, and is the fastest. This example illustrates a very interesting aspect of awk that is not

provided in many other languages that you use every day: dynamic and associative arrays.

Listing 24.3 shows the new application, which is similar to the original in Listing 24.2. In the `BEGIN` section (lines 1–7) you set up the field separator and then emit the table header.

For each record in the file, you have a default action (lines 9–29). You check each of the test elements (length, weight, range, and speed), and if you find one that exceeds the current (default of zero), then you save it and the current record. Recall that numeric variables are automatically initialized to zero. Note here that saving the current record is done with an associative array that is also dynamic. You have not declared the array (`saved`) or its size. The index is a string that identifies the particular record of interest. Note that you can remove an element from the associative array using the `delete` command, such as in the following:

```
delete saved["longest"]
```

which removes the entry identified by the `longest` index.

After the last record is processed, you perform the `END` section (lines 31–52). Here you simply emit the data stored from the previous extreme's capture. Note the use of the `split` command, which provides the means to split a line into its individual fields (as is done automatically when the record is read). The `split` command takes a string (stored in the associative array) and another variable that represents the array of split elements.

LISTING 24.3 Finding and Storing the Extremes (on the CD-ROM at `./source/ch21/order.awk`)

```
 1:    BEGIN {
 2:
 3:      FS = ":"
 4:      printf "\n           Name    Length     Range"
 5:      printf "     Speed    Weight\n"
 6:
 7:    }
 8:
 9:    {
10:      if ($2 > longest) {
11:        saved["longest"] = $0
12:        longest = $2
```

```
13:       }
14:
15:       if ($3 > heaviest) {
16:         saved["heaviest"] = $0
17:         heaviest = $3
18:       }
19:
20:       if ($4 > longest_range) {
21:         saved["longest_range"] = $0
22:         longest_range = $4
23:       }
24:
25:       if ($5 > fastest) {
26:         saved["fastest"] = $0
27:         fastest = $5
28:       }
29:     }
30:
31:     END {
32:
33:       printf "————————  ————  ————"
34:       printf "  ————  ————\n"
35:
36:       split( saved["longest"], var, ":")
37:       printf "%15s  %8d  %8d  %8d  %8d (Longest)\n\n",
38:              var[1], var[2], var[4], var[5], var[3]
39:
40:       split( saved["heaviest"], var, ":")
41:       printf "%15s  %8d  %8d  %8d  %8d (Heaviest)\n\n",
42:              var[1], var[2], var[4], var[5], var[3]
43:
44:       split( saved["longest_range"], var, ":")
45:       printf "%15s  %8d  %8d  %8d  %8d (Longest Range)\n\n",
46:              var[1], var[2], var[4], var[5], var[3]
47:
48:       split( saved["fastest"], var, ":")
49:       printf "%15s  %8d  %8d  %8d  %8d (Fastest)\n\n",
50:              var[1], var[2], var[4], var[5], var[3]
51:
52:     }
```

The sample output, given the previous data file, is shown in the following:

```
# awk -f order.awk missiles.txt
        Name    Length     Range     Speed    Weight
        ____    ____       ____      ____     ____
       Titan        98      6300     15000    221500 (Longest)
       Atlas        75      6300     17500    260000 (Heaviest)
       Snark        67      6325       650     48147 (Longest Range)
       Atlas        75      6300     17500    260000 (Fastest)
#
```

awk does provide some shortcuts to simplify the application. Consider the following replacement to the END section of Listing 24.3 (see Listing 24.4).

LISTING 24.4 Replacement of the END Section of Listing 24.3 (on the CD-ROM at ./source/ch21/order2.awk)

```
 1:    END {
 2:
 3:      printf "————————  ————  ————"
 4:      printf "  ————  ————\n"
 5:
 6:      for (name in saved) {
 7:
 8:        split( saved[name], var, ":")
 9:        printf "%15s  %8d  %8d  %8d  %8d (%s)\n\n",
10:               var[1], var[2], var[4], var[5], var[3], name
11:
12:      }
13:
14:    }
```

This example illustrates awk's for loop, but using an index other than an integer (what you might commonly think of for iterating through a loop). At line 6, you walk through the indices of the saved array (longest, heaviest, longest_range, and fastest). Using name at line 8, you split out the entry in the saved array for that index and emit the data as you did before.

Other awk Patterns

The awk programming language can be used for other tasks besides file processing. Consider this example that simply emits a table of various data (Listing 24.5).

Here you see an awk program that processes no input file (as the code exists solely in the BEGIN section, no file is ever sought). You perform a for loop using an integer iterator and emit the index, the square root of the index (sqrt), the natural logarithm of the index (log), and finally a random number between 0 and 1.

LISTING 24.5 Generating a Data Table (on the CD-ROM at ./source/ch21/table.awk)

```
1:    BEGIN {
2:      for (i = 1 ; i <= 10 ; i ++ ) {
3:        printf( "%2d %f %f %f\n", i, sqrt(i), log(i), rand() )
4:      }
5:    }
```

You can use a while loop instead of a for loop, as shown in Listing 24.6.

LISTING 24.6 Generating a Data Table Using a while Loop (on the CD-ROM at ./source/ch21/table2.awk)

```
1:    BEGIN {
2:      i = 1
3:      while (i <= 10) {
4:        printf( "%2d %f %f %f\n"", i, sqrt(i), log(i), rand() )
5:        i++
6:      }
7:    }
```

So awk gives you the basic looping and control constructs that you expect from a high-level language, but within a pattern-matching architecture.

This tour hopefully gives you a taste for the capabilities of the awk programming language, but you can find much more. awk provides a number of other built-in functions for reading a line from the input file (getline), searching for a substring within a string (index), returning the length of a string (length), and a sprintf command for string formatting.

Summary

This chapter took a quick tour of the awk programming language. It explored a number of text-processing applications using both one-line and multiline scripts. It also investigated the variety of control forms available in awk inherited from the C language, including loop constructs and conditionals.

Useful awk One-Liners

awk is a great language for useful one-line programs. The following are a number of useful awk scripts that can be coded in a single line.

```
# Emit every line that is not blank
awk 'NF > 0 {print}' file.txt
# Emit the number of lines in a file
awk '{num_lines++} END{print num_lines}' file.txt
# or (another number of lines example)
awk 'END { print NR }' file.txt
# Emit the first 10 lines of a file (like head)
awk 'NR < 11 { print $0 }' file.txt
# Print each line, preceded by its line number
awk '{print NR, $0}' missiles.txt
# Count the number of lines that contain 'PATTERN'
awk '/PATTERN/{num++} END{ print num }' file.txt
```

.

25 Parser Generation with `flex` and `bison`

In This Chapter

- An Introduction to Lexical Analysis
- An Introduction to Parser Generation
- The `flex` and `bison` Utilities
- The `flex` and `bison` Specification Files
- Lexer and Parser Integration

Introduction

The topic of parser construction using two very well known tools is the focus of this chapter. It first takes a quick tour of lexical analysis and grammar processing and then investigates the respective tools. The chapter investigates a few examples, including a simple firewall configuration parser. The `flex` (fast lexical analyzer generator) and `bison` (GNU parser generator) are the focus of this chapter.

Lexical Analysis and Grammar Parsing

This section begins with a short introduction to parser construction. A parser gives you the ability to process a file that has a known structure and grammar. Rather than build this parser from scratch, you can use tools to specify the parser. This is both faster and a less error prone approach to parser construction and is therefore very useful.

Parsers are very useful, and you will develop many of them in your career. Though few of you will take on the task of building full-featured compilers, parsers

are useful in a variety of areas. For example, configuration files for larger applications can require complex parsers to be built to describe how the application's behavior is specified. The techniques discussed here make building these kinds of parsers a snap.

The first task in parsing is the tokenization of your input file. In this phase, you break your input file down into its representative chunks. For example, breaking down the C source fragment:

```
if (counter < 9) counter++;
```

results in the tokens parsed:

```
'if', '(', 'counter', '<', '9', ')', 'counter', '++', ';'
```

That's quite a few tokens, but you will see why this is necessary shortly. Identifying how the tokens are made up is the specification part of lexical analysis. Further, the lexical analyzer returns metadata that describes what was parsed. For example, rather than the tokens, additional data is returned, such as:

```
IF_TOKEN  OPEN_PAREN  VARIABLE  OP_LESSTHAN  NUMBER  CLOSE_PAREN
VARIABLE  UNARY_INC  SEMICOLON
```

This is useful because when you're trying to understand whether the tokens have meaning, it's not important which variable you're dealing with, but just that you have a variable (identified by the `VARIABLE` token metadata).

After you've broken down your input file into tokens, these tokens can be passed to your grammar parser to understand if they are meaningful. For example, consider the following simple grammar to define an `if` statement:

```
IF_STMT:
    IF_TOKEN OPEN_PAREN TEST CLOSE_PAREN EXPRESSION SEMICOLON
```

This defines an `if` statement rule (`IF_STMT`) that specifies that you'll have an `IF_TOKEN` (`if`), a test expression surrounded by parens (`OPEN_PAREN TEST CLOSE_PAREN`), followed by an `EXPRESSION` terminated by a `SEMICOLON`. Further:

```
TEST:
    VARIABLE  OPERATOR  NUMBER
OPERATOR:
    OP_EQUALITY  |  OP_LESSTHAN |  OP_GREATERTHAN
```

defines that your test is represented by a variable, an operator (one of three), and a number. Finally, the simple `EXPRESSION` is:

```
EXPRESSION:
    VARIABLE UNARY_OP
UNARY_OP:
    UNARY_INC | UNARY_DEC
```

Now that you have your simple grammar rules specified, take a look at a couple of test fragments to see how it works. Consider your original code fragment. Figure 25.1 illustrates this `if` statement with its parse tree.

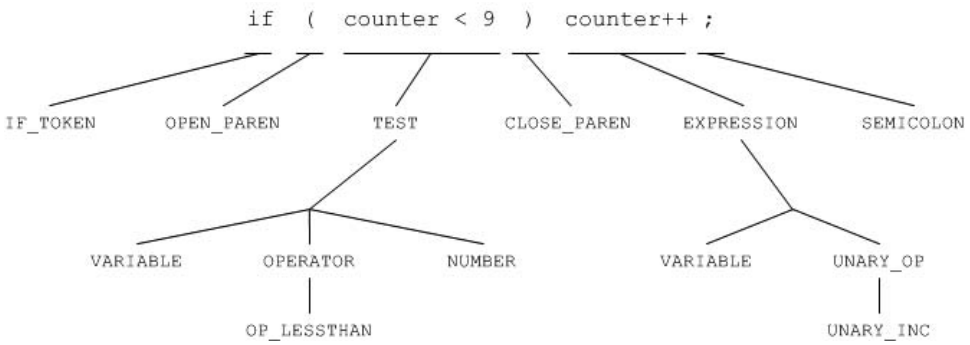

FIGURE 25.1 Parse tree for a sample C code fragment.

You can see in this diagram that each token from the C fragment is correctly recognized by our sample grammar. Each token is matched with the syntax and is therefore a proper line of C (in this simple example). Now take a look at another example, but in this case, an erroneous one (Figure 25.2).

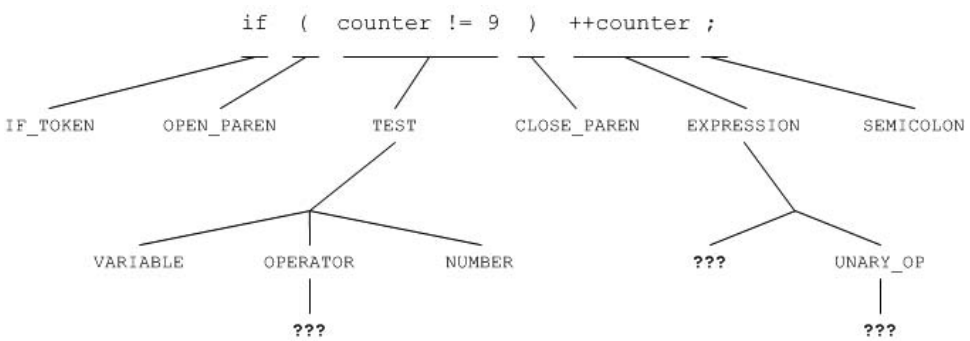

FIGURE 25.2 Incomplete parse tree for an erroneous C code fragment.

In this case, two errors are found. First, you parse down the TEST subtree and find an issue. The TEST is made up of a VARIABLE followed by an OPERATOR and ends with a NUMBER. You find the VARIABLE (counter), but upon trying to recognize the OPERATOR, you find something that is not matched (!=). Because an invalid OPERATOR was defined, you signal an error and exit (or more intelligently identify the error, but then try to move on with the parse to see if any other errors are found). In the next case, you expect an EXPRESSION following the CLOSE_PAREN. You try to recognize the EXPRESSION as a VARIABLE and a UNARY_OP, but instead find a UNARY_OP and then a VARIABLE. Both elements are recognized, but syntactically, they represent an error. Instead of first finding a VARIABLE, you see a UNARY_OP and signal the error.

Note that the tree illustrates two concepts in parsing. The endpoints of the tree (leaves) represent the symbols of the grammar, whereas the edges represent the derivation of the grammar to the endpoints. When a sequence of tokens is properly matched in the tree to a set of nodes, the tokens are recognized by the grammar.

Lexer and Parser Communication

A typical compiler contains numerous phases (commonly six). The lexical analyzer and grammar parser are two of the phases in compiler construction, but for the purposes here, they are all that are required. For the parsing of configuration files, you need to break the file down into tokens and then parse these into a parse tree. This is illustrated in Figure 25.3.

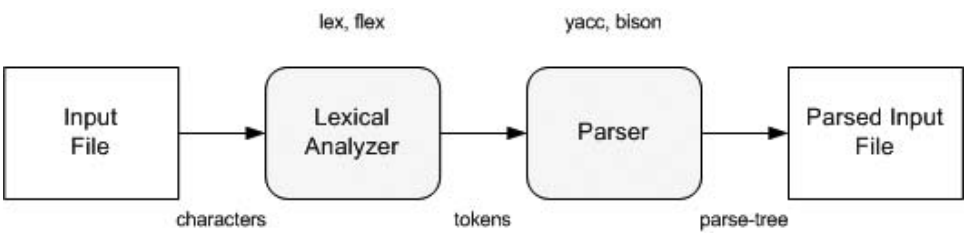

FIGURE 25.3 Typical phases in configuration parsing.

Given your input file, the lexical analyzer (or lexer) takes the file as characters and assembles them into tokens. The grammar parser takes the tokens and parses them into parse trees. Based upon the successful creation of the parse tree, you have syntactically correct code (or configuration). From here, you can utilize the parse tree to extract your configuration data.

The lexical analyzer (`flex`) takes a file of regular expressions and produces a finite state machine that recognizes the sequence of tokens for the target language. The parser generator (`bison`) takes a file defining a context free grammar and produces an LALR parser that recognizes the language.

While the focus here is on `flex` and `bison`, some might recognize these tools by their early UNIX names `lex` and `yacc` (yet-another-compiler-compiler). `flex` and `bison` are GNU projects; therefore this chapter utilizes them instead. The next sections look at the each of the tools separately and then bring them together in a single configuration example.

flex

The fast lexical analyzer generator (or `flex`) is a tool that uses a collection of regular expressions provided by the user to produce a lexical analyzer. The produced application can then be used to tokenize an input file from a sequence of characters to a sequence of tokens. The generated application is really nothing more than a finite state automaton derived from the set of regular expressions.

Now take a look at a simple example to understand what all of this means. First, a `flex` input file has the format:

```
%{
<C declarations>
%}
<definitions>
%%
<rules>
%%
<user code>
```

The first section introduces a set of C declarations into the resulting lexer. Next are definitions or simple name declarations that are used in the rules section (think of them as symbolic constant regular expressions). This section also contains start conditions for the lexical analysis, which can be used to conditionally activate rules in the rules section. Definitions have the form:

```
name        definition
```

The rules section defines the set of regular expressions used to tokenize the input. These rules define the finite automata to parse the tokens and take the form:

```
pattern        action
```

Finally, the last section is user code that is integrated into the resulting lexical analyzer code. Now on to a real example. In Listing 25.1 you see a simple `flex` input file.

LISTING 25.1 Simple `flex` Input File Example (on the CD-ROM at `./source/ch25/simple/example.fl`)

```
 1:    %{
 2:    #include <stdio.h>
 3:    %}
 4:
 5:    NUM       [0-9]
 6:    VAR       [a-zA-Z]
 7:    WS        [ \t]
 8:
 9:    %%
10:    set       printf( "(STMT set) " );
11:    {NUM}+    printf( "(NUM %s) ", yytext );
12:    {VAR}+    printf( "(VAR %s) ", yytext );
13:    -         printf( "(OP minus) " );
14:    \+        printf( "(OP plus) " );
15:    =         printf( "(OP equal) " );
16:    ;         printf( "(END stmt) " );
17:    \n        printf( "\n\n" );
18:    {WS}+     /* Ignore whitespace */
19:    <<EOF>>   printf( "End of parse\n." ); yyterminate();
20:    %%
```

Note first the three sections in Listing 25.1. You see the definitions section (everything prior to the first `%%`). At line 2, you see some C code that will be integrated into the final lexer. Because you intend to use some standard C library functions within your rules section, you tell `flex` to include the `stdio.h` header file so that these symbols (such as `printf`) can be resolved. You then provide three definitions that are later in your rules section (you can also think of these as aliases). First, the `NUM` definition defines a regular expression for a number. This defines that a number is a digit between 0 and 9. Next is the `VAR` definition, which specifies that variables are made up of lower- and uppercase letters. Finally, the `WS` definition indicates that whitespace is space or tab characters. Note that these are simply definitions; the rules section utilizes these to further specify tokens in your simple language.

The next section (between the two `%%` tags) identifies the rules. These represent patterns (regular expressions, herein called regex) and an action to take when the pattern is found. Consider the first rule at line 10. The rule defined here refers to the

discovery of the token set. When the set is found, you perform the printf (to emit (stmt SET)). The next rule is a regular expression that's used to match numbers of arbitrary length. You use your definition of NUM (in braces, as required by flex) and follow it with a plus. The plus indicates that you are looking for one or more of these digits. So, if you find a number (0–9) and then a character that is not a number, the match is complete, and you perform the action. In this action, you emit the value that you've recognized, which flex stores in a character string called yytext. The next rule at line 12 is similar to your number rule but instead (per the definition at line 6) collects alpha characters.

At lines 13 and 15, you see some very simple recognizers. When you see a - symbol, you emit a minus string, and when you see a + (escaped with \ to delineate it from the one or more regex symbol), you emit a plus string. Line 17 looks for newlines and simply emits a newline to stdout.

One final point to note is line 19. This special symbol recognizes the end of the file, for which you simply end the session using the supplied yyterminate function.

Take a look now at how you can build and then test this lexer using stdin. To build your lexer, you simply provide the file (example.fl) to the flex utility, as follows:

```
$ flex example.fl
```

The result is a file called lex.yy.c, which contains the lexer. You can now compile this file as follows:

```
$ gcc -o example lex.yy.c -lfl
```

The trailing -lfl tells gcc to link in the flex library (libfl.a). Now you have an executable for which you can test. Take a look at a few examples typed from the shell (see Listing 25.2).

LISTING 25.2 Test of Simple Lexer Specified in Listing 25.1

```
 1:    $ ./example
 2:    set counter = 0;
 3:    (STMT set) (VAR counter) (OP equal) (NUM 0) (END stmt)
 4:
 5:    set counter=0;
 6:    (STMT set) (VAR counter) (OP equal) (NUM 0) (END stmt)
 7:
 8:    set set = set;
 9:    (STMT set) (STMT set) (OP equal) (STMT set) (END stmt)
10:
```

```
11:     set a = a + 1;
12:     (STMT set) (VAR a) (OP equal) (VAR a) (OP plus) (NUM 1) (END stmt)
13:
14:     abc123def
15:     (VAR abc) (NUM 123) (VAR def)
16:
17:     End of parse
18:     $
```

At line 1, you start the lexer and then begin providing input. Note at line 18 in Listing 25.1 that you ignore any whitespace that's encountered. This is why lines 2 and 5 of Listing 25.2 result in an identical set of tokens being generated. At line 8, you see the `set` reserved word used as a variable. The lexer simply parses this as a statement, which would presumably be caught by the grammar parser. The point here is that the lexer doesn't know and therefore simply parses the tokens as it sees them (even though they might not make sense).

One other item to note (to further clarify how the lexer works) is the input shown at line 14. In this case, no whitespace is provided to delimit the input. The lexer correctly notes that alpha characters are `VAR` tokens and numeric characters represent `NUM` tokens. In this case, the three tokens are correctly parsed from the input without any reference to delimiters.

Finally, at line 16 of Listing 25.2, you perform a Ctrl+D to end the parse. The lexer catches this (via the `<<EOF>>` symbol) and ends the parse session.

In these examples, the actions to the rule patterns have been to write to `stdout`. Later in this chapter, you'll take a look at how to connect the lexical analyzer to the grammar parser.

bison

Now that you have some understanding of the lexical analysis process, it's time to dig into grammar parsing. You evolve the last lexer to work with a new grammar parser built with `bison`.

The grammar parser that you build with `bison` works in concert with the lexical analyzer. It accepts the tokens recognized by the lexer and matches them with the grammar symbols to identify a properly structured input. The `bison` tool builds a bottom-up parser and, using a process known as shift-reduce, attempts to map all of the lexer data elements to grammar symbols.

A Simple Grammar

Now, take a first look at a simple example that models your input from Listing 25.2. What you want your grammar to represent is a set of set operations. These can take the form as follows:

```
set counter = 1;
set counter = counter + 1;
set counter = lastcount;
set delta = counter - lastcount;
```

In this very simple grammar, you can reduce to the following rules. First each statement starts with a set command followed by a variable and an = symbol. After the assignment operator, you have what you call an expression and terminate with a ;. So your first rule can take the form:

```
'set' VARIABLE '=' EXPRESSION ';'
```

Your EXPRESSION has one of four forms:

```
NUMBER
VARIABLE
VARIABLE OPERATOR NUMBER
VARIABLE OPERATOR VARIABLE
```

Finally, you support two operators within an expression. It can be either an addition or a subtraction:

```
'+'
'-'
```

This is reasonable so far for this very simple grammar. It should be clear now how the lexer and grammar parser work together. The lexer breaks the input down into the necessary tokens, which the grammar parser takes, and using its rules, determines if the symbols can be recognized by the grammar.

Encoding the Grammar in bison

Now it's time to look at how the simple grammar can be represented for bison. The bison input file is similar to the flex input file. It starts with a section containing C code that is imported into the grammar parser, followed by a set of declarations:

```
%{
C Declarations
%}
Bison Declarations
%%
Grammar Rules
%%
C Code
```

The `bison` input file can be quite a bit more complex than the `flex` input file, so it's time to continue the illustration with a real example. This example provides a grammar for the previous `set` example. This is shown in Listing 25.3.

LISTING 25.3 `bison` Grammar File for Our Simple `set` Example (on the CD-ROM at `./source/ch25/setexample/grammar.y`)

```
 1:    %{
 2:    #include <stdio.h>
 3:
 4:    void yyerror( const char *str )
 5:    {
 6:            fprintf( stderr, "error: %s\n", str );
 7:    }
 8:
 9:    int main()
10:    {
11:            yyparse();
12:
13:            return 0;
14:    }
15:    %}
16:
17:    %token SET NUMBER VARIABLE OP_MINUS OP_PLUS ASSIGN END
18:
19:    %%
20:
21:    statements:
22:            | statements statement
23:            ;
24:
25:
26:    statement:
27:            SET VARIABLE ASSIGN expression END
28:            {
```

```
29:                             printf("properly formed\n");
30:             }
31:             ;
32:
33:
34:     expression:
35:             NUMBER
36:             |
37:             VARIABLE
38:             |
39:             VARIABLE operator NUMBER
40:             |
41:             VARIABLE operator VARIABLE
42:             ;
43:
44:
45:     operator:
46:             OP_MINUS
47:             |
48:             OP_PLUS
49:             ;
```

In the first section of C declarations (lines 1–15), you see code that is to be ported to the generated parser. You include any header files that are referenced in this C declarations section or in code in the rules section (such as the `printf` at line 29). Also of note are two functions that you provide here for the parser. The first is a special function called `yyerror` that is called by the generated parser if an error occurs. Next, you see the `main` function. Normally, you would have your own `main` and call the parser, but for this simple example, you embed the `main` within the parser itself. The `main` calls the function `yyparse`, which is the grammar parser. The grammar parser invokes the lexical analyzer internally.

In the next section, `bison` declarations, you see a special symbol called `%token`. This identifies all the tokens that are used by the parser to recognize the grammar. The lexical analyzer must know about these as well and must be built to return them. You will have another look at the lexer shortly and see how it's changed to support connectivity with the parser.

The bulk of your `bison` input file is the rules that make up the grammar. The rules section begins at the first `%%` symbol in the file (line 19). Your first rule is the `statements` rule (lines 21–23). A rule is made up of a name followed by a colon and then a series of tokens or nonterminals with an optional action sequence (within a set of braces), terminated by a `;`. The `statements` rule has two possibilities. The blank after the colon means that there might be no more tokens to parse (end of

file) or (|) or another possibility. If you had placed `statement` as the only possibility, then you could parse a single statement and no more. By specifying `statements` before `statement`, you allow the possibility of parsing more than one statement in your input.

The `statement` rule (lines 26–31) permits the parse of a single kind of statement (a variable assignment). Your statement must begin with a `set` command followed by a variable name and an assignment operator. You then see an expression rule, which covers a number of different possibilities. The statement ends with a semicolon (represented here as an `END` token). Note here that you provide an optional action sequence for the statement. If your rule matches without encountering any errors, the optional command is performed within the parser. Here you simply emit that the input provided was properly formed within the grammar.

The `expression` rule (lines 34–42) defines the varieties of expressions that are possible (the right value, or `rvar`, of your statement). Four possibilities are considered legal. It can be a number, another variable, a sum of a variable and a variable or number, or a difference between a variable and a variable or number. The `operator` rule (lines 45–49) covers the two types of operators that are legal (a `-` as defined by the `OP_MINUS` token or a `+` as defined by `OP_PLUS`).

The goal of this parse, as defined by the code that you have inserted into the grammar, is simply to identify a properly formed statement.

`bison` grammars are expressed in a machine-readable Backus-Naur Form (or BNF). A BNF is a formal syntax used to express context-free grammars and is a widely used notation to express grammars for computer programming languages.

Hooking the Lexer to the Grammar Parser

Now that you have your grammar parser done, it's time to look at how to modify the lexer so that both know about the proper set of tokens and also how to connect the lexer to the parser (see Listing 25.4). You can start by looking at the upgraded `flex` input file and reading a description of the changes to support the connectivity.

LISTING 25.4 Upgraded `flex` Input File (on the CD-ROM at `./source/ch25/setexample/tokens.l`)

```
1:    %{
2:    #include <stdio.h>
3:    #include "grammar.tab.h"
4:    %}
5:
```

```
 6:     NUM     [0-9]
 7:     VAR     [a-zA-Z]
 8:     WS      [ \t]
 9:
10:     %%
11:     set             return SET;
12:     {NUM}+          return NUMBER;
13:     {VAR}+          return VARIABLE;
14:     -               return OP_MINUS;
15:     \+              return OP_PLUS;
16:     =               return ASSIGN;
17:     ;               return END;
18:     \n              /* Ignore whitespace */
19:     {WS}+           /* Ignore whitespace */
20:     %%
```

The first point to note is that to know which tokens (constant names) the parser generator is using you must include them in the lexer. When the parser generator is built with `bison`, it also generates a header file that contains the tokens that were specified (via the `%token` symbol). These can now be included within the `flex` input file to connect them together. In the compile phase of the lexer, you can easily find if you have disconnects (because of missing symbol errors).

The second point to note is that you're no longer just printing the token that was recognized, but instead returning the token to the caller. The caller in this case is the parser generator.

Now it's time to walk through the process of building a parser from both the `flex` and `bison` input files. In this example, the `bison` grammar input file is named `grammar.y`, and the `flex` input file is called `tokens.l`. You start by building the parser itself because the lexer is dependent upon the header file generated here:

```
# ls
grammar.y  tokens.l
# bison -d grammar.y
# ls
grammar.tab.c  grammar.tab.h  grammar.y  tokens.l
#
```

You tell `bison` (via the `-d` flag) to generate the extra output file (which contains your macro definitions) so that you can hook it up to the lexer. You see here that two files are created: `grammar.tab.c` and `grammar.tab.h`. The C source file is the parser source, and the `.h` file the macro definitions file.

Next, it's time to build the lexer using `flex`:

```
# flex tokens.l
# ls
# grammar.tab.c  grammar.tab.h  grammar.y  lex.yy.c  tokens.l
```

You invoke `flex` with your `flex` input file, which results in a new file called `lex.yy.c`. This is your lexical analyzer. Finally, you can build these together with `gcc` and generate the parser as follows:

```
# gcc grammar.tab.c lex.yy.c -o parser -lfl
```

Now you have your parser executable in a file called `parser`. You can try it out and see how it works (see Listing 25.5).

LISTING 25.5 Sample Use of the `set` Parser

```
$ ./parser
set counter = 1;
properly formed
set counter = counter + 1;
properly formed
set counter   =   lastCount;
properly formed
set deltaCount = counter - lastCount;
properly formed
set set = set;
error: parse error
$
```

You can see from this use that it properly recognizes your grammar. When you encounter an error, it simply exits with a status. You can also exit the parser by pressing Ctrl+D.

Note that the last example, where an error was detected, presents an interesting case. The `set` token is valid as far as variables go (given the variable rule in the lexer at line 13 of Listing 25.4), but given the precedence of your regular expressions, it's detected as the `SET` token rather than a variable. Given the grammar, this is defined as an error.

So now you have built a parser that identifies whether the input is correct given the tokens that can be recognized and the grammar. The next step is actually doing something with the data, which you look at in the next section.

Building a Simple Configuration Parser

Now it's time to continue with another example of parser construction with `flex` and `bison`, but in this case, you use the data that's parsed. This example creates a parser for an e-mail firewall configuration that defines from whom you accept e-mail and also from whom you reject e-mail. In this example, mail that's not specified as allowed but also not explicitly specified as disallowed is simply quarantined for later review by the e-mail recipient. The goal of such an e-mail firewall is to quickly identify e-mail that you expect, disallow e-mail that you don't want, and then cache the rest for later review.

The configuration file is illustrated in Listing 25.6. You have two sections that define those e-mail addresses that you allow (those that you immediately allow to make it to the recipient) and also those that you disallow (reject or ignore).

LISTING 25.6 Sample E-Mail Firewall Configuration File (on the CD-ROM at `./source/ch25/config/config.file`)

```
allow {

  mtj@mtjones.com
  dan_5422@yahoo.com
  albert@camus.com

}

disallow {

  spammer@spamcorp.com
  viagera@pharma.com

}
```

The structure of this file (Listing 25.6) is very simple. Those e-mail addresses in the `allow` section are permitted, whereas those in the `disallow` section are rejected.

The parser is constructed in the two phases that were demonstrated in the first example earlier in the chapter. The configuration file lexer is defined in a file called `config.l`, and the grammar parser in `config.y`. Rather than taking your configuration file from `stdin`, in this example you allow the configuration file to be read from a file.

Configuration File Lexical Analyzer

The lexer is the first phase of your configuration file parser. It breaks down the file into its basic elements and returns them to the grammar parser. The tokens that you permit, from Listing 25.6, are minimal and consist of two reserved words (`allow` and `disallow`), section delimiters (`{` and `}`), and finally the e-mail addresses, which consist of an aggregate of tokens. Whitespace and newlines are simply ignored. The `flex` input file for the configuration file lexer is shown in Listing 25.7.

Listing 25.7 Configuration Parser `flex` Input File (on the CD-ROM at `./source/ch25/config/config.fl`)

```
 1:    %{
 2:    #include <stdio.h>
 3:
 4:    #include "config.tab.h"
 5:    %}
 6:
 7:    %%
 8:    allow                   return ALLOW;
 9:    disallow                return DISALLOW;
10:    [a-zA-Z]+[a-zA-Z0-9_]*  yylval=strdup(yytext); return WORD;
11:    \{                      return OPEN_BRACE;
12:    \}                      return CLOSE_BRACE;
13:    \@                      return ATSYM;
14:    \.                      return PERIODSYM;
15:    \n                      /* Ignore end-of-line */
16:    [ \t]+                  /* Ignore whitespace  */
17:    %%
```

The first item to note is that you have specified that you need the parser generator header file to understand what tokens are communicated (line 4). Next, at line 10, you see a new action for a string pattern. This particular pattern recognizes strings that begin with a letter (uppercase or lowercase) and then follow with optional characters of letters, numbers, or the _ character. When you recognize one of these, its value (what was tokenized) is copied into a special variable called `yylval`. When the lexer recognizes this token, it's stored in `yytext`, which allows the parser to see the actual token (in addition to the token type that's returned, in this case `WORD`).

Now you have a lexer that tokenizes your file and also returns the special words that it finds (ignoring the reserved words, given their precedence in the table). Now you can explore the new parser (Listing 25.8).

Listing 25.8 Configuration Parser bison Input File (on the CD-ROM at ./source/ch25/config/config.y)

```
%{
#include <stdio.h>
#include <string.h>

void yyerror( const char *str )
{
        fprintf( stderr, "error: %s\n", str );
}


int main()
{
        FILE *infp;

        infp = fopen( "config.file", "r" );

        yyrestart( infp );

        yyparse();

        fclose( infp );

        return 0;
}


char address[10][80];
int  addrCount = 0;

%}

%token ALLOW OPEN_BRACE CLOSE_BRACE DISALLOW WORD ATSYM PERIODSYM

%%

configs:
        | configs config
        ;

config:
        allowed
```

```
42:             |
43:             disallowed
44:             ;
45:
46:     allowed: ALLOW OPEN_BRACE targets CLOSE_BRACE
47:             {
48:                     int i;
49:                     printf("Allow these addresses:\n");
50:                     for ( i = 0 ; i < addrCount ; i++ ) {
51:                             printf( "\t%s\n", address[i] );
52:                     }
53:                     addrCount = 0;
54:             }
55:             ;
56:
57:     disallowed: DISALLOW OPEN_BRACE targets CLOSE_BRACE
58:             {
59:                     int i;
60:                     printf("Disallow these addresses:\n");
61:                     for ( i = 0 ; i < addrCount ; i++ ) {
62:                             printf( "\t%s\n", address[i] );
63:                     }
64:                     addrCount = 0;
65:             }
66:             ;
67:
68:
69:     targets:
70:             |
71:             targets email_address
72:             ;
73:
74:
75:     email_address:
76:             WORD ATSYM WORD PERIODSYM WORD
77:             {
78:              if ( addrCount < 10 ) {
79:                        sprintf( address[addrCount++],
80:                                  "%s@%s.%s", $1, $3, $5 );
81:                        free($1);  free($3);  free($5);
82:                       }
83:             }
84:             ;
```

The C code section of Listing 25.8 (lines 1–30) illustrates some of the key changes that make this parser useful in the configuration domain. The `main` function, rather than simply calling `yyparse`, opens the desired configuration file and then calls the function `yyrestart` with the file pointer. This allows you to change the input from `stdin` to the file of your choice. You then call the `yyparse` function to perform the grammar parse. Upon return (the parse is either complete or an error was encountered), you close your input file pointer. You also declare a few resources (lines 27 and 28), which store the e-mail addresses and the number parsed so far. You can see how these are actually used in the `bison` rules section.

In the `bison` declaration section (lines 31–33) you define the tokens that are expected from the lexer. The result of this line (when the parser generator is created) is a header file containing symbolic constants for each of these tokens.

The rules section (lines 34–84) defines the grammar for the configuration parser. You include a base rule to allow one or more configuration sections (at lines 36–38) and then a rule to match either an `allow` section or a `disallow` section (lines 40–44). The `allowed` and `disallowed` rules are similar in pattern, except for what they represent (permitted or rejected e-mail addresses). Each begins with the respective reserved word (`allow` or `disallow`) and then an open brace (`{`) followed by zero or more e-mail addresses and terminated by a close brace (`}`). You see C code sections for each of the two rules, but you can return to these after you first dive into the `targets` rule.

In the `targets` rule, you specify that zero or more e-mail addresses can be recognized (lines 69–72). The `email_address` rule (lines 75–84) specifies a simple e-mail address recognizer. In this case, that is a word followed by an `@` symbol, followed by another two words, separated by a ".". Therefore, e-mail addresses such as:

```
mtj@mtjones.com
```

are recognized, but:

```
mtj@mail.mtjones.com
```

is not. For this demonstration, the simple e-mail address spec suffices.

The action portion of the `email_address` rule defines what you do when you recognize a valid e-mail address. In this case (as long as you haven't exhausted your storage resources), you populate an address entry with the aggregated e-mail address. Note here that the token list:

```
WORD ATSYM WORD PERIODSYM WORD
```

is accessible through a special set of references. The reference `$1` represents the first `WORD` in the list. Recall in the lexer (Listing 25.7, line 10) that you duplicated the current token using `strdup` to `yylval`. The `$1` here represents the `yylval` stored by the lexer. Reference `$3` represents the third `WORD` and `$5` the last `WORD`. Therefore, you use `sprintf` (to concatenate the strings together into your address array), resulting in a contiguous e-mail address from the individual `WORD` parts. After you have stored the address recognized, you free each of the string references. Recall that you used `strdup` to copy the current token. This has the effect of `malloc`ing memory for the new object, and therefore, you must free this resource.

After the `CLOSE_BRACE` is recognized for the section (line 46 or 57), you emit the addresses that are found. You know the type of addresses these are (`allowed` or `disallowed`), given the context of the rule. If you are in the `allowed` rule, then you know that these addresses are within an `allow` section. Similarly, within the `disallowed` rule, you emit the addresses as disallowed. After you emit each section, you zero the `addrCount` to permit additional sections to be recognized and stored.

Now take a look at an example of the parser. Listing 25.9 provides a sample file (properly formed).

LISTING 25.9 Sample Configuration File

```
 1:     allow {
 2:
 3:       mtj@mtjones.com
 4:       dan_5422@yahoo.com
 5:
 6:     }
 7:
 8:     disallow {
 9:
10:       you@yahoo.com
11:       them@excite.com
12:
13:     }
```

Calling this file `config.file` (as required by the parser), you can demonstrate its use as follows:

```
$ ./parser
Allow these addresses:
        mtj@mtjones.com
        dan_4522@yahoo.com
```

```
Disallow these addresses:
        you@yahoo.com
        them@excite.com
$
```

If any sections were not properly formed, you would not see any addresses (because the addresses aren't actually `printf`'d until the entire section is parsed).

While this was a simple example, it nevertheless illustrates some of the power of `flex` and `bison`. `flex` and `bison` can be used to build lexers and parsers for very complex languages (such as the C language) or very simple grammars as illustrated here.

The Big Picture

Figure 25.3 showed the phases involved in taking an input file, breaking it down into tokens (via the lexical analyzer), and then passing these tokens to the parser to generate a parse tree. Figure 25.4 now amends the previous figure to illustrate what was done here.

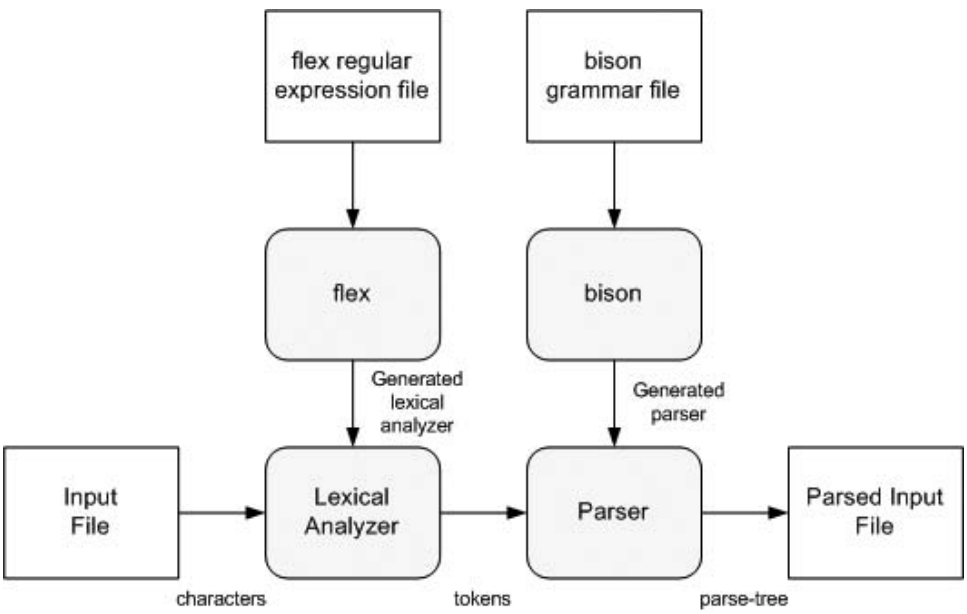

FIGURE 25.4 Parsing phases with `flex` and `bison` flows.

As this chapter has shown, the lexical analyzer is generated from the `flex` utility, given an input file representing the regular expressions that recognize the tokens of the grammar. The parser is generated from the `bison` utility, again specified by a grammar file. Each of these phases is built together in a single image, with connectivity between the two specified by the `flex` and `bison` tools and also by the developer.

The flow of the lexer and parser is partly provided internally but is visible in the grammar definitions (see Figure 25.5). Your main function (provided in the `bison` grammar file) calls `yyparse` to perform the parsing function. This in turn calls `yylex` to retrieve the tokens as they're extracted from the input stream. The `yylex` function returns the type of token found, with any other data needed by the parser returned in other variables (such as `yylval`).

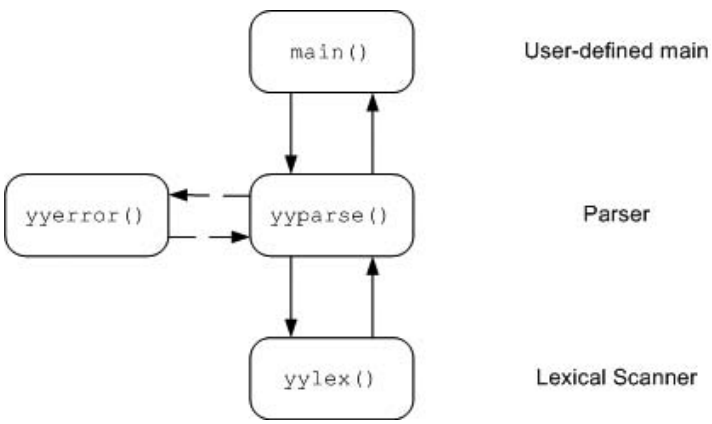

FIGURE 25.5 Grammar definitions in lexer and parser flow.

You have seen a few of the internal functions and variables provided in the scanner and parser. Table 25.1 provides a list of some of the others that you might encounter in your use of `flex` and `bison`.

Designing and specifying parsers (and lexers) can be a difficult task, but in the end, the act of specifying how the grammar works is necessary even if it's to be done by hand. After the specification is done, the generation of the parser with `bison` is trivial (compared to writing one by hand), and therefore `flex` and `bison` can be very useful tools in our development toolbox.

TABLE 25.1 Useful Scanner and Parser Functions and Variables

Name	Type	Description
yyparse	Function	Parser function (called by main)
yyerror	Function	Error function (can be provided by user)
yylex	Function	Scanner functions (returns tokens, used by yyparse)
yyterminate	Function	Terminates the parsing process
yylval	char*/union	Token value
yytext	char*	Pattern string used by the lexer
yydebug	int	Set to 1 to enable debug mode

Summary

The flex and bison tools can be two very important elements for software specification and generation of lexers and grammar parsers. The flex tool allows the specification and generation of lexical analyzers that provide the ability to tokenize the input. The bison tool allows the specification of a grammar, which when generated can take the tokens from the lexical analysis phase to recognize correctly formed input. This chapter demonstrated these tools in two scenarios, looking at two different parsing examples. The build process was also discussed, including linkage between the lexer and parser phases

This page intentionally left blank

26 Scripting with Ruby

In This Chapter

- Ruby Overview
- Ruby in Comparison to Other Languages
- Ruby Language Features

Introduction

It seems like a new language comes out every day. That's probably not too far from the truth, but the useful languages that take hold and build communities behind them are much less frequent. One of those languages that has grown in popularity in recent years is the Ruby language. Ruby is a portable object-oriented scripting language that was designed by Yukihiro Matsumoto. Ruby was partially inspired by the Eiffel and Ada languages and includes all of the features you'd expect in a modern language. This chapter explores the Ruby language and shows you why it's the next language that you should master.

An Introduction to Ruby

Like natural languages, computer languages allow programmers to express designs. Some languages make this easier than others. For example, machine languages that use lower level (register-level) instructions as the means for expressing design are naturally hard. Higher level languages hide the details of the given machine and make common concepts simpler to use (such as functions to build common behaviors).

But not all higher level languages are alike. Some languages make text processing easy (such as Icon), and others are ideal for building reliable systems (such as Ada). Ruby is a great general purpose language, but has some attributes that make it ideal for prototyping (testing ideas in the small), network programming, and also for teaching programming principles (including object-oriented programming).

Why Use Ruby?

In the end, programming languages are a matter of preference. Ruby is a great language, but practically anything that you can do in Ruby, you could also do in Python or Perl. Having said that, Ruby is a very special language. From a personal perspective, I find Ruby to be one of the most intuitive languages out there. The language just seems to make sense, and using new language features for the first time is natural and intuitive.

But preference plays a huge part in language use. The key question is whether or not you can be productive in a language. Given the intuitive nature of Ruby, it's one of the most productive languages available.

Comparing to Other Languages

Whereas Ruby was inspired by both Eiffel and Ada, it is nevertheless comparable to a variety of other languages. Take a look at some of the important attributes of Ruby in comparison to those in other popular languages. Some of Ruby's important characteristics are provided in Table 26.1.

TABLE 26.1 Important Attributes of the Ruby Language

Attribute	Other Languages
Multiparadigm	Ada, Objective Caml, Tcl, Lua
Dynamic	Smalltalk, Lisp, Lua, Eiffel, Python
Object oriented	Eiffel, Ada, C++, Smalltalk
Interpreted	Smalltalk, Tcl, BASIC, Scheme
Reflective	Smalltalk, Java, Lua, C#, Python

Multiparadigm languages are those that can support multiple language paradigms (imperative, object-oriented, functional, logic-based, as examples). Ruby supports each of the paradigms in an elegant way.

Dynamic languages perform functionality at runtime that is commonly performed at compile time. This can include adding new code at runtime and changing the type system. These features are commonly found in functional languages (such as closures and continuations).

Object-oriented languages are those that use an object model for development. In these languages, classes and methods (functions) are coupled with encapsulation, polymorphism, and inheritance. But Ruby (like Smalltalk and unlike most other object-oriented languages) is pure. Everything in Ruby is an object and can have properties or methods applied to it. You will explore the impact of this later in this chapter.

Languages that are compiled are translated from their high-level constructs to machine-level instructions. Running on the hardware directly means that these languages provide fast executables, but ones that are not portable. Interpreted languages are those in which the high-level language is interpreted directly without having gone to the machine level. This means that the execution of the scripts is slower, but it much more portable to run on other architectures without change.

Reflective programming refers to the ability to observe and modify the behavior of a program at runtime. From one perspective, if a language allows data to be interpreted as a program, then it supports reflection.

Ruby's most often discussed problem is its performance. When compared to other interpreted languages (such as Perl or Python), Ruby does not perform as well. But at the time of this writing, Ruby is moving towards version 1.9, which will include a new virtual machine with better performance.

Quick Ruby Examples

Now that you've explored Ruby at a high level, you can take a look at some of the high-level attributes of Ruby through some examples. These examples are provided to give you a quick taste of Ruby, and in the sections that follow take a deeper look at Ruby.

The classic example of language demonstration is the "Hello World" program. In Ruby, using `irb`, this is shown in Listing 26.1.

LISTING 26.1 The Ruby Hello World Program with `irb`

```
mtj@camus:~$ irb
irb(main):001:0> puts "Hello World."
Hello World.
=> nil
irb(main):002:0>
```

Creating a function (or method for Ruby) is shown in Listing 26.2. A method is defined with the `def` method. Note that after the method is defined, you can call it and see its result.

LISTING 26.2 Creating a Method (Function) with Ruby

```
irb(main):002:0> def my_sound
irb(main):003:1>   puts "bark"
irb(main):004:1> end
=> nil
irb(main):005:0> my_sound
bark
=> nil
irb(main):006:0>
```

You could create a class for animals very easily with Ruby using the `class` reserved word (see Listing 26.3). In this listing, you create a class called `Dog` and then create a new instance of this class and invoke its method.

LISTING 26.3 A Simple Class Example in Ruby

```
irb(main):001:0> class Dog
irb(main):002:1>   def my_sound
irb(main):003:2>     puts "bark"
irb(main):004:2>   end
irb(main):005:1> end
=> nil
irb(main):006:0> zoe = Dog.new
=> #<Dog:0xb7c5fd88>
irb(main):007:0> zoe.my_sound
bark
=> nil
irb(main):008:0>
```

Recall that Ruby is a pure object-oriented language, which means that everything is an object. Even numbers and primitive types are objects, for example:

```
irb(main):014:0> 5.+(1)
=> 6
```

This illustrates the application of a method on a number object (`5`). To that object, you apply the `+` method with an argument of `1`, resulting in `6`.

Finally, take a look at a multiparadigm aspect of Ruby. One of the most interesting language features of Ruby comes from functional languages like Lisp. The higher order function map is used to transform a list by applying a function to each element. The result of this operation is a new list. In Ruby, the `map` method is applied to the list object. Each element of the list (defined by variable `x`) is transformed as the square operation.

```
irb(main):017:0> [1, 2, 3, 4, 5].map { |x| x*x }
=> [1, 4, 9, 16, 25]
```

Note here that the `map` method takes a function as its argument and results in a new function that takes a list and applies each element of the list to it. Maps provide an interesting way to achieve iteration, but Ruby has others, which you will review.

Now that you've seen a short glimpse of what Ruby has to offer, the rest of the chapter takes a more systematic approach to reviewing the language and its offerings.

The Ruby language can be freely downloaded from the Ruby Web site at http://www.ruby-lang.org/en/. You can explore Ruby in two ways. The first is through traditional scripting (writing your Ruby in a script and then using the Ruby interpreter to execute). You can also use interactive Ruby (`irb`*) to interact with the interpreter on a line-by-line basis. This is a great way to experiment with the Ruby interpreter.*

Language Elements

Ruby is ranked in the top 10 languages worldwide, and some even call it artful. The next sections look at the language to see just why it's so popular and powerful.

Types and Variables

Ruby supports a number of different types, some of which might be foreign to you if you've not worked with a high-level (or domain-specific) language. Types in Ruby need not be explicitly defined, but instead can be inferred based upon value. This section looks at the basic types, and then others are introduced as you get into other features.

Ruby supports two numeric types, the `Fixnum` (or integer) and the `Float`. Strings are also supported, as are ranges. A range includes a lower and upper value and can be used in iterators (which you'll review later). Also shown is Ruby's ability to determine the type of a variable at runtime (see Listing 26.4). To determine

this, you use the `class` method (remember, everything is an object in Ruby, which implies that it's the instance of a class).

LISTING 26.4 Creating Variables and Then Determining Their Type

```
irb(main):001:0> x = 9
=> 9
irb(main):002:0> puts x.class
Fixnum
=> nil
irb(main):003:0> y = 1.414
=> 1.414
irb(main):004:0> puts y.class
Float
=> nil
irb(main):005:0> z = "text string"
=> "text string"
irb(main):006:0> puts z.class
String
=> nil
irb(main):007:0> w = 1..5
=> 1..5
irb(main):008:0> puts w.class
Range
=> nil
irb(main):009:0>
```

As you can guess, a language as powerful as Ruby is going to have more types than this. You'll see some of the other important types when you explore Ruby's object-oriented features later in the chapter.

Control

Ruby supports the standard range of conditionals, but also a few others that are useful to understand. The standard `if/then/else` constructs are available (see Listing 26.5) in addition to the `case` statement, with which you are no doubt familiar.

LISTING 26.5 Example of Ruby's `if/then/else` Conditional Structure

```
irb(main):001:0> legs = 4
=> 4
irb(main):002:0> if legs == 2 then
irb(main):003:1*   puts "biped"
irb(main):004:1> elsif legs == 4 then
```

```
irb(main):005:1*   puts "quadruped"
irb(main):006:1> elsif legs == 6 then
irb(main):007:1*   puts "hexaped"
irb(main):008:1> else
irb(main):009:1*   puts "undefined"
irb(main):010:1> end
quadruped
=> nil
irb(main):011:0>
```

The `case` statement so common in other languages is also available in Ruby. Just as in Ada (which also uses the `case`/`when` structure), in Ruby you can use ranges within `case` statements (see Listing 26.6). An `else` can also be used within the `case` statement, taking the place of the default block. The `then` statement is optional in this context.

LISTING 26.6 Illustrating the Ruby `case` Construct

```
irb(main):001:0> x = 9
=> 9
irb(main):002:0> case x
irb(main):003:1>   when 1..8 then puts "1..8"
irb(main):004:1>   when 9..12 then puts "9..12"
irb(main):005:1> end
9..12
=> nil
irb(main):006:0>
```

Finally, Ruby offers a couple of shorthand operations that are useful in certain cases. The `if` modifier and `unless` keyword are syntactic sugar, but are useful nonetheless.

The `if` modifier simply reverses the order of the `if` condition. For example, the following two examples are syntactically identical:

```
if legs == 2 then puts "biped" end
puts "biped" if legs == 2
```

The `unless` expression is another example of syntactic sugar. Using `unless` simply means it maps to the `not` condition, as shown in the following:

```
legs = false
if !legs then puts "no legs" end
unless legs then puts "no legs" end
```

"Syntactic sugar" is a term coined by Peter Landin, and it describes additions to a language that don't affect its functionality but make the language "sweeter" to use.

Iteration

Ruby offers a number of methods for iteration, many of which are recognizable from other languages, but Ruby also offers some novel additions. You can begin with the `while` statement that simply repeats a block of code while an expression is true. The `while` modifier is a special case of `while`, which essentially reverses expression ordering (as shown in Listing 26.7).

LISTING 26.7 `while` and `while` Modifier in Ruby

```
irb(main):016:0> while x < 10
irb(main):017:1>   x+=1
irb(main):018:1> end

irb(main):019:0> x+=1 while x < 10
```

Another interesting looping technique is called `until`. This repeats execution of a body of code until an expression returns true. Listing 26.8 shows two variants of `until` (including the `until` modifier).

LISTING 26.8 `until` and `until` Modifier in Ruby.

```
irb(main):005:0> until x == 10
irb(main):006:1>   x+=1
irb(main):007:1> end

irb(main):008:0> x+=1 until x == 10
```

Another looping construct that is familiar in other languages is the `for` loop. What's different primarily is the use of the range, but some other interesting models are available as well. Listing 26.9 first shows the standard `for` loop with a range. But the second example illustrates Ruby's use of an array with iteration. Here the `for` loop iterates through an array of primes to 13.

LISTING 26.9 `for` Loop and Its Variants.

```
irb(main):010:0> for i in 1..10
irb(main):011:1>   puts i
irb(main):012:1> end

irb(main):001:0> for i in [2, 3, 5, 7, 11, 13]
irb(main):002:1>   puts i
irb(main):003:1> end
```

But as it turns out, the `for` loop is just syntactic sugar for Ruby's real iterator. For example (see Listing 26.10), you can code your array iterator as shown below. This code snippet uses `each` to iterate through the array, with the iterating variable being `x`. The second example shows that this is much different from standard iterators in other languages. Because the iteration is over the elements of an array, the array can be made up of more than just `Fixnums`. The final example in Listing 26.10 is the `each_with_index` method. This method (of class `Array`) returns not only the element in the array but also its position.

LISTING 26.10 Iterating Over an Array

```
irb(main):005:0* [1, 3, 5, 7, 11, 13].each do |x|
irb(main):006:1*   puts x
irb(main):007:1> end

irb(main):021:0> names = ['Tim', 'Jill', 'Megan', 'Elise', 'Marc']
=> ["Tim", "Jill", "Megan", "Elise", "Marc"]
irb(main):022:0> names.each do |x|
irb(main):023:1*   puts x
irb(main):024:1> end
Tim
Jill
Megan
Elise
Marc
=> ["Tim", "Jill", "Megan", "Elise", "Marc"]
irb(main):025:0>
```

```
irb(main):031:0> names.each_with_index do |x, index|
irb(main):032:1*   puts "#{x} at pos #{index}"
irb(main):033:1> end
Tim at pos 0
Jill at pos 1
Megan at pos 2
Elise at pos 3
Marc at pos 4
=> ["Tim", "Jill", "Megan", "Elise", "Marc"]
irb(main):034:0>
```

String Handling in Ruby

Ruby is very interesting when it comes to string handling. The `String` object has an interesting set of methods that are available, including the arithmetic operators. For example, as shown in Listing 26.11, you can apply the `+` and `*` operators to Ruby strings, with the intuitive result.

LISTING 26.11 Ruby Arithmetic Operations on Strings

```
irb(main):001:0> str = "test"
=> "test"
irb(main):002:0> str = str + "string"
=> "teststring"
irb(main):003:0> str = str * 2
=> "teststringteststring"
irb(main):004:0>
```

Ruby also provides the standard set of operators that you would expect for natural string operations. For example, you can retrieve the `length` of a string and extract substrings (see Listing 26.12). The `delete` method can be used to remove characters from a string. Note the use of `delete!` here, which modifies the string in place. In all other cases, a new string is returned.

The final example in Listing 26.12 illustrates the ability to chain methods together on a given object. In this case, the `str` object has the `reverse` method applied, followed by the `delete` method (removing the `g` character).

LISTING 26.12 Ruby Arithmetic Operations on Strings

```
irb(main):001:0> str = "test"
=> "test"
irb(main):002:0> str = str + "string"
=> "teststring"
```

```
irb(main):003:0> str = str * 2
=> "teststringteststring"
irb(main):004:0> str.length
=> 20
irb(main):005:0> str[0,7]
=> "teststr"
irb(main):006:0> str.delete!("test")
=> "ringring"
irb(main):007:0> str.reverse
=> "gnirgnir"
irb(main):008:0> puts str.reverse.delete("g")
rinrin
=> nil
```

Associative Arrays

Ruby includes support for associative arrays. An associative array, which is also called a hash, is essentially a lookup table. You have a key, and with each key you have an associated value. The operation of lookup is provided in the language, which means simpler code. The associative array can be viewed as an array of associations. Listing 26.13 shows the creation of an associative array and its use. The `=>` symbol associates the key (such as `Tim`) with a value (such as `42`).

LISTING 26.13 Creating and Using an Associative Array

```
irb(main):001:0> a_array = { 'Tim' => 42, 'Jill' => 40, 'Megan' => 15 }
=> {"Jill"=>40, "Tim"=>42, "Megan"=>15}
irb(main):002:0> a_array['Tim']
=> 42
irb(main):003:0> a_array['Megan']
=> 15
irb(main):004:0> a_array['Elise']
=> nil
irb(main):005:0>
```

Classes and Methods

Ruby is an object-oriented language, so as you would expect, classes are part of the language. A class is a construct that groups related methods and variables together. Variables in this case are instance variables, which you'll review shortly.

Recall that everything in Ruby is an object. You can see this through an interactive Ruby session. With an empty class named `Dog`*, you invoke the* `class` *method to find that this is an object of type* `Module` *(which is itself an instance of an* `Object`*).*

```
irb(main):001:0> class Dog
irb(main):002:1> end
=> nil
irb(main):003:0> Dog.class
=> Module
irb(main):004:0> Dog.superclass
=> Object
```

Note in this example that `Dog` *has no methods defined, yet the* `superclass` *method was invoked. This is because* `superclass` *is a method of* `Object`*, of which* `Dog` *is a descendent.*

Take a look at an example of building a class in Ruby, and also extending a class through inheritance. Listing 26.14 provides an example of each through a class called `Dog`. Class `Dog` has two methods: `sound` and `name`. These methods simply emit a string when invoked. Next, you define a new class called `LittleDog`, which is a subclass of `Dog`. Note that in Ruby, you can have only one level of inheritance. In `LittleDog`, the `sound` method is redefined (to fit a smaller dog).

The remainder of Listing 26.14 shows creating an instance of each class. In the first case, an object `rover` is created using the new method (which is part of the `Class` class). The methods of instance `rover` are called, which results in `"Spot"` and `"Bark"`. A `LittleDog` instance is then created named `fifi`. The methods for `fifi` are invoked, resulting in `"Spot"` and `"Yip"`.

LISTING 26.14 Demonstrating Ruby Classes and Inheritance

```
class Dog

  def sound
    puts "Bark"
  end

  def name
    puts "Spot"
  end

end
```

```
class LittleDog < Dog

  def sound
    puts "Yip"
  end

end


rover = Dog.new
rover.name
rover.sound

fifi = LittleDog.new
fifi.name
fifi.sound
```

As you would expect, invoking the `class` method on `rover` and `fifi` results in `Dog` and `LittleDog`, respectively. Class instance variables can also be included inside the classes and initialized through a special function called `initialize`. Listing 26.10 presents a new class called `NewDog`. This class has the previous two methods for emitting the name and sound, but also a new method for initializing these instance variables. You can take a deeper look at `initialize` to see how this is done.

Method `initialize` is a special method that is invoked when the `new` method is called (the object is created). This method can take zero or more arguments. In this case, you specify two arguments that can be initialized for an instance of the class (`name` and `sound`). Within `initialize`, you check the length of the strings that are passed in, and if zero is the result, then you initialize with a default value. Note the use of the `@` symbol in the variable name. This defines that the variable is a class instance variable (associated with this instance of the class). A `@@` would represent a class variable, which would be common in all instances of the class.

The remainder of the class is simple. The `name` and `sound` methods emit the class instance variables that were set in `initialize`. At the end of Listing 26.15, the class is instantiated twice. In the first case, the `name` and `sound` class instance variables are initialized, but in the second example, only the `sound` class instance variable is initialized. In this case, the output is `"Spot"` and `"Woof"`.

Note that for greater error checking, you can use the `class` *method to ensure that the type passed in is a* `String`*. For example:*

```
if name.class != String) then error...
```

LISTING 26.15 Instance Variables and Initialization

```
class NewDog

  def initialize( name, sound )

    if name.length == 0
      @name = "Spot"
    else
      @name = name
    end

    if sound.length == 0
      @sound = "bark"
    else
      @sound = sound
    end

  end

  def name
    puts @name
  end

  def sound
    puts @sound
  end

end

zoe = NewDog.new( "Zoe", "Yalp" )
zoe.name
zoe.sound

noname = NewDog.new( "", "Woof" )
noname.name
noname.sound
```

One final example illustrates some of the freedom available in Ruby for object creation. You can construct an object and then add methods to it as needed. This is done by creating an object as an instance of `Object`, as illustrated in Listing 26.16. With the object instance (`dyn`), a method is added to it by simply defining a new method with the class instance prefix (`dyn.identify`).

LISTING 26.16 Dynamically Adding Methods to Objects

```
irb(main):001:0> dyn = Object.new
=> #<Object:0xb7c813ac>
irb(main):002:0> def dyn.identify
irb(main):003:1>   puts "identify method"
irb(main):004:1> end
=> nil
irb(main):005:0> dyn.identify
identify method
=> nil
irb(main):006:0> dyn.class
=> Object
irb(main):007:0>
```

Now that you have a basic understanding of Ruby language features, the next section looks at some of the more advanced features and then some examples.

Advanced Features

Ruby supports a number of advanced features, some of which might not be familiar. As Ruby has a number of influences, you'll find object-oriented features, imperative language features, and functional language features. This section shows a few of the more important elements available within Ruby.

Dynamic Code

Ruby provides a class called `Proc` that allows blocks of code to be created that can be dynamically bound to variables. This allows the same method to be bound to variables independently resulting in different contexts of the same method. An example can best illustrate what this means. Listing 26.17 shows a use of the `Proc` class within a method. The `gen_mul` method creates a new method bound to the argument passed in (`factor`). The `n` variable is the argument of the resulting method, as shown in the listing. The `call` method invokes the dynamic method.

At any time you can determine the binding to the parameter of the dynamic method using `eval`. The `eval` method comes from the `Kernel` module.

LISTING 26.17 Dynamically Adding Methods to Objects

```
irb(main):003:0* def gen_mul(factor)
irb(main):004:1>   return Proc.new { |n| n*factor }
irb(main):005:1> end
```

```
=> nil
irb(main):006:0> mulby3 = gen_mul(3)
=> #<Proc:0xb7d485b4@(irb):4>
irb(main):007:0> mulby10 = gen_mul(10)
=> #<Proc:0xb7d485b4@(irb):4>
irb(main):008:0> mulby3.call(7)
=> 21
irb(main):009:0> mulby10.call(8)
=> 80
irb(main):010:0> eval("factor", mulby3)
=> 3
irb(main):011:0> eval("factor", mulby10)
=> 10
irb(main):012:0>
```

This aspect of Ruby is similar to Ada's generic capability. The Ada generic allows for parameterized functions, not only of value but also of type.

Exception Handling

Exceptions are how languages handle dynamic or user-definable errors. Simply put, the language supports setting up exception handlers to catch runtime errors when they occur. For example, say you want to build a method to compute the square of a number. This can be done simply, as shown in Listing 26.18.

LISTING 26.18 A Simple `square` Method

```
def square( x )
  return x*x
end
```

However, then what happens when you pass something invalid? Here's an `irb` session showing just that (see Listing 26.19).

LISTING 26.19 Creating an Error by Passing an Incorrect Type

```
irb(main):004:0> square( "hello" )
TypeError: can't convert String into Integer
        from (irb):2:in `*'
        from (irb):2:in `square'
        from (irb):4
        from :0
irb(main):005:0>
```

As shown, a type error occurs because the `'*'` method can't be applied to a `String` type. You can detect this situation and report it to the caller with the `raise` method. So you can detect and return an exception, as shown in Listing 26.20. The `raise` method, in this case, returns a runtime error string if the type of the passed object is not `Fixnum`. Otherwise, the square is returned. Note the error returned after trying to pass in a `String`.

LISTING 26.20 Raising an Exception

```
irb(main):001:0> def square( x )
irb(main):002:1>   raise "Not a number (#{x.class})" if x.class !=
Fixnum
irb(main):003:1>   return x*x
irb(main):004:1> end
=> nil
irb(main):005:0> square(5)
=> 25
irb(main):006:0> square("string")
RuntimeError: Not a number (String)
        from (irb):2:in `square'
        from (irb):6
        from :0
irb(main):007:0>
```

What's really needed in this case is a way not only to throw an exception, but also to catch it in the caller. This is done, not surprisingly, with a keyword called `rescue`. You can see this in Listing 26.21. In the `square` method, you raise an `Exception` called `ArgumentError`, which is an object. The `test` method is where the exception catching occurs. You define a block (that starts with a `begin`) that includes a `rescue` clause. If an exception occurs within the block, then the `rescue` clause is invoked. Note here that the `rescue` clause is named with the particular exception to be caught. It could be "un-named" to catch all exceptions.

LISTING 26.21 Raising an Exception and Catching Within a Rescue Block

```
def square( a )

  raise ArgumentError if a.class != Fixnum

  return a*a

end
```

```
def test

  begin

    puts square( 5 )

    puts square( "hello" )

  rescue ArgumentError

    puts "Recover from this error!!!"

  end

  puts "done"

end
```

When the `test` method is invoked, it results in the following (see Listing 26.22). The first call to `square` returns normally, but the second call throws the `ArgumentError` exception, which is caught by the `rescue` clause.

LISTING 26.22 Executing the Exception `test` Method

```
irb(main):027:0> test
25
Recover from this error!!!
done
=> nil
irb(main):028:0>
```

By catching the exception, you can recover from the error, instead of the method simply causing the script to exit. This is an important distinction and allows the development of more reliable applications.

Introspection

Introspection, or reflection, refers to a language's ability to inspect itself at runtime. One aspect of introspection is the ability to enumerate living objects at runtime. Ruby provides an interesting module called `ObjectSpace` that provides this ability. You can refine this by iterating only through objects of a specific class or module. This is demonstrated in Listing 26.23.

LISTING 26.23 Enumerating All Numeric Objects

```
irb(main):040:0> ObjectSpace.each_object(Numeric) {|x| puts x}
2.71828182845905
3.14159265358979
2.22044604925031e-16
1.79769313486232e+308
2.2250738585072e-308
=> 5
irb(main):041:0>
```

The `ObjectSpace` module can also be used to add `finalizers` to other objects. A `finalizer` is a `proc` that is invoked when a particular object is destroyed. This operation is shown in Listing 26.24. A sample class is created (`Dog`), and then an instance of that class (`zoe`). The `ObjectSpace` module is included, so that you don't have to specify it as a prefix to its methods. The `finalizer` is created with a call to `define_finalizer`. This includes your object and a simple `proc` that is called before the object is garbage collected. You force the object to be removed, and a call to `garbage_collect` (part of the `ObjectSpace` module) finally removes the object causing a call to the `finalizer`.

LISTING 26.24 Creating a `finalizer` for an Object

```
irb(main):001:0> class Dog
irb(main):002:1> end
=> nil
irb(main):003:0> include ObjectSpace
=> Object
irb(main):004:0> zoe = Dog.new
=> #<Dog:0xb7d317c4>
irb(main):005:0> define_finalizer(zoe, proc {|id| puts "bye #{id}"})
=> [0, #<Proc:0xb7d1ca18@(irb):6>]
irb(main):006:0> zoe = nil
=> nil
irb(main):007:0> garbage_collect
bye -605436738
=> nil
```

Other Features

Entire books could be written on the useful features of Ruby, but unfortunately only a chapter is provided here. Ruby also provides an operating system–independent threads implementation. This makes the threads behavior uniform across all

platforms, which is a great advantage. Ruby supports many other features outside of the language, such as the standard interpreter, an interactive interpreter, and a debugger and profiler. Ruby compilers also exist if performance is more important than portability.

Ruby as an Embedded Language

One of the greatest features of this class of language is that many of them (including Ruby) can be embedded in another language. This is a popular technique in game engines where the scripting language allows user extension of the game (for example, non-player-character behaviors) while the graphics engine runs natively for speed.

In addition to embedding Ruby into another language, you can extend Ruby with another language such as C. This allows you to build methods in C that execute natively for performance, but still maintain the flexibility provided by Ruby.

Summary

While this has been a whirlwind tour of Ruby, what you've explored here should give you a sense of the power and intuitiveness of the Ruby language. Ruby is an infant language when you consider other languages from which it was inspired. Smalltalk, for example, began development in 1969, whereas Ruby first appeared in 1996 (almost 27 years later). But Ruby has learned from its predecessors and incorporates the most important features of many languages. From its simple object-oriented structure to its advanced features such as introspection and exception handling, Ruby is a complete language and worth the time and effort to learn.

Resources

Ruby Programming Language Web site at http://www.ruby-lang.org/en/.

Ruby Language Wikipedia page at http://en.wikipedia.org/wiki/Ruby_(programming_language).

Ruby-Doc, Help and Documentation for the Ruby Language at http://www.ruby-doc.org/.

27 Scripting with Python

In This Chapter

- Python Introduction
- Comparable Languages to Python
- Python Language Review

Introduction

The previous chapter discussed the object-oriented language Ruby. Whereas Ruby is gaining in popularity, Python is here today with an army of devoted users. Python is another object-oriented and multiparadigm scripting language that is both useful and very popular. One of the key ideas behind Python is that it be readable and accessible to everyone (experienced programmers and beginners alike). Today you can find Python everywhere and on every major OS platform. This chapter explores Python and gets into what makes it tick.

An Introduction to Python

Python is one of the older scripting languages because even though it was released in its first version in 1991, it was conceived and developing during the 1980s. Python was created by Guido van Rossum as a successor to the ABC programming language at the National Research Institute for Mathematics and Computer Science (or CWI, in Dutch) in the Netherlands. The ABC language was also developed at CWI as a general-purpose language whose goal was to educate.

Like Ruby and the other object-oriented scripting languages, Python has evolved over the years. Its first release was in 1991, but the "1.0" release didn't come until 1994. Some of the major elements introduced in the 1.0 release were numerous Lisp features such as `lambda`, `map`, and `reduce`. Python's evolution continued with the acquisition of new features from Modula-2, Haskell, Icon, Java, and Scheme. In the recent past Python unified its types and classes into a single hierarchy (as Ruby had done from the beginning).

Why Use Python?

You have many reasons to use Python, but this chapter focuses on a few of the more important aspects of the language. First, Python is used in many places, indicating that it's a solid language with a quality run-time environment. Python powers the Zope application server (which is a persistent object database) that can be easily managed through a standard web interface. You'll also find Python behind the YouTube video-sharing website and even in game systems (such as *Civilization IV*). Python is a great language that's highly readable with a reliable environment.

Python also includes a massive standard library that can be easily augmented with user-defined modules. The standard library includes standard types of operations that you would expect, but also provides high-level interfaces for things like application layer Internet protocols (HTTP, SMTP, etc.). This makes it easy to build feature-rich applications with a minimal amount of user code.

But one of the most interesting aspects of Python is its development process. Python includes what's called a PEP, or Python Enhancement Proposal, which is used to propose new features for Python. This is similar to the Request for Comments (RFC) process that defined the evolution of the Internet protocols. The PEPs document everything from new language feature requests, schedules for releases, application programming interfaces (APIs), and the PEP process itself. This openness to the user and developer community makes Python a powerful contender in the future.

One of the biggest complaints of Python is that it's slower than other languages of its type. This is true in certain cases, but Python serves another need first. Source changes to Python for the purposes of optimization are avoided if it means that Python itself is less readable. If performance is key and you desire to continue with Python, then you can write time-critical portions of source in a language that compiles to the machine's native instruction set. These native routines can then be joined with Python through its foreign function interface.

Comparing to Other Languages

Because it's always easier to understand something by comparing it to something else you already know, this section explores some of the important attributes of the

Python language and their application in other programming languages. Table 27.1 provides some of the major attributes of the Python language.

TABLE 27.1 Important Attributes of the Python Language

Attribute	Other Languages
Multiparadigm	Ada, Ruby, Tcl, Lua
Dynamic	Eiffel, Lua, Lisp, Smalltalk
Object oriented	Ada, C++, Eiffel, Ruby, Smalltalk
Interpreted	BASIC, Scheme, Smalltalk, Tcl
Reflective	C#, Java, Lua, Ruby, Smalltalk
Strongly typed	C, Haskell, Java, Pascal

Python is multiparadigm because it supports more than one programming approach. For example, you can write object-oriented programs with Python and also imperative programs. You can also write in a functional paradigm (such as Lisp) and also logic (for example, Prolog). All of this simply means that Python employs the best ideas of the best programming languages and makes them available for you to express your ideas.

A dynamic language permits activities at run time that in traditional languages are permitted only at compile time. This includes adding new elements or types to the language. Python permits such functionality (as is common in functional languages).

Python can be used in an object-oriented fashion (remember, Python is multiparadigm). This means that applications are constructed in terms of objects with encapsulation, inheritance, and polymorphism. Applications are built from collections of objects that cooperate to solve its given task.

Python is an interpreted language, but can also be compiled. This means that Python can benefit from multiplatform interpretation or can be compiled to run directly on the bare hardware for better performance. Python is also implemented in a number of variants. Standard Python (called CPython) is Python implemented in the C language. You also have Jython, which is Python written in Java. You can also find IronPython, which is Python for the .NET environment, and even a self-hosted Python implementation (Python written in Python, called PyPy).

Reflective programming simply means that the distinction of capabilities between compile time and run time is blurred (activities usually restricted to compile time can be performed at run time). A classic example of this is Lisp where program and data can be intermixed (program treated as data, and data treated as program).

Strong typing is unfortunately an overused term, and its definition can vary depending upon how it's used. Strong typing in this context (language design) simply means that a type system exists and rules exist for the way types are used and intermixed. For example, it should not be possible to take the square root of a string (though simple arithmetic operations are applied to strings successfully, such as string addition for concatenation). Certain languages have no typing (such as Forth, which requires the programmer to explicitly know how to treat a memory object). Other languages require all variables to have a type with casting to convert between types. Other languages do not support type conversion without working around the language (such as Ada and unchecked conversion).

Python does not enforce static typing, but is instead dynamically typed. This means that the particular type of a variable might not be known until it is used in execution (through the context of its use).

Quick Python Examples

One of the most interesting aspects of Python programming is that code is very readable and intuitive. Python can be used for script execution and also as an interactive interpreter. This allows you to experiment with the language and see immediate results from your input.

You can start with the simplest example, the classic "hello world" program (see Listing 27.1). Python is started without a script, so it begins in interactive mode. Note here that the lines that begin with >>> and ... are Python prompts and indicate user input into Python. Everything else is Python output. The >>> is the standard Python prompt and ... is the continuation prompt. In this example, you declare your function using the `def` keyword (short for define). This is followed by the function name and an optional list of arguments (in this case, none). All statements that follow the : are executed as the function. Only one statement follows, which uses the `print` statement to emit the message. Invoking the new function is then done by simply referencing it.

LISTING 27.1 The Hello World Program with Python

```
mtj@camus:~$ python
Python 2.5.1 (r251:54863, Nov  4 2007, 12:38:30)
[GCC 4.0.3 (Ubuntu 4.0.3-1ubuntu5)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> def helloworld():
...     print "Hello World"
...
```

```
>>> helloworld()
Hello World
>>>
```

One thing that's probably obvious in this example is the lack of an "end" statement. Python uses indentation to determine what is contained in a block. This can be a little confusing, but because most programs are written with indentation for readability, Python uses this to determine which code belongs to which block.

In Listing 27.1, you saw how to create a simple function, but now take a look at an example with arguments (see Listing 27.2). Here you define a method called `circle_area` that computes the area of a circle given its radius. You first import the `math` module (which binds the module into the local namespace, making its contents available). You then define your function with a single input argument that does not have an associated type. Its type is derived later when it's used in a floating-point operation. The `area` variable is calculated and then returned.

LISTING 27.2 Creating a Method (Function) with Python

```
>>> import math
>>> def circle_area(r):
...     area = math.pi * r * r
...     return area
...
>>> circle_area(5)
78.539816339744831
>>>
```

Now take a look at a simple example of Python's form of object-oriented programming. A class is defined using the `class` keyword, which is followed by the name of the class—for example, `Dog`. Within the class you can define the class objects. For example, in Listing 27.3, you create a single method called `my_sound`. The method includes a single statement that prints the sound that this animal makes. After the class is defined you instantiate a new instance of the class. Note here that you use the class object like a function (as a class instantiation can include arguments). After the new instance is created, you invoke the `my_sound` method through the instance to achieve the obvious result.

LISTING 27.3 A Simple Class Example in Python

```
>>> class Dog:
...     def my_sound(self):
...             print "bark"
```

```
...
>>> zoe = Dog()
>>> zoe.my_sound()
bark
>>>
```

As a final example, it's time to take a quick look at functional programming in Python. One of the most basic concepts in functional programming is the `map` operation. The `map` operation works by applying a function to a list and returning a new list (recall that functional languages like Lisp and Scheme are very list-oriented). The `map` operation can be performed outside of the functional style, but use of the `map` function results in very clean and readable code.

The `map` function is demonstrated in Listing 27.4. The `map` function takes two arguments, a function and a list. In this example, you create a new function that simply returns the square of its argument to represent your function. Then you create a simple list. Finally, the `map` function is used to iterate the list using the provided function. The result is a new list.

LISTING 27.4 Using the `map` Function to Iterate a List

```
>>> def square(x):
...     return x*x
...
>>> x = [1, 2, 3, 4, 5]
>>> map(square, x)
[1, 4, 9, 16, 25]
>>>
```

Note here how the `map` function results in very clean code that is simple and readable. You could implement this using an iterator as well, as shown in Listing 27.5. In this example, you create an empty list and then iterate the original list using the `for` keyword. For each item in `x`, you `append` the square of that item to the new list `n`. The result is the same, albeit a bit involved than before.

LISTING 27.5 Emulating the `map` Function Through Iteration

```
>>> n = []
>>> for y in x:
...     n.append(square(y))
...
>>> print n
[1, 4, 9, 16, 25]
>>>
```

Later in this chapter, you'll look at some of the other functional capabilities of the Python language, including `lambda` functions, `reduce` and `filter`.

You can download the latest version of Python from the Python website at http://www.python.org. This book uses version 2.5.1 of the Python interpreter. The Python website also includes a large number of tutorials and extensive documentation as well as audio/visual resources to help bring you up to speed.

Language Elements

Python is a hugely popular language, and for good reason. This section explores the Python language to see why it's so readable, easy to use, and popular.

Types and Variables

Python includes a number of built-in types, some of which you're probably familiar with and some you might not be. Python includes four basic numeric types: `int`, `long`, `float`, and `complex`. These are demonstrated in Listing 27.6, including the use of the `type` function to determine the type of an object at run time.

LISTING 27.6 Basic Numeric Types in Python

```
>>> a = 1
>>> lo = 98237498723498234234987234987234
>>> f = 3.14159
>>> c = complex(1, -1)
>>> type(a)
<type 'int'>
>>> type(lo)
<type 'long'>
>>> type(f)
<type 'float'>
>>> type(c)
<type 'complex'>
>>>
```

In addition to the basic types, Python also supports a number of aggregate types such as strings, lists, and tuples (see Listing 27.7). These types are also called container types. The string represents a list of characters and can be delimited with either the single quote (`'`) or double-quote (`"`) character. A string is an immutable object that can't be changed after it is created. A list on the other hand is a mutable

object. This means that you can add or remove items from the list without having to create a new list.

A tuple is another immutable type that contains a number of objects. You can think of a tuple as a kind of structure that contains a mixture of types. As shown in Listing 27.7, you can address a tuple and its individual elements.

LISTING 27.7 The Python Sequence Types

```
>>> s = 'my string'
>>> l = [1, 2, [3.1, 3.2], 4, 5, [6.1, 6.2, 6.3]]
>>> tp = ("Tim", 42, "Jill", 40)
>>> type(s)
<type 'str'>
>>> type(l)
<type 'list'>
>>> type(tp)
<type 'tuple'>
>>> type(tp[0])
<type 'str'>
>>> type(tp[1])
<type 'int'>
>>>
```

Finally, types are not specific to variables; all objects in Python have a type. For example, if you declare a function, that function has the type `'function'`. A class is of type `'classobj'` and a method within a class is a type `'instancemethod'`. Finally, an instance of a class is of type `'instance'`. These are demonstrated in Listing 27.8.

LISTING 27.8 Types of other Objects in Python.

```
>>> def square(x):
...     return x*x
...
>>> type(square)
<type 'function'>
>>> class dog:
...     def func():
...             return 0
...
>>>
>>> type(dog)
<type 'classobj'>
>>> type(dog.func)
```

```
<type 'instancemethod'>
>>> zoe = dog()
>>> type(zoe)
<type 'instance'>
>>>
```

Every object in Python has an associated type and does not change after the object has been created. An object's value might change (as indicated by its type), but never the type of object itself. The type of the object determines which methods can be applied to it. The coming sections explore some of these methods as they apply to particular objects.

Control

Python supports the standard `if` conditional. This operates the same way you'd expect, but has a variation between the `else` and `else-if` variant. Instead of `'else if'`, Python instead uses `elif` (see Listing 27.9).

LISTING 27.9 Using the `if` Conditional in Python

```
>>> mode = "test"
>>> if mode == "test":
...     print "in the test mode"
... else:
...     print "not in the test mode"
...
in the test mode
>>>
```

The use of `elif` is shown in Listing 27.10. This is simply the `'else if'` variant and can be chained for any number of tests. In this case, you have two tests, and then an `else` (the default case).

LISTING 27.10 Use of `else-if` to Chain Conditionals Together

```
>>> mode = "run"
>>> if mode == "test":
...     print "test mode"
... elif mode == "run":
...     print "run mode"
... else:
...     print "unknown"
...
```

```
run mode
>>>
```

Iteration

Python provides a number of constructs for iteration, including `while` and `for`, and a few others that exploit unique Python types. The `while` loop has the standard structure of a condition followed by a block of statements (see Listing 27.11). Two options are shown here, the first using a condition and the other demonstrating an infinite loop. Within this example, the `break` statement is used to control termination of the loop.

LISTING 27.11 Python `while` Loop Examples

```
>>> i = 0
>>> while i < 10:
...     i = i + 1
>>>
>>> while 1:
...     i = i - 1
...     if i == 0: break
```

The `for` loop uses the standard convention, iterating over a range. But Python provides some interesting variants of this using the range type and list. Listing 27.12 demonstrates a few useful forms of the `for` loop. Note that these two examples are identical, each iterates through the range 0 through 9. The first example uses a range, which for a range (x,y) means the values from x to y – 1.

LISTING 27.12 Python `for` Loop Examples

```
>>> for i in range(0,10):
...     print i
...
** prints 0 – 9
>>> for i in (0, 1, 2, 3, 4, 5, 6, 7, 8, 9):
...     print i
...
** prints 0 - 9
```

The next variation uses a sequence. A sequence can be a string, list, or tuple, and Python correctly iterates through it. Listing 27.13 iterates through a tuple and emits the types of each element as they are encountered using the type function.

LISTING 27.13 Python `for` Loop Example with a Tuple

```
>>> tp = ("Megan", 15, "Elise", 11, "Marc", 8)
>>> for v in tp:
...     print type(v)
...
<type 'str'>
<type 'int'>
<type 'str'>
<type 'int'>
<type 'str'>
<type 'int'>
>>>
```

One final looping construct that should be considered is the `map` function (Listing 27.14). The result of `map` is a list that contains the elements emitted within the `for` loop of Listing 27.13. Listing 27.14 contains two examples that are identical. One uses the `map` function and the other uses a `for` loop to construct the same list.

LISTING 27.14 Comparing the `map` Function with a Comparable `for` Loop

```
>>> m = map(type, tp)
>>> print m
[<type 'str'>, <type 'int'>, <type 'str'>, <type 'int'>, <type 'str'>,
<type 'int'>]
>>>
>>> l = []
>>> for v in tp:
...     l.append(type(v))
...
>>> print l
[<type 'str'>, <type 'int'>, <type 'str'>, <type 'int'>, <type 'str'>,
<type 'int'>]
>>>
```

String Handling in Python

Like most high-level scripting languages, Python has a very large library of operations that can be performed on strings. These built-in operations tend to be very useful and are one of the main reasons that Python is so important in text-processing applications. Recall that strings are immutable objects, which means that after they are created they cannot be modified. Instead, if a string is to be modified, a new string is returned instead. This results in greater higher memory utilization and eventual garbage collection, so it's something to consider.

Garbage collection is the periodic collection of unused objects and returning of their memory to the heap. Garbage collection is a good thing, but when a large number of objects need to be reaped, it can result in choppy execution. For example, if you are reaping a small number of objects, only a small amount of time is necessary. But if you are reaping a large number of objects, then it can take significant time. You can control this process through the garbage collector interface, which is reviewed later in this chapter.

One of the most basic string operations is string length (see Listing 27.15). The `len` function takes the string as its argument and returns the length of the string (the number of characters within the string). The `count` method can be used to count the number and letter sequences within a string. Note the difference between these two usages. The `len` function operates on the string whereas the `count` method is provided through the string object. In the latter case, you specify the object and method to invoke that object. Another useful approach is used to find a substring within another string. The result is the starting character position of that substring. You can also replace a substring with another substring using the `replace` method. This method takes the search string and replacement string and returns a new transformed string.

LISTING 27.15 Exploring Python String Functions

```
>>> st = "This is a test string"
>>> len(st)
21
>>> st.count('is')
2
>>> st.find('test')
10
>>> st.replace('test', 'new')
'This is a new string'
>>>
```

Where Python is really useful is in parsing applications. Python makes it easy to break a string down into its representative parts (into a list) that can then be further manipulated. The `split` method is used to create a list of the elements of a string. The list can then be manipulated normally, as shown in Listing 27.16, as an iterator. In this example, you iterate through the list `spstr`, capitalize each word with the `capitalize` method, and then create the new list `l`. This list is turned back into a string with the `join` method. Note here that `join` is applied to a string, which serves as the delimiter. Each string in the list is joined with the delimiter (separating each string), and the result is a new string.

LISTING 27.16 Python String Parsing Functions.

```
>>> spstr = st.split()
>>> print spstr
['This', 'is', 'a', 'test', 'string']
>>> l = []
>>> for i in spstr:
...     l.append(i.capitalize())
...
>>> print l
['This', 'Is', 'A', 'Test', 'String']
>>> jstr = " ".join(l)
>>> print jstr
This Is A Test String
>>>
```

Associative Arrays

Python includes built-in support for associative arrays, what in Python is called a dictionary. The dictionary is a way to retrieve information based upon not an index but instead by key. A dictionary is made up of key/value pairs; given the key, the value is returned. This is similar to an array, where given an array offset, a value is returned. But in this case, the key might be a value or a string. An example of a Python dictionary is provided in Listing 27.17.

LISTING 27.17 Creating and Using a Python Dictionary

```
>>> spouse = {"Tim":"Jill", "Bud":"Celeta", "Ernie":"Lila"}
>>> spouse["Tim"]
'Jill'
>>> spouse["Ed"] = "Maria"
>>> spouse
{'Tim': 'Jill', 'Ernie': 'Lila', 'Bud': 'Celeta', 'Ed': 'Maria'}
>>> del spouse["Tim"]
>>> spouse
{'Ernie': 'Lila', 'Bud': 'Celeta', 'Ed': 'Maria'}
>>>
```

In this example, you create a dictionary of spouses where the key is a string and the value is another string. The dictionary is defined as the key/value pairs delimited by a colon. After the spouse dictionary is created, you can treat it just like an array where the key is used as in the index (for example, `spouse["Tim"]`). Again, just like an array, you can add new elements to a dictionary by setting a new key with a new value. Finally, you can remove elements of a dictionary using the `del` function.

With `del`, you specify the dictionary and key, and the result is a dictionary without that key/value pair.

Classes and Methods

As Python supports object-oriented programming, classes are an integral part of program definition. In Python you can use existing classes, define your own classes, and also inherit from existing classes.

You can start with a simple class, and then move on to more complicated examples. All classes are defined with the `class` keyword (see Listing 27.18). Following this is the name of the class, which is typically capitalized. Within the class you define two methods. The first is a special method that serves as a constructor called `__init__`. This special method is invoked when a new class object is instantiated. It accepts two arguments, though only one is visible to the user. The `self` represents the instance of data for the particular class. This is discussed shortly.

In this constructor, you allow a string to be passed that represents the sound that the class instance makes when invoked. If the `sound` parameter is an empty string, then the default sound is used. Otherwise, the `sound` instance variable is copied from the user. Another method exists to emit the sound, which is called `my_sound`. This method simply prints the sound that was initialized during the object creation (constructor) process.

LISTING 27.18 A Simple Python Class Example

```
class Dog:
    def __init__(self, sound):
        if sound != "":
            self.sound = sound
        else:
            self.sound = "bark"

    def my_sound(self):
        print self.sound
```

Now that you've reviewed the class, you can return to the `self` parameter. Recall that a class can be instantiated into a new object, and that object has its own data but shares the methods with all other instances of this class. The `self` represents the instance data, which is unique to each instance, though the methods might not be unique. This is why the first parameter to each method is `self`. You don't actually pass in `self`; it's your instance of that object and is managed internally.

Creating new instances of the `Dog` class is illustrated in Listing 27.19. Creating a new instance looks similar to calling a function (of the class name). The result is an

object instance for the class with its own instance data (represented internally by `self`). Note here that when `my_sound` is invoked, `self` is implied as passed (as associated with the particular instance, such as `zoe`). Another way to think about the `zoe.my_sound()` call is `my_sound(zoe)`.

LISTING 27.19 Instantiating New Instances of the `Dog` Class

```
>>> zoe = Dog("woof")
>>> zoe.my_sound()
woof
>>> maddie = Dog("")
>>> maddie.my_sound()
bark
>>>
```

Now take a look at an example of multiple inheritances in Python. You can have a class with multiple base classes or a class with a single inherited base class (as is going to be demonstrated here). In this example, you have a class called `Thing` that inherits the class `Size`. Class `Size` inherits class `Color`, and class `Color` inherits class `Shape`. Each includes instance variables and their own set of class methods. This hierarchy is shown in Figure 27.1.

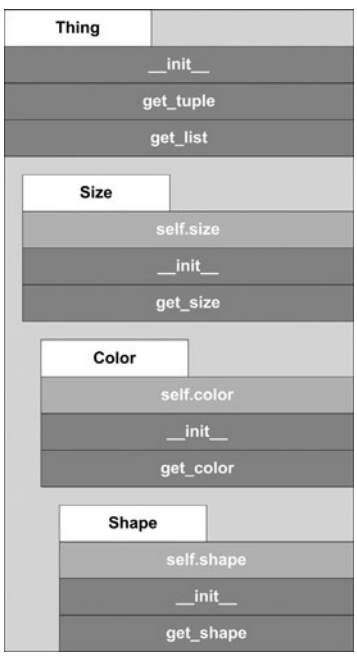

FIGURE 27.1 A Graphical View of the Listing 27.20

Note in Listing 27.20 the use of the named __init__ calls. In the constructors, the __init__ function of its base class is called by name. This allows the specific inherited class's __init__ function to be called directly. Finally, within the Thing class, you provide two helper functions that are used to return a list and tuple representation of the class's data.

LISTING 27.20 Demonstrating Python's Multiple Inheritance

```
class Shape:

        def __init__(self, shape):
                self.shape = shape

        def get_shape(self):
                return self.shape


class Color(Shape):

        def __init__(self, color, shape):
                Shape.__init__(self, shape)
                self.color = color

        def get_color(self):
                return self.color


class Size(Color):

        def __init__(self, size, color, shape):
                Color.__init__(self, color, shape)
                self.size = size

        def get_size(self):
                return self.size


class Thing(Size):

        def __init__(self, size, color, shape):
                Size.__init__(self, size, color, shape)
```

```
    def thing_tuple(self):
            return (self.get_size(), self.get_color(),
            self.get_shape())

    def thing_list(self):
            return [self.get_size(), self.get_color(),
            self.get_shape()]
```

The use of multiple inheritances is not without its issues, but in certain cases it can be useful. Avoiding name conflicts is one of the issues to be careful of, but with proper naming conventions, this can be easily avoided.

Advanced Features

Now it's time to explore some of the advanced features of Python that set it apart from other languages. This section reviews Python's dynamic code, functional programming, exception management, and a few other features.

Dynamic Code

The ability to treat code as data and data as code is one of the most striking aspects of dynamic languages. In functional languages this is called a higher order function (take a function as input, or present a function as output). Listing 27.21 demonstrates this capability in Python. First, you create a function that simply returns a square of a number. This is demonstrated next, by summing two squares. Next, you create a list that contains elements of code (summing two squares). This is joined to create a string and then passed to `eval` to evaluate the new string as code. As demonstrated, you can dynamically create new code and then execute it within the context of Python program.

LISTING 27.21 Higher Order Functions in Python

```
>>> def square(x):
...     return x*x
...
>>> square(5)+square(7)
74
>>> l = ['square', '(5)', '+', 'square', '(7)']
>>> mycode = "".join(l)
>>> print l
['square', '(5)', '+', 'square', '(7)']
>>> print mycode
```

```
square(5)+square(7)
>>> eval(mycode)
74
>>>
```

Functional Programming

In the last section, you explored higher order functions in Python, one of its many functional programming paradigms. Even earlier in this chapter, you looked at the `map` operation as a way to simplify and build more readable programs. This section now explores some of the other functional elements of Python.

Recall that the `map` function allows you to apply a user-defined function to each element of a list, providing an iterator with much greater readability. The `map` function results in a new list of the return values. Another iterating function that operates over a list is `reduce`. The difference is that whereas `map` and `reduce` operate over a list, `reduce` consumes the element of the list until its empty. An example is shown in Listing 27.22. This example results in a series of multiplications between each element of the list. The `range` results in a list consisting of `[1, 2, 3, 4]`, which when multiplied (1*2*3*4) results in 24.

LISTING 27.22 Using the `reduce` Function for List Computation

```
>>> def mul(x,y):
...     return x*y
...
>>> reduce(mul, range(1,5))
24
>>>
```

Python also includes an important aspect of functional programming called `lambda` functions. `lambda` functions are anonymous functions that are created at run time through the `lambda` construct. Take a look at an example of a creating and using a `lambda` function in Python (see Listing 27.23).

LISTING 27.23 Creating and Using a `lambda` Function

```
>>> g = lambda x: x*x
>>> g(7)
49
>>>
```

This example looks very similar to creating a function, but these functions can be treated differently from standard Python functions. Take a look now at an example of creating dynamic functions with `lambda`. The example shown in Listing 27.24 creates a new function at run time based upon an argument. Here you create a multiplier function at run time that's tailored at run time. The `construct_multiplier` function actually returns a `lambda` function based upon the argument defined by the user. This new function is then used to create two dynamic functions with different multipliers.

LISTING 27.24 Creating Dynamic Functions with `lambda`

```
>>> def construct_multiplier(x): return lambda n: x*n
...
>>> mulby2 = construct_multiplier(2)
>>> mulby7 = construct_multiplier(7)
>>>
>>> print mulby2(10)
20
>>> print mulby7(4)
28
>>>
```

Exception Handling

Python includes a rich exception handling mechanism that allows you to handle run-time errors or special conditions in your applications. This is similar to exception handling in other languages, where a `try` clause surrounds the block of code to check and an `except` clause provides the error-handling code.

```
try:
    …block of code to check…
except:
    …error handling…
```

Now take a look at a concrete example. Listing 27.25 demonstrates a simple example of exception handling. In this case, you create a function to create a list from a string and return the list. But if the argument passed to the function is not a string, you need to handle this rather than simply raising an exception (which is explored shortly). If something other than a list is provided, you simply return an empty list.

LISTING 27.25 Handling Exceptions with `try/except`

```
>>> def str2list(str):
...     try:
...             return str.split()
...     except:
...             return []
...
>>> print str2list("this is a test string")
['this', 'is', 'a', 'test', 'string']
>>> print str2list(50.19)
[]
>>>
```

The only problem with this kind of solution is that you hide the error from the caller. This can be useful in many cases, but in others, it's necessary to notify the caller of the issue so that it can deal with it in the way it needs. In this case, you remove the exception handling from the `str2list` function and instead move it to the caller (as shown in Listing 27.26).

LISTING 27.26 Handling Named Exceptions

```
>>> def str2list(str):
...     return str.split()
...
>>> try:
...     print str2list(5)
... except AttributeError:
...     print "caught attribute error"
...
caught attribute error
>>>
```

In this case, you move the exception handling code from the function out to the calling code and also update it to handle a specific exception. Within the `except` clause, you define the specific error that is to be handled (in this case, you know that if an integer is passed, an `AttributeError` results). This allows the caller to handle the issue in whichever way the caller determines. A list of exceptions can be provided in parenthesis, if multiple exceptions are to be handled by a single `except` clause. A user can also raise existing or new exceptions with the `raise` statement.

Python also allows the creation of new exceptions (which are classes in themselves). A list of the standard exceptions is provided in Table 27.2.

TABLE 27.2 Typical Exceptions in Python (Incomplete)

Python Exception	Meaning
`ArithmeticError`	Indicates overflow, zero division, floating-point error
`AssertionError`	Raised when an assert fails
`AttributeError`	Object does not support attribute references
`MemoryError`	Operation runs out of memory
`NameError`	Local or global variable not found in namespace
`SystemError`	Internal interpreter error

Summary

Hopefully this short review has provided a sample of Python's strengths and why it's one of the most popular scripting languages available today. With Python's built-ins and massive standard library, it's easy to build simple programs that accomplish amazing things in code on an order of magnitude smaller than other high-level languages. Python is yet another object-oriented scripting language in an already crowded scripting space, but it is well worth your time to learn and use.

Resources

Python Programming Language—Official Web site at http://www.python.org/.
Wikipedia Python Programming Language at http://en.wikipedia.org/wiki/Python_(programming_language).
Python online documentation at http://docs.python.org/.

This page intentionally left blank

28 GNU/Linux Administration Basics

In This Chapter

- Navigating the Linux Filesystem
- Managing Packages in GNU/Linux
- Managing Your Kernel

Introduction

This chapter explores GNU/Linux system administration from the developer's perspective. This means that topics such as basic configuration, user management, and so forth, are not covered. Instead, the chapter explores navigating the Linux filesystem, package management, kernel management, and other topics.

Navigating the Linux Filesystem

With so many UNIX and Linux distributions it's easy for them to become forked and therefore incompatible. One of the main areas where incompatibility can result is with the filesystem (its structure and where things are stored). Luckily, the Filesystem Hierarchy Standard (FHS) was created to address this problem (not only for the large number of Linux distributions, but also for BSD and other UNIX-like operating systems). You can find more information about the FHS at its home http://www.pathname.com/fhs/.

The FHS defines which subdirectories appear in the root (`/`) filesystem and what their contents should be (see Table 28.1).

TABLE 28.1 Standard Subdirectories of the FHS

Directory	Purpose
`/`	Primary root hierarchy (for the entire root filesystem)
`/bin`	Primary commands used in single-user mode
`/sbin`	System binaries
`/dev`	Devices files (user-space to kernel links)
`/etc`	System-wide configuration files
`/home`	Home directories for local users
`/lib`	Libraries needed for binaries in `/bin` and `/sbin`
`/mnt`	Directories for temporary mounted filesystems
`/opt`	Directory for optional software packages
`/proc`	Virtual filesystem (kernel status and user-space process status)
`/tmp`	Directory for temporary files
`/usr`	User-dependent hierarchy (`/usr/bin`, `/usr/sbin`, `/usr/include`, and so on)
`/var`	Variable files (such as mail and print spools, e-mail, logs, and so on)

The FHS goes into greater detail, but this first-level introduction provides some perspective on the contents of the root filesystem. The standard actually delves into considerable detail regarding the location of files, including the rationale, and is therefore useful to peruse.

Package Management

Package management is the process by which software packages are maintained (installed, upgraded, or uninstalled). Software packages are commonly distributed within a single file (for simplicity) but can result in many files installed in numerous places within a system. The package includes not only the software itself (in source or binary form) but also automated instructions on how and where to install the package.

This section explores some of the package management possibilities, starting with the most basic.

Tarball Distribution

The most common and platform-independent way to distribute software packages is through a compressed tarball. A tarball is a file format that is created and manipulated using the `tar` command. The tarball typically requires a bit more work, but is so common that it's useful to understand.

The `tar` command's name comes from its initial use as a format for tape backup. The name itself is derived from "tape archive" but is standardized under the POSIX umbrella. The `tar` utility retains its heritage from its old tape usage days. For example, access (like a tape) is sequential. Using `tar`, if the last file is to be extracted, the entire archive is scanned to get to that file (just as it would in the case of tape hardware).

Many software packages are distributed as tarballs because tarballs are universally understood and compress to a manageable size for storage and transfer. Distributing software using a tarball typically implies a set of other tools (such as `automake`, `autoconf`, `configure`, and so on), which are described in Chapter 8, "Building Packages with `automak/autoconf`."

Tarballs typically follow a standard naming convention so it's simple to know how a file was constructed. A standard tarball is named `file.tar`, which indicates an uncompressed tarball. If the tarball is compressed, it is named `file.tar.gz` for gzip compression and `file.tar.bz2` for bzip2 compression. As you see later in the chapter, tarballs can be automatically compressed or decompressed through the `tar` command.

After a tarball is downloaded, a common place to manipulate it is in `/usr/local/src`. Recall that this location stores local data that is specific to this host. If the tarball is to be removed after its software package is installed, then a better place is `/tmp`. This location is ideal because it is routinely and automatically purged. But it's sometimes desirable to keep the source around, so in this example, the former directory is used. Next (see Listing 28.1), the particular tarball to be installed is downloaded (in this example, the latest version of Ruby is used for demonstration purposes).

With the compressed Ruby tarball, you invoke `tar` on the tarball with three options. The first option `x` indicates that you're extracting an archive (instead of using `c` to create one). Next, you specify that a filename follows on the command line (not using `STDIN`). Finally, you specify `z` to indicate that it's a `gzip`-compressed archive and to gunzip the tarball automatically. The result is a new directory (in this case `ruby-1.8.6-p111`), which contains the source to the Ruby object-oriented scripting language.

LISTING 28.1 Downloading the Tarball to the Working Directory

```
root@camus:~# cd /usr/local/src
root@camus:/usr/local/src# wget http://xyz.lcs.mit.edu/ruby/
ruby-1.8.6-p111.tar.gz
—18:35:21—  http://xyz.lcs.mit.edu/ruby/ruby-1.8.6-p111.tar.gz
           => `ruby-1.8.6-p111.tar.gz'
...
Length: 4,547,579 (4.3M) [application/x-tar]
18:35:41 (218.60 KB/s) - `ruby-1.8.6-p111.tar.gz' saved
[4547579/4547579]

root@camus:/usr/local/src# tar xfz ruby-1.8.6-p111.tar.gz
root@camus:/usr/local/src#
```

The next step is to build the source to your package, so you descend into the packages subdirectory. If the subdirectory contains a `Makefile` (which is the make specification for the `make` utility), then you typically just need to type **`make`** to build the source. If `Makefile` is not present, then you need to go through the configuration process. This process automatically creates the `Makefile` to build the source. This process is necessary because systems can differ in a number of ways. Tools, for example, can be stored in different places. You might be running a different shell than is expected. A compiler might not support the prerequisites of the package, or prerequisite packages might not exist (requiring a download of additional modules). The configure process automatically checks your system and builds the `Makefile` appropriately.

The remainder of the process is then quite simple (see Listing 28.2). First, run the `configure` script that's present in the working directory (where the source was just installed). Next, type `make` to build the source. The `configure` script can also be used to provide instructions for the configuration—for example, where to install, which options to enable, and so on. You can usually see a list of the available options by typing **`./configure —help`**.

LISTING 28.2 The `configure` and `make` Process

```
root@camus:/usr/local/src/ruby-1.8.6-p111# ./configure
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking target system type... i686-pc-linux-gnu
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
```

```
checking whether we are cross compiling... no
...
creating config.h
configure: creating ./config.status
config.status: creating Makefile
root@camus:/usr/local/src/ruby-1.8.6-p111#
root@camus:/usr/local/src/ruby-1.8.6-p111# make
gcc -g -O2  -DRUBY_EXPORT -D_GNU_SOURCE=1  -I. -I.  -c array.c
gcc -g -O2  -DRUBY_EXPORT -D_GNU_SOURCE=1  -I. -I.  -c bignum.c
...
make[1]: Leaving directory `/usr/local/src/ruby-1.8.6-p111/ext/zlib'
making ruby
make[1]: Entering directory `/usr/local/src/ruby-1.8.6-p111'
gcc -g -O2  -DRUBY_EXPORT -D_GNU_SOURCE=1  -L.  -rdynamic -Wl,-export-
dynamic   main.o  -lruby-static -ldl -lcrypt -lm   -o ruby
make[1]: Leaving directory `/usr/local/src/ruby-1.8.6-p111'
root@camus:/usr/local/src/ruby-1.8.6-p111#
```

Finally (see Listing 28.3), type **`make install`** to install the resulting images to their respective locations (as indicated in the `Makefile` and defined through configure). Other `make` options like `install` (which are rules within the `Makefile`) are usually available, including options such as `test` to run a configured test on the resulting image or `clean` to remove the built objects. To fully clean the distribution directory, `dist-clean` is typically included.

LISTING 28.3 The Final Installation Process

```
root@camus:/usr/local/src/ruby-1.8.6-p111# make install
./miniruby  ./instruby.rb —dest-dir="" —extout=".ext" —make="make" —
mflags="" —make-flags="" —installed-list .installed.list —mantype="doc"
installing binary commands
installing command scripts
installing library scripts
installing headers
installing manpages
installing extension objects
installing extension scripts
root@camus:/usr/local/src/ruby-1.8.6-p111# ruby —version
ruby 1.8.6 (2007-09-24 patchlevel 111) [i686-linux]
root@camus:/usr/local/src/ruby-1.8.6-p111#
```

At this point, the Ruby interpreter and associated tools have been installed and are available for use. While this process was relatively simple, it can be considerably

more complex if you system does not have the prerequisite resources (such as libraries or other packages necessary to complete the build and installation). The advantage to this approach is that it works in almost all cases and allows you to tailor the package to your specific needs.

But when this process goes badly, it can go very, very badly. For example, say the package you want to install has a dependency on a set of other packages. Each of these dependencies has its own set of dependencies. This can happen in practice, so there must be another way. Luckily, you have multiple ways in the form of package management systems.

Advanced Package Tool

At the other end of the package management spectrum is the advanced package tool, or Apt. Apt, which is used in the Ubuntu Linux distribution, is actually a front end for the `dpkg` package manager that originated in the Debian Linux distribution. Apt is a set of simple utilities that automates the secure retrieval, configuration, and installation of software packages.

Apt consists of a number of utilities under the `apt` prefix (as well as a more curses-based implementation that's a bit simpler to use). This section explores the basic `apt-` commands and then reviews the curses-based `aptitude`.

The `apt-get` command is the basic apt package handling utility. It serves a number of needs, such as installing new packages, upgrading packages, and even upgrading an entire system with the latest versions of installed packages. Table 28.2 lists some of the important functions of the `apt-get` utility.

TABLE 28.2 Commands for the `apt-get` Utility

`apt-get` **Command**	**Description**
`update`	Resynchronize the package index files from the repository.
`upgrade`	Install the newest versions of all packages currently installed.
`dist-upgrade`	Install newest versions as well as handle changing dependencies of the packages.
`install`	Install one or more new packages.
`remove`	Remove one or more installed packages.
`source`	Have apt retrieve source packages.
`check`	Update the package index files and check dependencies.
`autoclean`	Remove retrieved packages that are out of date.

A special file called `/etc/apt/sources.list` maintains the list of servers that are used for package synchronization. These servers keep track of packages, versions, dependencies, and availability. The local cache can be used with another utility called `apt-cache` to find packages that offer a particular feature. You can start with an example of finding a package with `apt-cache` and then install it with `apt-get`.

Say that you want to install the very useful embeddable scripting language Lua. The Lua language is a simple and extensible language that was designed to be embedded into other systems (such as games) to extend them in various ways. It's an interesting language, so it's useful to have it available. You start by using `apt-cache` with the `search` command to find all packages that have the feature `lua` (see Listing 28.4).

LISTING 28.4 Finding a Package with `apt-cache`

```
root@camus:~# apt-cache search lua
autogen - an automated text file generator
eperl - Embedded Perl 5 Language
example-content - Ubuntu example content
exuberant-ctags - build tag file indexes of source code definitions
...
libtolua-dev - Tool to integrate C/C++ code with Lua - development files
libtolua0 - Tool to integrate C/C++ code with Lua - runtime libraries
lighttpd-mod-cml - Cache meta language module for lighttpd
lua-mode - Emacs mode for editing Lua programs
lua40 - Small embeddable language with simple procedural syntax
lua40-doc - Documentation for the Lua 4.0 programming language
lua5.1 - Simple, extensible, embeddable programming language
lua5.1-doc - Simple, extensible, embeddable programming language
luasocket - TCP/UDP socket library for Lua 5.0
luasocket-dev - TCP/UDP socket library for Lua 5.0
```

As you can see from this example, `apt-cache` finds many packages with `lua` in the name (in actual fact, over 60 were found). But you can see at the end that the package you want is `lua5.1`, so you want to install this using `apt-get`. You use the `install` command to `apt-get` and specify the particular package that you want to install, as shown in Listing 28.5.

LISTING 28.5 Installing a New Package with `apt-get`

```
root@camus:~# apt-get install lua5.1
Reading package lists... Done
Building dependency tree... Done
```

```
The following NEW packages will be installed:
  lua5.1
0 upgraded, 1 newly installed, 0 to remove and 58 not upgraded.
Need to get 112kB of archives.
After unpacking 283kB of additional disk space will be used.
Get:1 http://us.archive.ubuntu.com dapper/universe lua5.1 5.1-1 [112kB]
Fetched 112kB in 1s (69.6kB/s)
Selecting previously deselected package lua5.1.
(Reading database ... 33053 files and directories currently installed.)
Unpacking lua5.1 (from .../archives/lua5.1_5.1-1_i386.deb) ...
Setting up lua5.1 (5.1-1) ...

root@camus:~# lua
Lua 5.1  Copyright (C) 1994-2006 Lua.org, PUC-Rio
>
```

As can be seen from Listing 28.5, the package list and dependency tree are read, and the package to be installed is found. The `apt-get` utility determines the size of the archive (what's going to be downloaded) and also how much room is required for the install. Then the package is downloaded, unpacked, and installed. In the end, the `lua` interpreter is executed to demonstrate the completed process.

If Lua has any dependencies that are not satisfied on the system, those packages are also downloaded and installed to ensure that the required package is complete.

Now to the real strength of `apt`: It also includes the ability to remove packages. This is something that's not possible with the old fashioned `install` process, but because `apt` keeps track of what is installed for a package, it can use this information to remove what was installed. Listing 28.6 demonstrates removing a package with `apt-get`.

LISTING 28.6 Removing an Installed Package with `apt`

```
root@camus:/home/mtj# apt-get remove lua5.1
Reading package lists... Done
Building dependency tree... Done
The following packages will be REMOVED:
  lua5.1
0 upgraded, 0 newly installed, 1 to remove and 58 not upgraded.
Need to get 0B of archives.
After unpacking 283kB disk space will be freed.
Do you want to continue [Y/n]? Y
(Reading database ... 33060 files and directories currently installed.)
Removing lua5.1 ...
root@camus:/home/mtj#
```

Apt is but one example of the package managers available for GNU/Linux. Some of the other useful package managers are provided in Table 28.3.

TABLE 28.3 Other Useful Package Managers for GNU/Linux

Name	Description
`dpkg`	The Debian package manager (on which `apt` is based)
`rpm`	Red Hat Package Manager
`yum`	Yellow Dog Updater
`pacman`	Package Manager for Arch Linux
`slapt`	Slackware Package Manager (`apt`-like).

The aptitude tool provides a text-oriented graphical front end to `apt-get` (as shown in Figure 28.1). If you prefer this type of control, this tool is a great front end to `apt`.

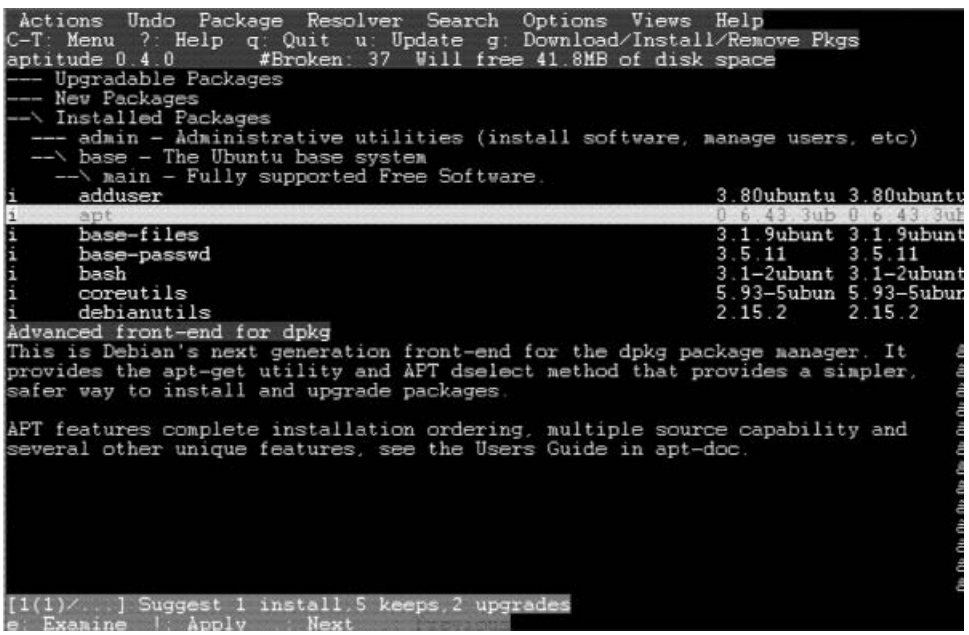

FIGURE 28.1 The graphical front end to `apt`, aptitude.

Kernel Upgrades

Even if you never plan on making changes to the Linux kernel, it's worth knowing how to configure and install a new kernel. For example, improvements to kernels are being made all of the time, so evolving with the kernel allows you to take advantage of these evolutions. Consider the latest happenings in virtualization. Rather than waiting for a new Linux distribution that includes these enhancements, you can download the latest stable kernel and take advantage right away.

The other advantage is that the Linux kernel delivered in the typical Linux distributions was configured and built to run on virtually any of the x86-based platforms. This means that it's extremely versatile, but not ideal in terms of performance. What's desired is a kernel that's built for the system on which it's going to be run. This is a great reason to learn how to configure and build the Linux kernel.

Getting the Latest Kernel

The Linux kernel archives are kept at the Kernel.org site (http://www.kernel.org). At this site you find not only current and past kernels, but also patches and information about the development trees and mailing lists.

Before downloading the new kernel, start by moving to the subdirectory where the new kernel should go (`/usr/src/linux`). From here you download the kernel, uncompress it with `bunzip2`, and untar it into a kernel tree. This process is shown in Listing 28.7.

LISTING 28.7 Downloading and Installing the Kernel Source

```
[root@plato ~]# cd /usr/src
[root@plato src]#
[root@plato src]# wget http://kernel.org/pub/linux/kernel/v2.6/
linux-2.6.23.1.tar.bz2
Resolving kernel.org... 204.152.191.5, 204.152.191.37
Connecting to kernel.org[204.152.191.5]:80... connected.
…
12:31:19 (343.51 KB/s) - `linux-2.6.23.1.tar.bz2' saved
[45,477,128/45,477,128]

[root@plato src]#
[root@plato src]# bunzip2 linux-2.6.23.1.tar.bz2
[root@plato src]# tar xf linux-2.6.23.1.tar
[root@plato src]# cd linux-2.6.23.1
[root@plato linux-2.6.23.1]#
```

The result of Listing 28.7 is a Linux kernel source tree at `/usr/src/linux-2.6.23.1`. The next step is to configure the kernel and build.

Configuring the Kernel

Kernel configuration is a process by which you define the kernel that you want to build. This includes the elements to include in the kernel, the drivers to build (and those to build as modules), as well as many (many) other options. For example, you can define the specific processor on which the kernel runs (which can be beneficial to define for performance). This section starts with a review of some of the options for kernel configuration.

When a kernel component is compiled into the kernel, it adds to the kernel size and required resources. When a component is compiled as a module, it is not included in the kernel and is loaded only when it's needed. This means that even though many components are defined as modules, only those used by your system are actually loaded when they're needed. There's also no performance benefit to keeping a component in the kernel compared to compiling as a module. Therefore, configuring components as modules is clearly advantageous.

The kernel includes a number of mechanisms for configuration, each initiated from the kernel `Makefile`. The oldest method (`make config`) is a command-line interface that enumerates each kernel configuration option, one at a time. This is tedious and is rarely used these days. The second method (`make menuconfig`) uses a text-mode GUI and is much faster and easier (see Figure 28.2).

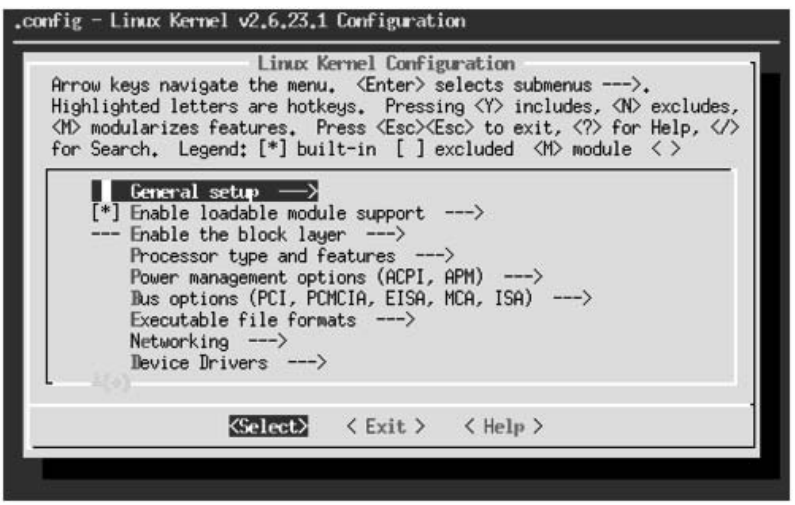

FIGURE 28.2 The text-oriented GUI of `make menuconfig`.

The kernel source also includes a graphical configuration tool that's based on Trolltech's Qt toolkit. This is a great configuration editor, but unfortunately the Qt toolkit isn't installed by default on all Linux systems. Therefore, if you intend to use this, you need to download the freely available version.

After you've waded through the more than 2,400 options, the result is a configuration file called `.config`. This human-readable text file contains all of the options and specifies which are enabled, which are disabled, and which are to be compiled as modules.

Building the Kernel

After the kernel has been configured, the next step is to build the kernel. Typing **`make all`** builds both the kernel and kernel modules (which used to require two steps):

```
[root@plato linux-2.6.23.1]# make all
```

If you're curious, you can also type **`make help`**, which emits each of the available options (from kernel configuration to kernel build and beyond). Depending upon the number of options enabled and the speed of the build machine, the build process can take anywhere from a short time to a very, very long time. When the build process completes, it's time to install the kernel and the associated kernel modules.

Prior to beginning the kernel configuration process, you might find it's useful to collect some information about the target hardware environment. This can be accomplished in a number of ways, but two of the most important elements here are `lspci` *and* `/proc/cpuinfo`*. The* `lspci` *command identifies the PCI devices that are attached to the system (helping out in the configuration of devices). The* `/proc/cpuinfo` *file identifies the processor on which the system is running, and can be used to build a kernel that runs efficiently on the target hardware.*

Installing the Kernel

Installing the kernel involves installing the kernel modules, compressed Linux image, and system map file. Installing the kernel modules is performed very simply as follows:

```
[root@plato linux-2.6.23.1]# make modules_install
```

This step installs the kernel modules and associated information into `/lib/modules` (into a new subdirectory of the kernel version, in this case `2.6.23.1`).

Next, the kernel and system map are installed into the boot directory. This is a special directory that is accessed during the boot process. Copying the kernel and system map is done as follows:

```
[root@plato linux-2.6.23.1]# cp arch/i386/boot/bzImage /boot/
vmlinuz-2.6.23.1
[root@plato linux-2.6.23.1]# cp System.map /boot/System.map-2.6.23.1
```

The file `bzImage` is the bzip compressed kernel image (that resulted during the early kernel build process). The `System.map` contains the symbol table for the new kernel image. In other words, the `System.map` contains the names of all symbols along with their addresses. This is used for kernel debugging (oops information is resolved using data from this file). The symbol information is available dynamically through the kernel through the `/proc/kallsyms` file.

Some systems require an initial ramdisk image, so it's safe to create an initial ramdisk image using the following command in case it's necessary:

```
[root@plato linux-2.6.23.1]# mkinitrd /boot/initrd-2.6.23.1 2.6.23.1
```

Configuring the Bootloader

The final step before testing the new kernel is to tell the bootloader about the new kernel image. Most Linux distributions today use GRUB (GRand Unified Bootloader), which has taken over from the older LILO (LInux LOader). They accomplish the same task, but GRUB is preferable (a new and improved LILO).

The file `/boot/grub/grub.conf` tells the bootloader about the available kernel images that can be booted. This file defines (for each kernel) the disk for the root filesystem, the kernel image (and options), and the initial ramdisk image. For the kernel image created in this section, Listing 28.8 demonstrates the kernel information that allows GRUB to boot this new image.

LISTING 28.8 Configuring the GRUB Bootloader for the New Kernel

```
title Kernel-2.6.23.1 (2.6.23.1)
        root (hd0,0)
        kernel /vmlinuz-2.6.23.1 ro root=/dev/VolGroup00/LogVol00 rhgb
        quiet
        initrd /initrd-2.6.23.1.img
```

After the GRUB configuration is modified, the remaining step is to reboot the system and then hit the Enter key when the GRUB prompt appears. At this point, the new kernel can be selected and booted.

Summary

The administration of GNU/Linux systems is much more than user management and service configuration. Understanding the layout of the GNU/Linux filesystem, managing packages and their dependencies, and keeping up to date on the latest Linux kernels are important topics in developer-based administration. Upgrading the Linux kernel not only keeps your system up to date, but it also makes available the latest features of the kernel (such as virtualization or new storage technologies such as Serial Attached SCSI). Therefore, it's an important skill to both know and use.

Part V Debugging and Testing

This final part of the book looks at the topics of debugging and testing. This includes unit testing frameworks, using the GNU source-level debugger, advanced debugging techniques (including memory debugging), and finally code-hardening techniques for creating higher quality and more reliable GNU/Linux applications.

Chapter 29: Software Unit Testing Frameworks

The topic of software testing is an important one with quite a bit of development in the open source community. After an introduction to unit and system testing, unit testing frameworks are explored, including a look at how to make your own and at two open source distributions. The `expect` utility is also covered as a means to test applications at a high level.

Chapter 30: Debugging with GDB

The GNU Debugger (GDB) is a source-level debugger that is a staple for GNU/Linux application development. GDB is integrated into the GNU toolchain and offers both command-line and GUI versions. This chapter presents GDB in a tutorial fashion and walks through the debugging of a simple application using breakpoints. Features such as data inspection and stack frame viewing and the GDB commands used most are covered. More advanced features for multiprocess and multithreaded application debugging and core-dump file debugging are also discussed.

Chapter 31: Code Hardening

The topic of code hardening, otherwise known as defensive programming, is an umbrella for a variety of techniques that have the goal of increasing the quality and reliability of software. This chapter looks at numerous coding methods as well as tools (such as static source-checking tools) to help build better software.

Chapter 32: Coverage Testing with GNU `gcov`

In this chapter, the topic of testing is analyzed from the perspective of coverage testing using the `gcov` utility. The `gcov` utility provides a way to identify path execution of an application. This tool can be very useful in determining full test path coverage of an application (where all paths are taken for a given regression of an application). It can also be useful in identifying how often a given path was taken and, therefore, is a useful performance tool.

Chapter 33: Profiling with GNU `gprof`

Profiling tools can be useful in identifying where the majority of time is taken for a given application. This chapter investigates the `gprof` utility, which can be used to help focus optimization efforts in an application by profiling the application to see where the majority of time is spent.

Chapter 34: Advanced Debugging Topics

This final chapter explores a number of debugging topics including memory debugging. Some of the most difficult debugging can center around memory management, so this chapter reviews a number of techniques and open source tools for supporting memory debugging. We'll also explore some of the cross-referencing tools as well as some useful tracing tools.

29 Software Unit Testing Frameworks

In This Chapter

- Unit Testing Versus System Testing
- Brew Your Own Frameworks
- Testing with the C Unit Test Framework
- Testing with the Embedded Unit Test Framework
- Testing with `expect`

Introduction

Writing software is difficult, even when building on an outstanding operating system such as GNU/Linux. The complexity of software grows as the sizes of the systems that you develop grow. But even when you are developing smaller systems, problems can still find their way in. This is where testing comes in. Even if you develop software that doesn't work the first time, performing tests on the software can identify the shortcomings, allowing you to fix them.

If the tests are repeatable, you can easily retest your software after you have made changes to it (otherwise known as *regressing*). This makes it much easier to update your software, because you know it's easy to verify that you haven't broken anything when you're done. If you do find something that's broken but wasn't tested before, you simply update the regression test to check it in the future.

This chapter looks at a number of available open source unit testing frameworks and how they can be used to improve the quality of your software.

Unit Testing

First, you need to understand what is meant by "unit" testing. For this discussion, it's best to divide testing into two unique categories. The first is end-to-end (or system) tests, which test specific user-level features of the software (see Figure 29.1).

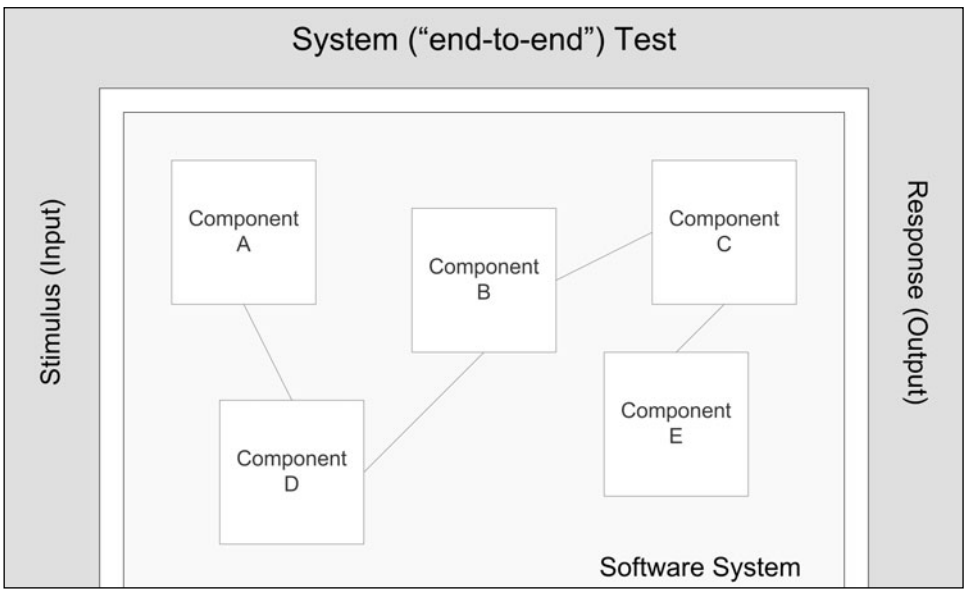

FIGURE 29.1 System testing (or end-to-end) perspective.

You can also think about this as testing based upon the requirements of the software. In this category, you typically don't consider how the software is constructed (black-box testing) but instead simply test for what the system should do. As illustrated in Figure 29.2, the components of your system can be further broken down into smaller modules called *units*.

Unit testing assumes more knowledge and insight into the software to be tested. Unit tests address units of a software system and, therefore, don't typically address system requirements but instead the internal behavior of the system. Say that your software system included a queue unit that was to be used for internal task communication. You can separate this queue unit from the rest of the system and test it in isolation with a number of different tests. This is a unit test, because you are addressing a unit of the system (see Figure 29.3). You also commonly consider how the unit was constructed and, therefore, test with this knowledge (white-box testing). This permits you to ensure that you've tested all of the elements of the unit (recall the use of the GNU `gcov` utility in Chapter 32, "Coverage Testing with GNU `gcov`").

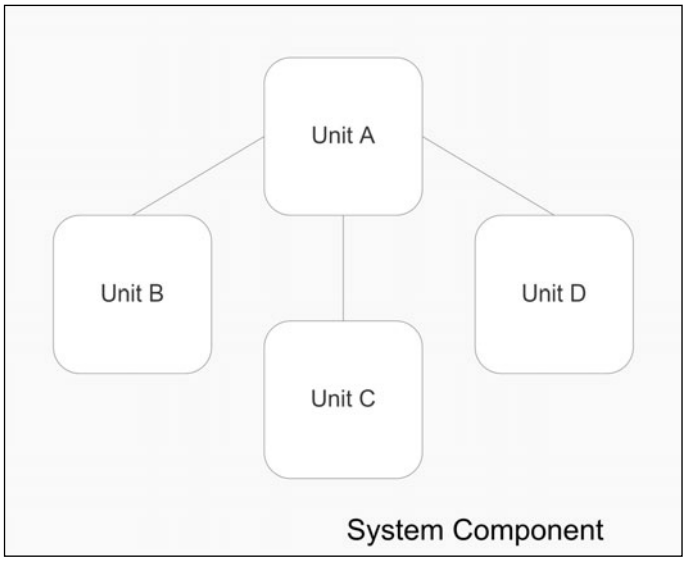

FIGURE 29.2 System components are made up of units.

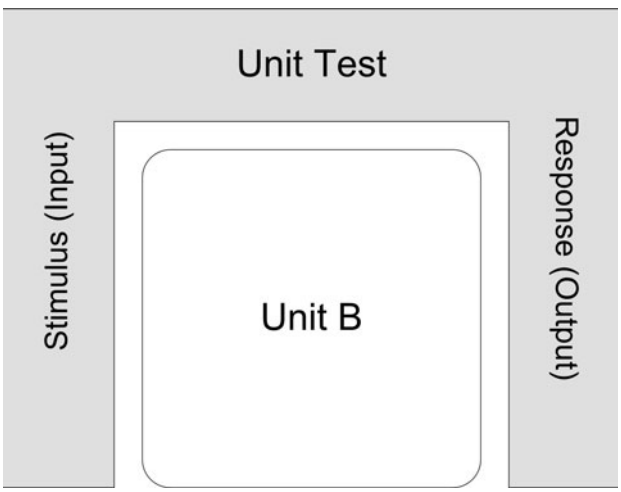

FIGURE 29.3 Unit testing perspective of an individual software unit.

Therefore, a unit test simply invokes the unit's APIs and verifies that a given stimulus produces an expected result.

Unit Testing Frameworks

Now that you've identified the scope of the unit test, it's time to look at some unit testing frameworks to explore how they can be used to increase the quality of your software. The following are the frameworks reviewed in the sections that follow.

- Brew your own
- C unit test (cut) system
- Embedded unit test
- `expect`

Brew Your Own

Building your own simple unit test framework is not difficult. Even the simplest architecture can yield great benefits. Take a look at a simple architecture for testing a software unit.

Consider that you have a simple stack module with an API consisting of the following:

```
typedef struct { ... } stack_t;
int stackCreate( stack_t *stack, int stackSize );
int stackPush( stack_t *stack, int element );
int stackPop( stack_t *stack, int *element );
int stackDestroy( stack_t *stack );
```

This very simple Last-In-First-Out (LIRO) stack API permits you to create a new stack, push an element on the stack, pop an element from the stack, and finally destroy the stack. Listing 29.1 shows the code (`stack.c`) for this simple module, and Listing 29.2 shows the header file (`stack.h`).

LISTING 29.1 Stack Module Source (on the CD-ROM at `./source/ch24 /byo/stack.c`)

```
 1:    #include <stdlib.h>
 2:    #include "stack.h"
 3:
 4:
 5:    int stackCreate( stack_t *stack, int stackSize )
 6:    {
 7:      if ((stackSize == 0) || (stackSize > 1024)) return -1;
 8:
 9:      stack->storage = (int *)malloc( sizeof(int) * stackSize );
10:
```

```
11:       if (stack->storage == (void *)0) return -1;
12:
13:       stack->state = STACK_CREATED;
14:       stack->max = stackSize;
15:       stack->index = 0;
16:
17:       return 0;
18:     }
19:
20:
21:     int stackPush( stack_t *stack, int element )
22:     {
23:       if (stack == (stack_t *)NULL) return -1;
24:       if (stack->state != STACK_CREATED) return -1;
25:       if (stack->index >= stack->max) return -1;
26:
27:       stack->storage[stack->index++] = element;
28:
29:       return 0;
30:     }
31:
32:
33:     int stackPop( stack_t *stack, int *element )
34:     {
35:       if (stack == (stack_t *)NULL) return -1;
36:       if (stack->state != STACK_CREATED) return -1;
37:       if (stack->index == 0) return -1;
38:
39:       *element = stack->storage[—stack->index];
40:
41:       return 0;
42:     }
43:
44:
45:     int stackDestroy( stack_t *stack )
46:     {
47:       if (stack == (stack_t *)NULL) return -1;
48:       if (stack->state != STACK_CREATED) return -1;
49:
50:       stack->state = 0;
51:       free( (void *)stack->storage );
52:
53:       return 0;
54:     }
```

LISTING 29.2 Stack Module Header File (on the CD-ROM at `./source/ch24/byo/stack.h`)

```
 1:    #define STACK_CREATED   0xFAF32000
 2:
 3:    typedef struct {
 4:
 5:      int state;
 6:      int index;
 7:      int max;
 8:      int *storage;
 9:
10:    } stack_t;
11:
12:
13:    int stackCreate( stack_t *stack, int stackSize );
14:
15:    int stackCreate( stack_t *stack, int element );
16:
17:    int stackPop( stack_t *stack, int *element );
18:
19:    int stackDestroy( stack_t *stack );
```

Take a look at a simple regression that, when built with this stack module, can be used to verify that it works as expected. Because many individual tests can be used to validate this module, this section concentrates on just a few to illustrate the approach.

First, in this regression, you provide two infrastructure functions as wrappers for the regression. The first is a simple `main` function that invokes each of the tests, and the second is a result checking function. The result checking function simply tests the result input. If it's zero, the test failed; otherwise, the test passed. This function is shown in Listing 29.4 (which you will look at shortly).

You declare a failed integer, which is used to keep track of the number of actual failures. This is used by your `main`, which allows it to determine if the regression passed or failed. The `checkResult` function (Listing 29.3) takes two inputs: a test number and the individual test result. If the test result is zero, then you mark the test as failed (and increment the failed count). Otherwise, the test passes (result is nonzero), and you indicate this.

LISTING 29.3 Result Checking Function for a Simple Regression (on the CD-ROM at `./source/ch24/byo/regress.c`)

```
 1:    int failed;
 2:
 3:    void checkResult( int testnum, int result )
 4:    {
 5:      if (result == 0) {
 6:        printf( "*** Failed test number %d\n", testnum );
 7:        failed++;
 8:      } else {
 9:        printf( "Test number %2d passed.\n", testnum );
10:      }
11:    }
```

The `main` program simply calls your regression tests in order, clearing the failed count (as shown in Listing 29.4).

LISTING 29.4 Simple Regression `main` (on the CD-ROM at `./source/ch24/byo/regress.c`)

```
1:    int main()
2:    {
3:
4:      failed = 0;
5:      test1();
6:
7:      return 0;
8:    }
```

Now take a look at a regression that focuses on creation and destruction of stacks. As you saw in the stack module source, you encounter numerous ways that a stack creation and destruction can fail. This test tries to address each of them so that you can convince yourself that it's coded properly (see Listing 29.5).

In this regression, you call an API function with a set of input data and then check the result. In some cases, you pass good data, and in others you pass data that causes the function to exit with a failure status. Consider lines 8–9, which test stack creation with a null stack element. This creation should fail and return -1. At line 9, you call `checkResult` with your test number (first argument) and then the test result as argument two. Note here that you test the `ret` variable with -1, because that's what you expect for this failure case. If `ret` wasn't -1, then the expression results in 0, indicating that the test failed. Otherwise, if `ret` is -1, the expression reduces to 1, and the result is a pass.

This regression also explores the stack structure to ensure that the creation function has properly initialized the internal elements. At line 20, you check the internal state variable to ensure that it has been properly initialized with `STACK_CREATED`.

LISTING 29.5 Stack Module Regression Focusing on Creation and Destruction (on the CD-ROM at `./source/ch24/byo/regress.c`)

```
 1:    void test1( void )
 2:    {
 3:      stack_t myStack;
 4:      int ret;
 5:
 6:      failed = 0;
 7:
 8:      ret = stackCreate( 0, 0 );
 9:      checkResult( 0, (ret == -1) );
10:
11:      ret = stackCreate( &myStack, 0 );
12:      checkResult( 1, (ret == -1) );
13:
14:      ret = stackCreate( &myStack, 65536 );
15:      checkResult( 2, (ret == -1) );
16:
17:      ret = stackCreate( &myStack, 1024 );
18:      checkResult( 3, (ret == 0) );
19:
20:      checkResult( 4, (myStack.state == STACK_CREATED) );
21:
22:      checkResult( 5, (myStack.index == 0) );
23:
24:      checkResult( 6, (myStack.max == 1024) );
25:
26:      checkResult( 7, (myStack.storage != (int *)0) );
27:
28:      ret = stackDestroy( 0 );
29:      checkResult( 8, (ret == -1) );
30:
31:      ret = stackDestroy( &myStack );
32:      checkResult( 9, (ret == 0) );
33:
34:      checkResult( 10, (myStack.state != STACK_CREATED) );
35:
```

```
36:       if (failed == 0) printf( "test1 passed.\n");
37:       else printf("test1 failed\n");
38:     }
```

At the end of this simple regression, you indicate whether the entire test passed (all individual tests passed) or failed. A sample run of the regression is illustrated as follows:

```
# gcc -Wall -o test regress.c stack.c
# ./test
Test number  0 passed.
Test number  1 passed.
Test number  2 passed.
Test number  3 passed.
Test number  4 passed.
Test number  5 passed.
Test number  6 passed.
Test number  7 passed.
Test number  8 passed.
Test number  9 passed.
Test number 10 passed.
test1 passed.
#
```

While this method is quite simple, it's also very effective and permits the quick revalidation of a unit after changes are made. After you have run your regression, you can deliver it safely with the understanding that you are less likely to break the software that uses it.

C Unit Test System

The C unit test system, or *cut* for short, is a simple architecture for building unit test applications. Cut provides a simple environment for the integration of unit tests, with a set of tools that are used to pull together tests and then build a main framework to invoke and report them. Cut parses the unit test files and then builds them together into a single image, performing each of the provided unit tests in order.

Cut provides a utility, written in Python, called `cutgen.py`. This utility parses the unit test source files and then builds a main function around the tests that are to be run. The unit test files must be written in a certain way; you can see this in the sample source.

Now it's time to walk through an example that verifies the push and pop functions of your previous stack example. Cut presents a simple interface that we build to, which is demonstrated in Listing 29.6.

You declare your locals at line 5 and 6 (your two stacks for which you perform your tests) and then four functions that serve as the interface to cut.

The first two functions to note are the first and last. These are the initialization function, `__CUT_BRINGUP__Explode` (at lines 8–20), and the post test execution function, `__CUT_TAKEDOWN__Explode` (at lines 74–85). The cut framework calls the `bringup` function prior to test start, and after all of the test functions have been performed, the `takedown` function is called. Note that you simply perform your necessary initialization (initialize your two stacks with `stackCreate`), but you also perform unit testing here to ensure that the required elements are available for you in the unit test. Similarly, in the `takedown` function, which is called after all unit tests have been called, you destroy the stacks, but you also check the return status of these calls.

The unit tests are encoded as function with a prefix `__CUT__`. This allows the `cutgen` utility to find them and call them within the test framework. As you perform the necessary elements of your unit tests, you call a special function called `ASSERT`, which is used to log errors. The `ASSERT` function has two pieces: a test expression and a string emitted if the test fails. The test expression identifies the success condition, and if false, then the test element fails.

Note that in some cases, you have multiple tests for each API element (such as shown for lines 64–65).

LISTING 29.6 Unit Test Example Written for Cut (on the CD-ROM at `./source/ch24/cut/test_1.c`)

```
 1:    #include <stdio.h>
 2:    #include "stack.h"
 3:    #include "cut.h"
 4:
 5:    stack_t myStack_1;
 6:    stack_t myStack_2;
 7:
 8:    void __CUT_BRINGUP__Explode( void )
 9:    {
10:      int ret;
11:
12:      printf("Stack test bringup called\n");
13:
14:      ret = stackCreate( &myStack_1, 5 );
```

```
15:       ASSERT( (ret == 0), "Stack 1 Creation." );
16:
17:       ret = stackCreate( &myStack_2, 5 );
18:       ASSERT( (ret == 0), "Stack 2 Creation." );
19:
20:     }
21:
22:
23:     void __CUT__PushConsumptionTest( void )
24:     {
25:       int ret;
26:
27:       /* Exhaust the stack */
28:
29:       ret = stackPush( &myStack_1, 1 );
30:       ASSERT( (ret == 0), "Stack Push 1 failed." );
31:
32:       ret = stackPush( &myStack_1, 2 );
33:       ASSERT( (ret == 0), "Stack Push 2 failed." );
34:
35:       ret = stackPush( &myStack_1, 3 );
36:       ASSERT( (ret == 0), "Stack Push 3 failed." );
37:
38:       ret = stackPush( &myStack_1, 4 );
39:       ASSERT( (ret == 0), "Stack Push 4 failed." );
40:
41:       ret = stackPush( &myStack_1, 5 );
42:       ASSERT( (ret == 0), "Stack Push 5 failed." );
43:
44:       ret = stackPush( &myStack_1, 6 );
45:       ASSERT( (ret == -1), "Stack exhaustion failed." );
46:
47:     }
48:
49:
50:     void __CUT__PushPopTest( void )
51:     {
52:       int ret;
53:       int value;
54:
55:       /* Test two pushes and then two pops */
56:
57:       ret = stackPush( &myStack_2, 55 );
```

```
58:       ASSERT( (ret == 0), "Stack Push of 55 failed." );
59:
60:       ret = stackPush( &myStack_2, 101 );
61:       ASSERT( (ret == 0), "Stack Push of 101 failed." );
62:
63:       ret = stackPop( &myStack_2, &value );
64:       ASSERT( (ret == 0), "Stack Pop failed." );
65:       ASSERT( (value == 101), "Stack Popped Wrong Value.");
66:
67:       ret = stackPop( &myStack_2, &value );
68:       ASSERT( (ret == 0), "Stack Pop failed." );
69:       ASSERT( (value == 55), "Stack Popped Wrong Value." );
70:
71:     }
72:
73:
74:     void __CUT_TAKEDOWN__Explode( void )
75:     {
76:       int ret;
77:
78:       ret = stackDestroy( &myStack_1 );
79:       ASSERT( (ret == 0), "Stack 1 Destruction." );
80:
81:       ret = stackDestroy( &myStack_2 );
82:       ASSERT( (ret == 0), "Stack 2 Destruction." );
83:
84:       printf( "\n\nTest Complete\n");
85:     }
```

Now that you have your unit test file (encoded for the cut environment), you can look at how you make this an executable test. While this can be easily encoded in a simple Makefile, this example demonstrates command-line building.

You need three files from the cut system (the URL from which these files can be obtained is provided in the "Resources" section of this chapter). The `cutgen.py` utility builds the unit test environment, given your set of unit test source files. This is a Python file, so a Python interpreter is needed on the target system. Two other files are `cut.h` and `libcut.inc`, which are ultimately linked with your unit test application.

The first step is creating the cut unit test environment. This creates C `main` and brings together the necessary APIs used by the unit tests. The `cutgen.py` utility provides this for you, as demonstrated here:

```
cutgen.py test_1.c > cutcheck.c
```

Given your unit test file (Listing 29.6), you provide this as the single argument to `cutgen.py` and redirect the output into a source file called `cutcheck.c`. To further understand what `cutgen` has provided, you can now look at this file (shown in Listing 29.7).

The automatically generated file from `cutgen` simply brings together the unit tests present in your unit test files and calls them in order. You can provide numerous unit test files to `cutgen`, which results in additional unit test functions being invoked within the generated file.

LISTING 29.7 Unit Test Environment Source Created by `cutgen`

```
#include "libcut.inc"


extern void __CUT_BRINGUP__Explode( void );
extern void __CUT__PushConsumptionTest( void );
extern void __CUT__PushPopTest( void );
extern void __CUT_TAKEDOWN__Explode( void );


int main( int argc, char *argv[] )
{
  cut_init( -1 );

  cut_start( "Explode", __CUT_TAKEDOWN__Explode );
  __CUT_BRINGUP__Explode();
  __CUT__PushConsumptionTest();
  __CUT__PushPopTest();
  cut_end( "Explode" );
  __CUT_TAKEDOWN__Explode();


  cut_break_formatting();
  return 0;
}
```

Finally, you simply compile and link the files together to build a unit test image. This image implies the automatically generated source file, unit test file, and the source to test (`stack.c`). This image is illustrated here:

```
# gcc -o cutcheck stack.c test_1.c cutcheck.c
```

You can then execute the unit test by simply invoking `cutcheck`. This emits numbers and . characters to indicate progress through the unit test process.

```
# ./cutcheck
Stack test bringup called

    0.........   10.....

Test Complete

#
```

The cut system provides some additional features that have not been addressed here, but from this quick review, you can see how powerful and useful this simple utility can be.

Embedded Unit Test

The embedded unit test framework (called Embunit) is an interesting framework that's designed for embedded systems. The framework can operate without the need for standard C libraries and allocates objects from a `const` area. Embunit also provides a number of tools to generate test templates and also the `main` function for the test environment.

The Embunit framework is very similar to cut and provides a very useful API for testing. Some of the test functions that are provided in Embunit are shown in Table 29.1.

TABLE 29.1 Test Functions Provided by Embunit

Function	Purpose
`TEST_ASSERT_EQUAL_STRING(exp,actual)`	Assert on failed string compare.
`TEST_ASSERT_EQUAL_INT(exp,actual)`	Assert on failed integer compare.
`TEST_ASSERT_NULL(pointer)`	Assert if pointer is `NULL`.
`TEST_ASSERT_NOT_NULL(pointer)`	Assert if pointer is not `NULL`.
`TEST_ASSERT_MESSAGE(cond,message)`	Assert and emit message if the condition is false.
`TEST_ASSERT(condition)`	Assert if the condition is false.
`TEST_FAIL(message)`	Fail the test, emitting the message.

Now it's time to take a look at a unit test coded for the Embunit framework and then look at the `main` program that sets up the environment. In Listing 29.8, a minimal unit test for Embunit is shown. At line 1, you include the `embUnit` header file (which makes the test interface available to your unit test). You then define two functions that are called before and after each unique unit test that's identified to `embUnit`: `setUp` at lines 7–10 and `tearDown` at lines 13–16.

You then define your individual unit tests (lines 19–54). Three tests are illustrated here; the first is called `testInit` (lines 19–25), the second is called `testPushPop` (lines 28–45), and the third is called `testStackDestroy` (lines 48–54). As with your earlier unit tests, you perform an action and then test the result. In this case, you use the Embunit-provided `TEST_ASSERT_EQUAL_INT` function to check the response, and if the assert fails, an error message is printed.

Finally, you specify the unit tests that are available within the `StackTest_tests` function (lines 57–68). This is done in the context of a test fixtures structure. You define `fixtures` as simply an array that's initialized with the provided functions and function names. Note that you provide your three unit tests here, providing a name for each to indicate the specific test in the event a failure occurs. You then create another structure at lines 64–65 that defines your test, `setup` function, `teardown` function, and `fixtures` (your list of unit tests). This new structure (called `StackTest`) is returned to the caller, which is explored shortly.

LISTING 29.8 Unit Test Coded for the Embunit Framework (on the CD-ROM at `./source/ch24/emb/stackTest.c`)

```
 1:    #include <embUnit/embUnit.h>
 2:    #include <stdio.h>
 3:    #include "stack.h"
 4:
 5:    stack_t myStack;
 6:
 7:    static void setUp( void )
 8:    {
 9:      printf("setUp called.\n");
10:    }
11:
12:
13:    static void tearDown( void )
14:    {
15:      printf("tearDown called.\n");
16:    }
17:
18:
```

```
19:     static void testInit( void )
20:     {
21:       int ret;
22:
23:       ret = stackCreate( &myStack, 5 );
24:       TEST_ASSERT_EQUAL_INT( 0, ret );
25:     }
26:
27:
28:     static void testPushPop( void )
29:     {
30:       int ret, value;
31:
32:       ret = stackPush( &myStack, 55 );
33:       TEST_ASSERT_EQUAL_INT( 0, ret );
34:
35:       ret = stackPush( &myStack, 101 );
36:       TEST_ASSERT_EQUAL_INT( 0, ret );
37:
38:       ret = stackPop( &myStack, &value );
39:       TEST_ASSERT_EQUAL_INT( 0, ret );
40:       TEST_ASSERT_EQUAL_INT( 101, value );
41:
42:       ret = stackPop( &myStack, &value );
43:       TEST_ASSERT_EQUAL_INT( 0, ret );
44:       TEST_ASSERT_EQUAL_INT( 55, value );
45:     }
46:
47:
48:     static void testStackDestroy( void )
49:     {
50:       int ret;
51:
52:       ret = stackDestroy( &myStack );
53:       TEST_ASSERT_EQUAL_INT( 0, ret );
54:     }
55:
56:
57:     TestRef StackTest_tests( void )
58:     {
59:       EMB_UNIT_TESTFIXTURES( fixtures ) {
60:         new_TestFixture("testInit", testInit ),
61:         new_TestFixture("testPushPop"", testPushPop ),
62:         new_TestFixture("testStackDestroy", testStackDestroy ),
```

```
63:       };
64:       EMB_UNIT_TESTCALLER( StackTest, "StackTest",
65:                            setUp, tearDown, fixtures );
66:
67:       return( TestRef)&StackTest;
68:     }
```

The `EMB_UNIT_TESTFIXTURES` *and* `EMB_UNIT_TESTCALLER` *are macros that create special arrays representing the individual unit tests (*`fixtures`*) as well as the unit test aggregate (*`StackTest`*).*

The `main` program provides the means to invoke the unit tests (see Listing 29.9). You include the `embUnit` header file to gather the types and symbols. At line 3, you declare the previous function, which creates the text `fixtures` array (recall from Listing 29.8, lines 57–68). You call the `TestRunner_start` function to initialize the test environment and then invoke `TestRunner_runTest` with your unit test fixture `init` function (`StackTest_tests`). This invokes all of the unit tests from Listing 29.8. When done, you call `TestRunner_end`, which emits statistics about the unit test, including the number of tests run and the number of failed tests.

LISTING 29.9 Embunit `main` Program (on the CD-ROM at `./source/ch24/emb/main.c`)

```
1:    #include <embUnit/embUnit.h>
2:
3:    TestRef StackTest_tests( void );
4:
5:    int main( int argc, const char *argv[] )
6:    {
7:      TestRunner_start();
8:        TestRunner_runTest( StackTest_tests() );
9:      TestRunner_end();
10:     return 0;
11:   }
```

Building the unit test within Embunit simply involves compiling and linking your source files together with the Embunit library. To find the Embunit library, you must specify its location (as well as the location of the header files). Building and running the unit test is illustrated as follows:

```
# gcc -Wall -I/usr/local/src/embunit/ \
      -L/usr/local/src/embunit/lib \
      -o stackTest main.c stack.c stackTest.c  -lembUnit
# ./stackTest
```

```
.setUp called.
testInit called
tearDown called.
.setUp called.
tearDown called.
.setUp called.
tearDown called.

OK (3 tests)
#
```

Using GNU Compiler Collection (GCC), you build the image called `stackTest` and then invoke it. You can see that `setUp` and `tearDown` are called three times each, before and after each of your unit tests. In the end, you can see that the Embunit test environment reports that three tests were run and all were okay.

`expect` Utility

The `expect` utility has found wide use in the testing domain. `expect` is an application that scripts programmed dialogues with interactive programs. Using `expect`, you can spawn an application and then perform a script consisting of dialogue with the application. This dialogue consists of a series of statements and expected responses. In the test domain, the statements are the stimulus to the application, and the expected response is what you expect from a valid application under test.

The `expect` utility can even talk to numerous applications at once and has a very rich structure for application test.

Consider the following test in Listing 29.10. At line 3, you set an internal variable of `timeout` to 1. This tells `expect` that when the `timeout` keyword is used, the `timeout` represents 1 second (instead of the default of 10 seconds). Next, you declare a procedure called `sendexpect`. This function provides both the `send` and `expect` behaviors in one function. It sends the `out` string (argument one) to the attached process. The `expect` function in this case uses the pattern match behavior. Two possibilities exist for what you expect as input from the attached process. If you receive the `in` string (argument two of the `sendexpect` procedure), then you have received what you expected and emit a `passed` message. Otherwise, you wait, and when the timeout occurs, you call the test a failure and exit.

At line 14, you spawn the test process (in this case you are testing the `bc` calculator). You consume the input from `bc`'s startup by expecting the string warranty. You then emit an indicator to `stdout`, indicating that the test has started (line 17). At this point, you begin the test. Using the `sendexpect` procedure, you send a command to the application and then provide what you expect as a response. Because you are testing a calculator process, you provide some simple expressions (lines

20–22), followed by a slightly more complex function example. In the end, you emit the `bc quit` command, expecting `eof` (an indication that the `bc` application terminated).

LISTING 29.10 Testing the `bc` Calculator with `expect` (on the CD-ROM at `./source/ch24/expect/test_bc`)

```
#!/usr/local/bin/expect -f

set timeout 1

proc sendexpect { out in } {
  send $out
  expect {
    $in         { puts "passed\n" }
    timeout     { puts "***failed\n" ; exit }
  }
}

# Start the bc test
spawn bc
expect "warranty"

puts "Test Starting\n"

# Test some simple math
sendexpect "2+4\n" "6"
sendexpect "2*8\n" "16"
sendexpect "9-2\n" "7"

# Test a simple function
sendexpect \
  "define f (x) { if (x<=1) return(1); return(f(x-1) * x); }\n"
  "\r"
sendexpect "f(5)\n" "120"

# End the test session
sendexpect "quit\n" eof

puts "Test Complete\n"

exit
```

The `expect` method differs greatly from your unit test examples, but it is a very powerful mechanism not only for testing but also for automated tasks, even those on remote systems.

Summary

The unit testing discipline has improved with open source tools. These tools (and more complex ones such as the DejaGnu framework) provide efficient and simple mechanisms for unit testing. This chapter explored unit testing (as opposed to system testing) and investigated a number of methods to achieve it. First you reviewed a brew your own method for testing that was simple but got the job done. Then you reviewed two open source tools for unit testing, the C unit test system (cut) and the embedded unit test system. Finally, you took a quick look at `expect` and its capabilities for process-based testing.

Resources

The C unit test system at http://sourceforge.net/projects/cut/.
Embedded unit test framework at http://sourceforge.net/projects/embunit/.
Expect home page at http://expect.nist.gov/.

30 Debugging with GDB

In This Chapter

- Source Debugging with GDB
- Debugging Multiprocess Applications
- Debugging Multithreaded Applications
- Debugging Programs Already Running
- Post-Mortem Debugging

Introduction

The GNU Debugger (also known as GDB) is a source-level debugger that provides the ability to debug applications at the source and machine levels. Additionally, GDB permits the debugging of already running applications (by attaching to the application's process ID) as well as debugging applications post-mortem. All of the traditional features you expect from a source-level debugger are available with GDB, including multilanguage and multi-architecture support.

This chapter introduces GDB and explores its features and capabilities in a tutorial manner.

Compiling for GDB

Before you jump into GDB, you first need to know how to build your application so that it's debuggable by GDB. The `-g` flag tells the compiler to include debugging information in the image, which can be used by GDB to understand variable types (for data inspection) and machine instruction to source-line mappings. Compiling is illustrated as follows:

```
# gcc -g testapp.c -o testapp
```

The test image can now be successfully debugged via GDB. One very important point to note is that debugging with optimization enabled can yield odd results. The optimizer might move code around or remove code altogether. This can make an optimized debugging session confusing and hard to follow. Therefore, while GDB can still debug optimized code, it's much easier to debug an unoptimized image.

Using GDB

Now it's time to dive into GDB and look at its capabilities for debugging C applications. In this section you look at some of the most common methods for debugging with GDB using breakpoints. It demonstrates using command-line GDB, though GUI versions exist.

One important point to note before you jump in is that the program being debugged uses the same terminal input and output as GDB. This is suitable for your purposes here.

You can redirect `stdin` and `stdout` for the program's I/O. You can redirect the output on the command line when we start the GDB session. You can also specify a new terminal for the program's `stdin` using the `tty` shell command.

To demonstrate GDB, you can use the source shown in Listing 30.1. This source represents a very simple stack implementation that provides math operators.

LISTING 30.1 Sample Source for the GDB Debugging Session (on the CD-ROM at `./source/ch30/testapp.c`)

```
 1:    #include <stdio.h>
 2:    #include <assert.h>
 3:
 4:    #define MAX_STACK_ELEMS         10
 5:
 6:    #define OP_ADD                  0
 7:    #define OP_SUBTRACT             1
 8:    #define OP_MULTIPLY             2
 9:    #define OP_DIVIDE               3
10:
11:
12:    typedef struct {
13:      int stack[MAX_STACK_ELEMS];
14:      int index;
```

```
15:     } STACK_T;
16:
17:
18:     void initStack( STACK_T *stack )
19:     {
20:       assert( stack );
21:       stack->index = 0;
22:     }
23:
24:
25:     void push( STACK_T *stack, int elem )
26:     {
27:       assert( stack );
28:       assert( stack->index < MAX_STACK_ELEMS );
29:
30:       stack->stack[stack->index++] = elem;
31:       return;
32:     }
33:
34:
35:     int pop( STACK_T *stack )
36:     {
37:       assert( stack );
38:       assert( stack->index > 0 );
39:
40:       return( stack->stack[—stack->index] );
41:     }
42:
43:
44:     void operator( STACK_T *stack, int op )
45:     {
46:       int a, b;
47:
48:       assert( stack );
49:       assert( stack->index > 0 );
50:
51:       a = pop(stack); b = pop(stack);
52:
53:       switch( op ) {
54:
55:         case OP_ADD:
56:           push( stack, (a+b) ); break;
57:
58:         case OP_SUBTRACT:
```

```
59:           push( stack, (a-b) ); break;
60:
61:         case OP_MULTIPLY:
62:           push( stack, (a*b) ); break;
63:
64:         case OP_DIVIDE:
65:           push( stack, (a/b) ); break;
66:
67:         default:
68:           assert(0); break;
69:
70:       }
71:
72:     }
73:
74:
75:     int main()
76:     {
77:       STACK_T stack;
78:
79:       initStack(&stack);
80:
81:       push( &stack, 2 );
82:       push( &stack, 5 );
83:       push( &stack, 2 );
84:       push( &stack, 3 );
85:       push( &stack, 5 );
86:       push( &stack, 3 );
87:       push( &stack, 6 );
88:
89:       operator( &stack, OP_ADD );
90:       operator( &stack, OP_SUBTRACT );
91:       operator( &stack, OP_MULTIPLY );
92:       operator( &stack, OP_DIVIDE );
93:       operator( &stack, OP_ADD );
94:       operator( &stack, OP_SUBTRACT );
95:
96:       printf( "Result is %d\n", pop( &stack ) );
97:       return 0;
98:     }
```

You compile your source with the `-g` flag to include debugging information for GDB, as in the following:

```
# gcc -g -Wall -o testapp testapp.c
#
```

Starting GDB

To debug a program with GDB, you simply execute GDB with your program name as the first argument. You can also start GDB and then load your program using the `load` command. In this example you start GDB with your application:

```
# gdb testapp
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are welcome to change it and/or distribute
copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"
for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb)
```

The `(gdb)` is the regular prompt for GDB and indicates that it is available for commands. You can start your application using the `run` command at this point, but the next sections look at a few other commands first.

Looking at Source

After you start GDB, your application is not yet running, but instead is just loaded into GDB. Using the `list` command, you can view the source of your application, as demonstrated in the following:

```
(gdb) list
70          }
71
72        }
73
74
75        int main()
76        {
77          STACK_T stack;
78
```

```
79          initStack(&stack);
    (gdb)
```

The `main` is our entry point, so `list` here shows this entry. You can also specify the lines of interest with `list` as follows:

```
    (gdb) list 75,85
75        int main()
76        {
77          STACK_T stack;
78
79          initStack(&stack);
80
81          push( &stack, 2 );
82          push( &stack, 5 );
83          push( &stack, 2 );
84          push( &stack, 3 );
85          push( &stack, 5 );
    (gdb)
```

Using the `list` command with no arguments always lists the source with the current line centered in the list.

Using Breakpoints

The primary strategy for debugging with GDB is the use of breakpoints to stop the running program and allow inspection of the internal data. A breakpoint can be set in a variety of ways, but the most common is by specifying a function name. Here, you tell GDB to break at your `main` program:

```
(gdb) break main
Breakpoint 1 at 0x804855b: file testapp.c, line 79.
(gdb) run
Starting program: /home/mtj/gnulinux/ch30/testapp

Breakpoint 1, main () at testapp.c:79
79        initStack(&stack);
(gdb)
```

After you give the `break` command, GDB tells you your breakpoint number (because you can set multiple) and the address, filename, and line number of the breakpoint. You then start your application using the `run` command, which results in hitting your previously set breakpoint. After the breakpoint is hit, GDB shows

the line that is to be executed next. Note that this statement, line 79, is the first executable statement of your application.

Recall that in all C applications, the `main` *function is the user entry point to the application, but various other work goes on behind the scenes to start and end the program. Therefore, when you break at the* `main` *function, you break at your user entry point, but not the true start of the application.*

You can view the available breakpoints using the `info` command:

```
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x0804855b in main at testapp.c:79
        breakpoint already hit 1 time
(gdb)
```

You see your single breakpoint and an indication from GDB that this breakpoint has been hit.

If your breakpoint is now of no use, you remove it using the `clear` command:

```
(gdb) clear 79
Deleted breakpoint 1
(gdb)
```

Other methods for setting breakpoints are shown in Table 30.1.

TABLE 30.1 Available Methods for Setting Breakpoints

Command	Breakpoint Method
`break function`	Set a breakpoint at a function.
`break file:function`	Set a breakpoint at a function (named file).
`break line`	Set a breakpoint at a line number.
`break file:line`	Set a breakpoint at a line number (named file).
`break address`	Set a breakpoint at a physical address.

One final interesting breakpoint method is the conditional breakpoint. Consider the following command:

```
(gdb) break operator if op = 2
Breakpoint 2 at 0x8048445: file testapp.c, line 48.
```

This tells GDB to break at the `operator` function if the `op` argument is equal to `2` (`OP_MULTIPLY`). This can be very useful if you're looking for a given condition and can save your having to break at each call and check the variable.

Stepping Through the Source

When you last left your debugging session, you had hit a breakpoint on your `main` function. Now it's time to step forward through the source. You have a few different possibilities, depending upon what you want to achieve (Table 30.2 lists these). To execute the next line of code, you can use the `step` command. This also steps into a function (if a function call is the next line to execute). If you prefer to step over a function, you can use the `next` command, which executes the next line and, if it's a function, simply performs it and sets the next line to execute to the line after the function. The `cont` command (short for continue) simply starts the program running.

TABLE 30.2 Methods for Stepping Through the Source

Command (Shortcut)	Operation
`next` (`n`)	Execute next line, step over functions.
`step` (`s`)	Execute next line, step into functions.
`cont` (`c`)	Continue execution.

You can also provide a count after the `next` and `step` commands, which performs the command the number of times specified as the argument. For example, issuing the command `step 5` performs the `step` command five times.

You can see the `next` and `step` commands within your debugging session as follows:

```
Breakpoint 1, main () at testapp.c:79
79        initStack(&stack);
(gdb) s
initStack (stack=0xbfffde60) at testapp.c:20
20        assert( stack );
```

```
(gdb) s
21        stack->index = 0;
(gdb) s
22      }
(gdb) s
main () at testapp.c:81
81        push( &stack, 2 );
(gdb) n
82        push( &stack, 5 );
(gdb)
```

In this last debugging fragment, you step into the `initStack` function. GDB then lets you know where you are (the function name and stack address). You step through the lines of `initStack`, and upon returning, GDB lets you know again that you are back in the `main` function. You then use the `next` command to perform the `push` function with a value of 2.

Inspecting Data

GDB makes it easy to inspect the data within a running program. Continuing with your debugging session, it's now time to look at your stack structure. You do this with the `display` command:

```
(gdb) display stack
1: stack = {stack = {2, 0, 1107383313, 134513378,
    1108545272, 1108544020, -1073750392, 134513265,
    1108544020, 1073792624}, index = 1}
(gdb)
```

If you simply display the `stack` variable, you see the aggregate components of the structure (first the array itself, then the `index` variable). Note that many of the stack elements are unusually large numbers, but this is only because the structure is not initialized. You can inspect specific elements of the `stack` variable, also using the `display` command:

```
(gdb) display stack.index
2: stack.index = 1
(gdb)
```

If you are dealing with an object reference (a pointer to the structure), you can deal with it as you would in C. For example, in this next example, you step into the `push` function to illustrate dealing with an object reference:

```
(gdb) s
push (stack=0xbffffae0, elem=5) at testapp.c:27
27        assert( stack );
(gdb) display stack->index
3: stack->index = 1
(gdb) display stack->stack[0]
4: stack->stack[0] = 2
(gdb)
```

One important consideration is the issue of static data. Static data names might be used numerous times in an application (bad coding policy, but it happens). To display a specific instance of static data, you can reference both the variable and file, such as `display 'file2.c'::variable`.

The `print` command (or its shortcut, `p`) can also be used to display data.

Changing Data

It's also possible to change the data in an operating program. You use the `set` command to change data, illustrated as follows:

```
(gdb) set stack->stack[9] = 999
(gdb) p *stack
$11 = {stack = {2, 0, 1107383313, 134513378,
    1108545272, 1108544020, -1073743096, 134513265,
    1108544020, 999}, index = 1}
(gdb)
```

Here you see that you have modified the last element of your stack array and then printed it back out to monitor the change.

Examining the Stack

The `backtrace` command (or `bt` for short) can be used to inspect the stack. This can tell you the current active function trace and the parameters passed. You are currently in the `push` function in the debugging session; take a look at the stack backtrace:

```
(gdb) bt
#0  push (stack=0xbffffae0, elem=5) at testapp.c:27
#1  0x08048589 in main () at testapp.c:82
#2  0x42015504 in __libc_start_main () from /lib/tls/libc.so.6
(gdb)
```

At the top is the current stack frame. You are in the `push` function, with a stack reference and an element of 5. The second frame is the function that called `push`, in this case, the `main` function. Note here that `main` was called by a function `__libc_start_main`. This function provides the initialization for `glibc`.

Stopping the Program

It's also possible to stop a debugging session using Ctrl+C. If the program is stopped in a function for which no debugging information is available (it wasn't compiled with `-g`), then only assembly is displayed (because source debugging information is not available).

Other GDB Debugging Topics

This section touches on some other topics of GDB, such as multiprocess application debugging and post-mortem debugging.

Multiprocess Application Debugging

One problem with the debugging of multiprocess applications is which process to follow when a new process is created. Recall from Chapter 13, "Introduction to Sockets Programming," that the `fork` function returns to both the parent and child processes. You can tell GDB which to return to `follow` using the `follow-fork-mode` command. For example, if you want to debug the child process, you specify to `follow` the child process as follows:

```
set follow-fork-mode child
```

Or, if you instead want to follow the parent (the default mode), you specify this as follows:

```
set follow-fork-mode parent
```

In either case, when GDB follows one process, the other process (child or parent) continues to run unimpeded. You can also tell GDB to ask you which process to follow when a `fork` occurs, as follows:

```
set follow-fork-mode ask
```

When the `fork` occurs, GDB asks which to follow. Whichever is not followed executes normally.

Multithreaded Application Debugging

There's no other way to put it: debugging multithreaded applications is difficult at best. GDB offers some capabilities that assist in multithreaded debugging, and this section looks at those.

The breakpoint is one of the most important aspects of debugging, but its behavior is different in multithreaded applications. If a breakpoint is created at a source line used by multiple threads, then every thread is affected by the breakpoint. You can limit this by specifying the thread to be affected. For example:

```
(gdb) break pos.c:17 thread 5
```

This installs a breakpoint at line 20 in `myfile.c`, but only for thread number 5. You can further refine these breakpoints using thread qualifiers. For example:

```
11      void *posThread( void *arg )
12      {
13        int ret;
14
15        ret = checkPosition( arg );
16
17        if (ret == 0) {
18
19          ret = move( arg );
20        }
21    (gdb) b pos.c:17 thread 5 if ret > 0
      Breakpoint 1 at 0x8048550: file pos.c, line 19
      (gdb)
```

In this example, you specify to break at line 17 in file `pos.c` for thread 5, but here you qualify that thread 5 is stopped only if the local `ret` variable is greater than 0.

You can identify the threads that are currently active in a multithreaded application using the `info threads` command. This command lists each of the active threads and its current state. For example:

```
(gdb) info threads
  5 Thread -161539152 (LWP 2819)  posThread (arg=0x0) at pos.c:17
...
* 1 Thread -151046720 (LWP 2808)  init at init.c:154
(gdb)
```

The * before thread 1 identifies that it is the current focus of the debugger. You can switch to any thread using the `thread` command, which allows you to change the focus of the debugger to the specified thread.

```
(gdb) thread 1
[Switching to thread 1 (Thread -161539152 (LWP 2819))]#0  posThread
17    if (ret == 0) {
(gdb)
```

As you step through a multithreaded program, you will find that the focus of the debugger can change at any step. This can be annoying, especially when the current thread is what you are interested in debugging. You can instruct GDB not to preempt the current thread by locking the scheduler. For example:

```
(gdb) set scheduler-locking on
```

This tells GDB not to preempt the current thread. When you want to allow other threads to preempt your current thread, you can set the mode to off:

```
(gdb) set scheduler-locking off
```

Finally, you can identify the current mode using the `show` command:

```
(gdb) show scheduler-locking
Mode for locking scheduler during execution is "on".
(gdb)
```

One final important command for thread debugging is the ability to apply a single command to all threads within an application. The `thread apply all` command is used for this purpose. For example, the following command emits a stack backtrace for every active thread:

```
(gdb) thread apply all backtrace
```

The `thread apply` command can also apply to a list of threads instead of all threads, as illustrated in the following:

```
(gdb) thread apply 1 4 9 backtrace
```

This performs a stack backtrace on threads 1, 4, and 9.

Debugging an Existing Process

You can debug an application that is currently running by attaching GDB to the process. All that you need is the process identifier for the process to debug. In this example, you have started the application in one terminal and then started GDB in another. After GDB has started, you issue the `attach` command to attach to the process. This suspends the process, allowing you to control it.

```
$ gdb
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
...
This GDB was configured as "i386-redhat-linux-gnu".
(gdb) attach 23558
Attaching to process 23558
Reading symbols from /home/mtj/gnulinux/ch30/testapp...done.
Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0x08048468 in operator (stack=0xbfffe9e0, op=1) at testapp.c:51
51        a = pop(stack); b = pop(stack);
(gdb) bt
#0  0x08048468 in operator (stack=0xbfffe9e0, op=1) at testapp.c:51
#1  0x080485cc in main () at testapp.c:93
#2  0x42015504 in __libc_start_main () from /lib/tls/libc.so.6
(gdb)
```

This method is very useful for dealing with "hung" programs where the fault occurs only after some period of time, or for dealing with unexpected hangs in production environments.

GDB starts by loading the symbols for the process and then identifying where the process was suspended (in the `operator` function). You issue the `bt` command to list the backtrace, which tells you which particular invocation of `operator` you are in (in this case, an `OP_SUBTRACT` call). Finally, if you are done debugging, you can release the process to continue by detaching from it using the `detach` call:

```
(gdb) detach
Detaching from program: /home/mtj/gnulinux/ch30/testapp,
process 23558
(gdb) quit
$
```

After the `detach` command has finished, the process continues normally.

Post-Mortem Debugging

When an application aborts and dumps a resulting core dump file, GDB can be used to identify what happened. Your application has been hardened, but you can remove a couple of asserts in the `push` function to force a core dump.

To enable GNU/Linux to generate a core dump, the command `ulimit -c unlimited` *should be performed. Otherwise, with limits in place, core dump files are not generated.*

You execute our application to get the core dump:

```
$ ./testapp
Segmentation fault (core dumped)
$ ls
core.23730  testapp  testapp.c
```

Now that you have your core dump, you can use GDB to identify where things went wrong. In the following example, you specify the executable application and the core dump image to GDB. It loads the app and uses the core dump file to identify what happened at the time of failure. After all the symbols are loaded, you see that the function failure occurred at `push` (but you already knew that). What's most important is that you see someone called `push` with a stack argument of 0 (null pointer). You might have caught this with our `assert` function, but it was conveniently removed for the sake of demonstration.

Further down, you see that the offending call was made at `testapp` line 30. This happens to be a call that you added to force the creation of this core file.

```
# gdb testapp core.23730
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
...
Core was generated by `./testapp'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x0804839c in push (stack=0x0, elem=2) at testapp.c:30

30        stack->stack[stack->index++] = elem;
```

```
(gdb) bt
#0  0x0804839c in push (stack=0x0, elem=2) at testapp.c:30
#1  0x08048536 in main () at testapp.c:81
#2  0x42015504 in __libc_start_main () from /lib/tls/libc.so.6
(gdb)
```

Although that was a quick review, it covers many of the necessary features that are needed for debugging with GDB.

Summary

A source-level debugger such as GDB is a necessary tool for developing applications of any size. This quick review of GDB introduced compiling for GDB debugging and many of the most useful commands. Other topics such as multiprocess application debugging and post-mortem debugging were also discussed.

Resources

GDB: The GNU Project Debugger website at http://www.gnu.org/software/gdb/.

31 Code Hardening

In This Chapter

- An Introduction to Code Hardening
- Code Hardening Techniques
- Tools Support for Code Hardening
- Tracing Binary Applications

Introduction

The practice of code hardening (or defensive programming) is a useful technique to increase the quality and reliability of software. The practice entails anticipating where errors can occur in our code and then writing that code in a way that either avoids them altogether or identifies them immediately so that their source can be more easily tracked. Because C is not a safe language, some methods have proven invaluable to help build more reliable programs, and this chapter details them.

This chapter covers a number of techniques under the umbrella of code hardening, all of which can be applied immediately. Because the benefits are clear, you can jump right into this chapter and look at a variety of code-hardening methods, as well as tool-based techniques such as using the compiler or open source tools to help build secure and reliable GNU/Linux applications.

Code Hardening Techniques

Code hardening can take a number of different forms, and entire books have been written on the topic. This section looks at a variety of techniques that can help build better code.

Return Values

The failure to check return values is one of the most common mistakes made in modern software. Many applications call user or system functions and are very optimistic about their successful operation. When building hardened software, you should make all reasonable attempts to check return values, and if you find failures, deal with them appropriately. *Reasonable attempts* is a key here; consider the following bogus example:

```
ret = printf( "Current mode is %d\n", mode );
if ( ret < 0 ) {
  ret = printf( "An error occurred emitting mode.\n" );
}
```

The point is easily illustrated, but in most cases (of user and system calls) the return value is relevant and should be checked in every case.

Strongly Consider User/Network I/O

Whenever you develop applications that take input either from a user or from the network (such as a Sockets application), it's even more critical to scrutinize the incoming data. Errors such as insufficient data for a given operation or more data received than buffer space is available for are two of the most common.

Use Safe String Functions

A number of standard C library functions suffer from security problems. The problem they present is that they have no bounds checking, which means that they can be exploited (this chapter discusses the buffer overflow issue shortly). The simple solution to this problem is to avoid unsafe functions and instead use the safe versions (as shown in Table 31.1).

TABLE 31.1 Safe Replacements for C Library Functions

Unsafe Function	Safe Replacement	Header
gets	fgets	stdio.h
sprintf	snprintf	stdio.h
strcat	strncat	string.h
strcpy	strncpy	string.h
strcmp	strncmp	string.h
strcasecmp	strncasecmp	strings.h
vsprintf	vsnprintf	stdio.h

Buffer Overflow

Buffer overruns cause unpredictable software behavior in the best case and security exploits in the worst. Buffer overruns can be avoided very simply. Consider the following erroneous example:

```
static char myArray[10];
...
int i;
for ( i = 0 ; i < 10 ; i++ ) {
  myArray[i] = (char)(0x30+i);
}
myArray[i] = 0;    // <—Overrun
```

In this example, you have overrun the bounds of your array by writing to the 11th element. Whatever object follows this array is now corrupted. You actually have a very simple solution to this problem, and it involves a better programming practice using symbolic constants. In the next example, you create a constant defining the size of your array, but then you add one more element for the trailing NULL.

```
#define ARRAY_SIZE    10
static char myArray[ARRAY_SIZE+1];
...
int i;
for (i = 0 ; i < ARRAY_SIZE ; i++ ) {
  myArray[i] = (char)(0x30+i);
}
myArray[ARRAY_SIZE] = 0;
```

You've automatically protected the array by an extra element at the end, but also—in good programming practice—you've used a symbol to denote the size of the array, rather than relying on a number.

Provide Logical Alternatives at Decision Points

A very common mistake that can yield unpredictable results is the absence of a `default` section in a `switch` statement. Consider the following example:

```
switch( mode ) {
  case OPERATIONAL_MODE:
    /* switch to operational mode processing */
    break;
  case BUILT_IN_TEST_MODE:
    /* switch to test processing */
    break;
}
```

Here, in the event, another mode was added, but this particular code segment was not updated; the result after this segment has executed is unpredictable. The solution is to *always* include a `default` section that either asserts (in debugging mode) or at a minimum notifies the caller that a problem has occurred. If you're really not expecting another mode, you can simply call `assert` here to catch the condition during debugging:

```
switch( mode ) {
  case OPERATIONAL_MODE:
    /* switch to operational mode processing */
    break;
  case BUILT_IN_TEST_MODE:
    /* switch to test processing */
    break;
  default:
    assert(0);
    break;
}
```

A similar problem exists with `if/then/else` chains. The following example illustrates the problem:

```
float multiplier = 0.0;
if (state == FIRST_STAGE) multiplier = 0.75;
else if (state == SECOND_STAGE) multiplier = 1.25;
```

If your state is corrupted or takes on a value that you did not expect, then your multiplier takes on the value of 0.0, and the result is unpredictable at best and, depending upon the application, catastrophic at worst. At a minimum you should provide an `else` to catch the issue, such as seen here:

```
float multiplier = 0.0;
if (state == FIRST_STAGE) multiplier = 0.75;
else if (state == SECOND_STAGE) multiplier = 1.25;
else multiplier = SAFE_MULTIPLIER;
```

In many cases, the trailing `else` isn't necessary, but whenever one is seen, you should give it extra scrutiny to avoid erroneous results.

Self-Identifying Structures

A self-identifying structure is a method that mimics the concept of runtime type checking present in strongly typed languages. In a strongly typed language, the use of an invalid type results in a runtime error. Consider the passing of pointers in a weakly typed language such as C. With C typecasting, it's not difficult to confuse one structure for another.

With a simple policy change to C structures and a limited amount of checking, you can help ensure that functions are dealing with the right types. Consider the C source shown in Listing 31.1. At lines 6–12, you see your target structure, which contains a special header called a *signature* (sometimes called a *runtime type identifier*). The type is shown at line 4, in this case simply a signature that uniquely represents your structure. You then provide two macro functions that initialize (`INIT_TARGET_MARKER`) and then check (`CHECK_TARGET_MARKER`) the signature in the structure.

Skip ahead a little and take a look at the `main` function at lines 34–54. You allocate two objects (both of size `targetMarket_t`) and then initialize one of them as an actual target marker using the `INIT_TARGET_MARKER` macro. Finally, you try to display each of the objects by passing each to the `displayTarget` function.

In your `displayTarget` function (lines 22–31), your first task is to check the signature of the received object by calling `CHECK_TARGET_MARKER`. If the signature is not correct, you call `assert` rather than risk providing bogus information. Granted, in a production system you can probably handle this better, but this illustrates the concept.

LISTING 31.1 Illustrating a Self-Identifying Structure (on the CD-ROM at `./source/ch31/selfident.c`)

```
 1:    #include <stdio.h>
 2:    #include <assert.h>
 3:
 4:    #define TARGET_MARKER_SIG       0xFAF32000
 5:
 6:    typedef struct {
 7:
 8:      unsigned int signature;
 9:      unsigned int targetType;
10:      double       x, y, z;
11:
12:    } targetMarker_t;
13:
14:
15:    #define INIT_TARGET_MARKER(ptr) \
16:        (((targetMarker_t *)ptr)->signature = TARGET_MARKER_SIG)
17:    #define CHECK_TARGET_MARKER(ptr) \
18:        assert(((targetMarker_t *)ptr)->signature == \
19:                        TARGET_MARKER_SIG)
20:
21:
22:    void displayTarget( targetMarker_t *target )
23:    {
24:
25:      /* Pre-check of the target structure */
26:      CHECK_TARGET_MARKER(target);
27:
28:      printf( "Target type is %d\n", target->targetType );
29:
30:      return;
31:    }
32:
33:
34:    int main()
35:    {
36:      void *object1, *object2;
37:
38:      /* Create two objects */
39:      object1 = (void *)malloc( sizeof(targetMarker_t) );
40:      assert(object1);
41:      object2 = (void *)malloc( sizeof(targetMarker_t) );
```

```
42:       assert(object2);
44:       /* Init object1 as a target marker struct */
45:       INIT_TARGET_MARKER(object1);
46:
47:       /* Try to display object1 */
48:       displayTarget( (targetMarker_t *)object1 );
49:
50:       /* Try to display object2 */
51:       displayTarget( (targetMarker_t *)object2 );
52:
53:       return 0;
54:     }
```

Reporting Errors

The reporting of errors is an interesting topic because the policy that's chosen can be very different depending upon the type of application you're developing. For example, if you are writing a command-line utility, emitting error messages to `stderr` is a common method to communicate errors to the user. But what happens if you're building an application that has I/O capabilities, such as an embedded Linux application? You have a number of possibilities, including the generation of a specialized log or use of the standard system log (`syslog`). The `syslog` function has the prototype:

```
#include <syslog.h>
void syslog( int priority, char *format, ... );
```

To the `syslog` function, you provide a priority, a format string, and some arguments (similar to `printf`). The priority can be one of `LOG_EMERG`, `LOG_ALERT`, `LOG_CRIT`, `LOG_ERR`, `LOG_WARNING`, `LOG_NOTICE`, `LOG_INFO`, or `LOG_DEBUG`. An example of using `syslog` to generate a message to the system log is shown in Listing 31.2.

LISTING 31.2 Simple Example of `syslog` Use (on the CD-ROM at `./source/ch31/simpsyslog.c`)

```
1:    #include <syslog.h>
2:
3:    int main()
4:    {
5:
6:      syslog( LOG_ERR, "Unable to load configuration!" );
7:
8:      return 0;
9:    }
```

This results in your system log (stored within your filesystem at `/var/log/messages`) being updated as follows:

```
Jul 21 18:13:10 camus sltest: Unable to load configuration!
```

In this example, your application in Listing 31.2 was called `sltest`, with the hostname of `camus`. The system log can be especially useful because it's an aggregate of many error reports. This allows a developer to see where a message was generated in relation to others, which can be very useful in helping to understand problems.

The `syslog` *is very useful for communicating information for system applications and daemons.*

One final topic on error reporting is that of being specific about the error being reported. The error message must uniquely identify the error for the user to be able to deal with it reasonably.

Reducing Complexity Also Reduces Potential Bugs

Code that is of higher complexity potentially contains more bugs. It's a fact of life, but one that you can use to help reduce defects. In some disciplines this is called *refactoring*, but the general goal is to take a complex piece of software and break it up so that it's more easily understood. This very act can lead to higher quality software that is more easily maintained.

Self-Protective Functions

Writing self-protective functions can be a very useful debugging mechanism to ensure that your software is correct. The programming language Eiffel includes language features to provide this mechanism (known as *programming by contract*).

Being self protective means that when you write a function, you scrutinize the input to the function and, upon completion of its processing, scrutinize the output to ensure that what you've done is correct.

Take a look at an example of a simple function that illustrates this behavior (see Listing 31.3).

If an expression results in false (0), the `assert` function causes the application to fail and an error to be generated to `stdout`. To disable `asserts` within an application, the `NDEBUG` symbol can be defined, which causes the `assert` calls to be optimized away.

LISTING 31.3 Example of a Self-Protective Function (on the CD-ROM at `./source/ch31/selfprot.c`)

```
 1:    STATUS_T checkAntennaStatus( ANTENNA_T antenna, MODE_T *mode )
 2:    {
 3:      ANTENNA_STS_T retStatus;
 4:
 5:      /* Validate the input */
 6:      assert( validAntenna( antenna ) );
 7:      assert( validMode( mode ) );
 8:
 9:
10:      /*————————————————————*/
11:      /* Internal checkAntennaStatus processing */
12:      /*————————————————————*/
13:
14:
15:      /* We may have changed modes, check it. */
16:      assert( validMode( mode ) );
17:
18:      return retStatus;
19:    }
```

In Listing 31.3 you see a function that first ensures that it's getting good data (validating input) and then that what it's providing is correct (checking output). You also can return errors upon finding these conditions, but for this example, you are mandating proper behavior at all levels. If all functions performed this activity, finding the real source of bugs would be a snap.

The use of `assert` isn't restricted just to ensuring that function inputs and outputs are correct. It can also be used for internal consistency. Any critical failure that should be identified during debugging is easily handled with `assert`.

Using the `assert` call for internal consistency is often the only practical way to find timing (race condition) bugs in threaded code.

Maximize Debug Output

Too much output can disguise errors; too little and an error can be missed. The right balance must be found when emitting debug and error output to ensure that only the necessary information is presented to avoid overloading an already overloaded user.

Memory Debugging

You have many libraries available that support debugging dynamic memory management on GNU/Linux. One of the most popular is called Electric Fence, which programs the underlying processor's MMU (memory management unit) to catch memory errors via segment faults. Electric Fence can also detect exceeding array bounds. The Electric Fence library is very powerful and identifies memory errors immediately.

Compiler Support

The compiler itself can be an invaluable tool to identify issues in your code. When you build software, you should always enable warnings using the `-Wall` flag. To further ensure that warnings aren't missed in large applications, you can enable the `-Werror` flag, which treats warnings as errors and therefore halts further compilation of a source file. When building an application that has many source files, this combination can be beneficial. This is demonstrated as follows:

```
gcc -Wall -Werror test.c -o test
```

If you want your source to have ANSI compatibility, you can enable checking for ANSI compliance (with pedantic checking) as follows:

```
gcc -ansi -pedantic test.c -o test
```

Identifying uninitialized variables is a very useful test, but in addition to the warning option, optimization must also be enabled, because the data flow information is available only when the code is optimized:

```
gcc -Wall -O -Wuninitialized test.c -o test
```

Chapter 5, "The GNU Compiler Toolchain," provides additional warning information. The `gcc` main page also contains numerous warning options about those enabled via `-Wall`.

Source Checking Tools

To identify security vulnerabilities as well as common programming mistakes, source-checking tools should be part of the development process. In addition to being simple to use, they can easily be automated as part of the build process. One important note when using source-checking tools is that while they can identify

flaws, they can also miss them. Therefore, use your best judgment when using the tools, and always know your source.

The `splint` tool (short for *secure programming lint*) is a static source-checking tool built by the Inexpensive Program Analysis group at the University of Virginia. It provides strong and weak checking of source and, with annotation, can perform a very complete analysis of source.

With unannotated source, you can use the `-weak` option (with header files found in the `./inc` subdirectory):

```
splint -weak *.c -I./inc
```

`splint` also supports modes for standard checking (`-standard`, the default mode), moderate checking (`-checks`), and extremely strict checking (`-strict`).

The `flawfinder` tool (developed by David Wheeler) is another useful tool that statically checks source in search of errors. `flawfinder` provides useful error messages that can be tutorial in nature. Consider the following example:

```
$ flawfinder test.c
test.c:11:  [2] (buffer) char:
  Statically-sized arrays can be overflowed. Perform bounds
  checking, use functions that limit length, or ensure that
  the size is larger than the maximum possible length.
$
```

In this case, an array is found that does not necessarily present a security issue, but a gentle reminder is provided concerning the potential for exploitation.

Many other source-checking tools exist, such as `RATS` (Rough Auditing Tool for Security) and `ITS4` (static vulnerability scanner). URLs for these tools can be found in the "Resources" section of this chapter.

Code Tracing

One final useful topic is that of system call tracing. While not specifically a source auditing tool, it can be a very useful tool for understanding the underlying operation of a GNU/Linux application. The `strace` utility provides the capability to trace the execution of an application from the perspective of system calls (such as `fopen` or `fwrite`, to name just two).

Consider the application shown in Listing 31.4. This application violates many of the code hardening principles already discussed, but you can see how you can still debug it using `strace`.

LISTING 31.4 Poorly Hardened Application (on the CD-ROM at `./source/ch31/badprog.c`)

```
 1:    #include <unistd.h>
 2:    #include <fcntl.h>
 3:
 4:    #define MAX_BUF         128
 5:
 6:    int main()
 7:    {
 8:      int fd;
 9:      char buf[MAX_BUF+1];
10:
11:      fd = open( "myfile.txt", O_RDONLY );
12:
13:      read( fd, buf, MAX_BUF );
14:
15:      printf( "read %s\n", buf );
16:
17:      close( fd );
18:    }
```

The first thing to note about this application is that at line 11, where you attempt to open the file called `myfile.txt`, you have no checking to ensure that the file actually exists. Executing this application gives an unpredictable result:

```
$ gcc -o bad bad.c
$ ./bad
read @êÿ¿8Z@
$
```

This is surely not what you expected, so now you can use `strace` to see what's going on. The following shrinks the output a bit, because in this case you are not interested in the libraries that are loaded and such.

```
$ strace ./bad
execve("./bad", ["./bad"], [/* 20 vars */]) = 0
uname({sys="Linux", node="camus", ...}) = 0
...
open("myfile.txt", O_RDONLY)       = -1 ENOENT (No such file or
                                                directory)
read(-1, 0xbfffef20, 128)          = -1 EBADF (Bad file descriptor)
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
```

```
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0x40017000
write(1, "read \300\357\377\2778Z\1@\n", 14read Àïÿÿÿÿÿÿÿÿ8Z@
) = 14
close(-1)                          = -1 EBADF (Bad file descriptor)
munmap(0x40017000, 4096)           = 0
exit_group(-1)                     = ?
$
```

After executing the app, you can see that the `execve` system call is used to actually start the program. You then see an `open` shortly after execution, which matches your source (line 11, Listing 31.4). You can see at the right that the `open` system call returned –1, with an error of `ENOENT` (the file doesn't exist). This tells you right away what's going on with your application. The attempted `read` also fails, with the error of a bad file descriptor (because the `open` call failed).

The `strace` tool can be useful not only to understand the operation of your programs, but also the operation of programs for which you might not have source. From the perspective of system calls, you can at some level understand what binary applications are up to.

Summary

Code hardening can increase your development time, but it routinely reduces your debugging time. By anticipating faults while you design, you automatically increase the reliability and quality of your software, so this technique is one to be mastered. This chapter discussed a variety of code hardening techniques, as well as non-coding methods to help create better software and understand its operation.

Resources

Secure Programming for Linux and UNIX HOWTO at http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html.

Electric Fence malloc() Debugger at http://perens.com/FreeSoftware/.

Splint Source Checking Tool at http://www.splint.org/.

Flawfinder Source Checker at http://www.dwheeler.com/flawfinder/.

RATS Source Checker at http://www.fortifysoftware.com/security-resources/rats.jsp.

ITS4 Static Vulnerability Scanner at http://www.cigital.com/its4/.

This page intentionally left blank

32 Coverage Testing with GNU gcov

In This Chapter

- Understanding GNU's gcov Tool
- Exploring the Different Uses for gcov
- Building Software for gcov
- Understanding gcov's Various Data Products
- Illustrating Problems with gcov and Optimization

Introduction

This chapter explores the gcov utility and demonstrates how it can be used to both help test and support software profiling and optimization. You will learn how to build software for use with gcov and then understand the various types of data that are provided. Finally, the chapter investigates things to avoid when performing coverage testing.

What Is gcov?

The chapter begins with an overview of what gcov can do for you. The gcov utility is a coverage testing tool. When built with an application, the gcov utility monitors the application under execution and identifies which source lines have been executed and which have not. Further, gcov can identify the number of times a particular line has been executed, making it useful for performance profiling (where an application is spending most of its time). Because gcov can tell which lines have not

been executed, it is useful to determine which code is not covered in test. In concert with a test suite, gcov can identify whether all source lines have been adequately covered [FSF02].

This chapter discusses the use of gcov bundled with version 3.2.2 of the GNU compiler toolchain.

Preparing the Image

First take a look at how an image is prepared for use with gcov. You get more detail of gcov options in the coming sections, but this section serves as an introduction. For an example you can use the simple bubblesort source file shown in Listing 32.1.

LISTING 32.1 Sample Source File to Illustrate the gcov Utility (on the CD-ROM at ./source/ch32/bubblesort.c)

```
#include <stdio.h>

void bubbleSort( int list[], int size )
{
  int i, j, temp, swap = 1;

  while (swap) {

    swap = 0;

    for ( i = (size-1) ; i >= 0 ; i– ) {

      for ( j = 1 ; j <= i ; j++ ) {

        if ( list[j-1] > list[j] ) {

          temp = list[j-1];
          list[j-1] = list[j];
          list[j] = temp;
          swap = 1;

        }

      }

    }

```

```
28:       }
29:
30:     }
31:
32:     int main()
33:     {
34:       int theList[10]={10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
35:       int i;
36:
37:       /* Invoke the bubble sort algorithm */
38:       bubbleSort( theList, 10 );
39:
40:       /* Print out the final list */
41:       for (i = 0 ; i < 10 ; i++) {
42:         printf("%d\n", theList[i]);
43:       }
44:
45:     }
```

The gcov utility is used in conjunction with the compiler toolchain. This means that the image on which you want to perform coverage testing must be compiled with a special set of options. These are illustrated as follows for compiling the source file bubblesort.c:

```
gcc bubblesort.c -o bubblesort -ftest-coverage -fprofile-arcs
```

The resulting image, when executed, produces a number of files containing statistics about the application (along with statistics emitted to standard-out). These files are then used by the gcov utility to report statistics and coverage information to you. When the -ftest-coverage option is specified, two files are generated for each source file. These files use the extension .bb (basic-block) and .bbg (basic block graph) and help to reconstruct the program flow graph of the executed application. For the option -fprofile-arcs, a .da file is generated that contains the execution count for each instrument branch. These files are used after execution, along with the original source file, to identify the execution behavior of the source.

Using the gcov Utility

Now that you have the image, you can continue to walk through the rest of the process. Executing the new application yields the set of statistics files discussed previously (.bb, .bbg, and .da). You then execute the gcov application with the source file that you want to examine as follows:

```
$ ./bubblesort
...
$ gcov bubblesort.c
100.00% of 17 source lines executed in file bubblesort.c
Creating bubblesort.c.gcov.
```

This tells you that all source lines within your sample application were executed at least once. You can see the actual counts for each source line by reviewing the generated file `bubblesort.c.gcov` (see Listing 32.2).

LISTING 32.2 File `bubblesort.c.gcov` Resulting from Invocation of `gcov` Utility

```
 1:          #include <stdio.h>
 2:
 3:          void bubbleSort( int list[], int size )
 4:     1    {
 5:     1      int i, j, temp, swap = 1;
 6:
 7:     3      while (swap) {
 8:
 9:     2        swap = 0;
10:
11:     22        for ( i = (size-1) ; i >= 0 ; i— ) {
12:
13:     110          for ( j = 1 ; j <= i ; j++ ) {
14:
15:     90             if ( list[j-1] > list[j] ) {
16:
17:     45               temp = list[j-1];
18:     45               list[j-1] = list[j];
19:     45               list[j] = temp;
20:     45               swap = 1;
21:
22:                }
23:
24:              }
25:
26:            }
27:
28:          }
29:
30:        }
31:
32:        int main()
```

```
33:     1    {
34:     1      int theList[10]={10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
35:     1      int i;
36:
37:            /* Invoke the bubble sort algorithm */
38:     1      bubbleSort( theList, 10 );
39:
40:           /* Print out the final list */
41:     11      for (i = 0 ; i < 10 ; i++) {
42:     10        printf("%d\n", theList[i]);
43:             }
44:
45:           }
```

Now, you can take a look at some of the major points of Listing 32.2 to see what's provided. The first column shows the execution count for each line of source (line 4 shows a count of one execution, the call of the `bubbleSort` function). In some cases execution counts aren't provided. These are simply C source elements that don't result in code (for example, lines 22 through 30).

The counts can provide some information about the execution of the application. For example, the test at line 15 was executed 90 times, but the code executed within the test (lines 17–20) was executed only 45 times. This tells you that while the test was invoked 90 times, the test succeeded only 45. In other words, half of the tests resulted in a swap of two elements. This behavior is because of the ordering of the test data at line 34.

The `gcov` *files (*`.bb`*,* `.bbg`*, and* `.da`*) should be removed before running the application again. If the* `.da` *file isn't removed, the statistics simply accumulate rather than start over. This can be useful but, if unexpected, problematic.*

The code segment executed most often, not surprisingly, is the inner loop of the sort algorithm. This is because line 13 is invoked one time more than line 15 because of the exit test (to complete the loop).

Looking at Branch Probabilities

You can also see the branch statistics for the application using the `-b` option. This option writes branch frequencies and summaries for each branch in the instrumented application. For example, when you invoke `gcov` with the `-b` option, you now get the following:

```
$ gcov -b bubblesort.c
100.00% of 17 source lines executed in file bubblesort.c
100.00% of 12 branches executed in file bubblesort.c
100.00% of 12 branches taken at least once in file bubblesort.c
100.00% of 2 calls executed in file bubblesort.c
Creating bubblesort.c.gcov.
$
```

The resulting `bubblesort.c.gcov` file is shown in Listing 32.3. Here you see a similar Listing to 32.2, but this time the branch points have been labeled with their frequencies.

LISTING 32.3 File `bubblesort.c.gcov` Resulting from Invocation of `gcov` Utility with `-b`

```
 1:             #include <stdio.h>
 2:
 3:             void bubbleSort( int list[], int size )
 4:         1   {
 5:         1     int i, j, temp, swap = 1;
 6:
 7:         3     while (swap) {
 8:     branch 0 taken = 67%
 9:     branch 1 taken = 100%
10:
11:         2       swap = 0;
12:
13:        22       for ( i = (size-1) ; i >= 0 ; i— ) {
14:     branch 0 taken = 91%
15:     branch 1 taken = 100%
16:     branch 2 taken = 100%
17:
18:       110         for ( j = 1 ; j <= i ; j++ ) {
19:     branch 0 taken = 82%
20:     branch 1 taken = 100%
21:     branch 2 taken = 100%
22:
23:        90           if ( list[j-1] > list[j] ) {
24:     branch 0 taken = 50%
25:
26:        45             temp = list[j-1];
27:        45             list[j-1] = list[j];
28:        45             list[j] = temp;
29:        45             swap = 1;
30:
```

```
31:                     }
32:
33:                   }
34:
35:                }
36:
37:               }
38:
39:             }
40:
41:             int main()
42:         1   {
43:         1     int theList[10]={10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
44:         1     int i;
45:
46:               /* Invoke the bubble sort algorithm */
47:         1     bubbleSort( theList, 10 );
48:     call 0 returns = 100%
49:
50:               /* Print out the final list */
51:        11     for (i = 0 ; i < 10 ; i++) {
52:     branch 0 taken = 91%
53:     branch 1 taken = 100%
54:     branch 2 taken = 100%
55:        10       printf("%d\n", theList[i]);
56:     call 0 returns = 100%
57:               }
58:
59:             }
```

The branch points are very dependent upon the target architecture's instruction set. Line 23 is a simple `if` statement and therefore has one branch point represented. Note that this is `50%`, which cross-checks with your observation of line execution counts previously. Other branch points are a little more difficult to parse. For example, line 7 represents a `while` statement and has two branch points. In x86 assembly, this line compiles to what you see in Listing 32.4.

LISTING 32.4 x86 Assembly for the First Branch Point of `bubblesort.c.gcov`

```
1:  cmpl        $0, -20(%ebp)
2:  jne         .L4
3:  jmp         .L1
```

The swap variable is compared at line 1 to the value 0 in Listing 32.4. If it's not equal to zero, the jump at line 2 is taken (jump-nonzero) to .L4 (line 11 from Listing 32.3). Otherwise, the jump at line 3 is taken to .L1. The branch probabilities show that line 2 (branch 0) was taken 67 percent of the time. This is because the line was executed three times, but the jne (line 2 of Listing 32.3) was taken only twice (2/3 or 67 percent). When the jne at line 2 is not taken, you do the absolute jump (jmp) at line 3. This is executed once, and after it is executed, the application ends. Therefore, branch 1 (line 9 of Listing 32.3) is taken 100 percent of the time.

So the branch probabilities are useful in understanding program flow, but consulting the assembly can be required to understand what the branch points represent.

Incomplete Execution Coverage

When gcov encounters an application whose test coverage is not 100 percent, the lines that are not executed are labeled with ###### rather than an execution count. Listing 32.5 shows a source file created by gcov that illustrates less than 100 percent coverage.

LISTING 32.5 A Sample Program with Incomplete Test Coverage (on the CD-ROM at ./source/ch32/incomptest.c)

```
 1:             #include <stdio.h>
 2:
 3:             int main()
 4:        1    {
 5:        1      int a=1, b=2;
 6:
 7:        1      if (a == 1) {
 8:        1        printf("a = 1\n");
 9:               } else {
10:   ######         printf("a != 1\n");
11:               }
12:
13:        1      if (b == 1) {
14:   ######         printf("b = 1\n");
15:               } else {
16:        1        printf("b != 1\n");
17:               }
18:
19:        1      return 0;
20:             }
```

The gcov utility also reports this information to standard-out when it is run. It emits the number of source lines possible to execute (in this case 9) and the percentage that were actually executed (here, 78 percent):

```
$ gcov incomptest.c
 77.78% of 9 source lines executed in file incomptest.c
Creating incomptest.c.gcov.
$
```

If your sample application has multiple functions, you can see the breakdown per function through the use of the -f option (or -function-summaries). This is illustrated using your previous bubblesort application as follows:

```
$ gcov -f bubblesort.c
100.00% of 11 source lines executed in function bubbleSort
100.00% of 6 source lines executed in function main
100.00% of 17 source lines executed in file bubblesort.c
Creating bubblesort.c.gcov.
$
```

Options Available for gcov

Now that you have seen gcov in action in a few scenarios, it's time to look at gcov's full list of options (see Table 32.1). The gcov utility is invoked with the source file to be annotated as follows:

```
gcov [options] sourcefile
```

From Table 32.1, you can see a short single letter option and a longer option. The short option is useful when you are using gcov from the command line, but when gcov is part of a Makefile, you should use the longer options should because they're more descriptive.

To retrieve version information about the gcov utility, you use the -v option. Because gcov is tied to a given compiler toolchain (it's actually built from the gcc toolchain source), the versions for gcc and gcov are identical.

An introduction to gcov and the option help for gcov can be displayed using the -h option.

The branch probabilities can be emitted to the annotated source file using the -b option (see the section "Looking at Branch Probabilities" earlier in this chapter). Rather than producing branch percentages, you can emit branch counts using the -c option.

TABLE 32.1 `gcov` Utility Options

Option	Purpose
`-v, —version`	Emit version information (no further processing).
`-h, —help`	Emit help information (no further processing).
`-b, —branch-probabilities`	Emit branch frequencies to the output file (with summary).
`-c, —branch-counts`	Emit branch counts rather than frequencies.
`-n, —no-output`	Do not create the `gcov` output file.
`-l, —long-file-names`	Create long filenames.
`-f, —function-summaries`	Emit summaries for each function.
`-o, —object-directory`	Directory where `.bb`, `.bbg`, and `.da` files are stored.

If the annotated source file is not important, you can use the `-n` option. This can be useful if all that's important is to understand the test coverage of the source. This information is emitted to standard-out.

When you are including source in header files, it can be useful to use the `-l` option to produce long filenames. This helps make filenames unambiguous if multiple source files include headers containing source (each getting its own `gcov` annotated header file).

You can emit coverage information to standard-out for each function rather than the entire application using the `-f` option. This is discussed in the section "Incomplete Execution Coverage," earlier in this chapter.

The final option, `-o`, tells `gcov` where the `gcov` object files are stored. By default, `gcov` looks for the files in the current directory. If they're stored elsewhere, this option specifies where `gcov` can find them.

Considerations

Certain capabilities should be avoided when you are using `gcov` for test coverage. You should disable optimization when you are using `gcov`. Because optimization can result in source lines being moved or removed, coverage is less meaningful. Coverage testing is also less meaningful when you are using source macro expansion in the source after the preprocessor stage. These aren't shown in `gcov` and therefore miss identification of full test coverage.

If you are a GNU/Linux kernel developer, gcov can be used for certain architectures within the kernel. A patch is available from IBM to allow gcov use in the kernel. Its availability is provided in the "Resources" section.

Summary

This chapter introduced GNU's gcov test coverage tool. It explored the capabilities for gcov, including coverage testing, identifying branch probabilities, and emitting summaries for each function under review. It investigated building software for use with gcov and some considerations for options to avoid, such as optimization and source macro expansion.

References

[FSF02] "Using the GNU Compiler Collection (GCC)," Free Software Foundation at http://gcc.gnu.org/onlinedocs/gcc-3.2.3/gcc/index.html#toc_Gcov.

Resources

The LTP GCOV-kernel extension (GCOV-kernel) at http://ltp.sourceforge.net/coverage/.

This page intentionally left blank

33 Profiling with GNU `gprof`

In This Chapter

- An Introduction to Performance Profiling
- An Introduction to the GNU `gprof` Profiler Utility
- Preparing an Image for Use with `gprof`
- Discussing the Data Products Provided by `gprof`
- Exploring Some of the Most Important `gprof` Utility Options

Introduction

This chapter investigates the `gprof` utility and explores how it can be used to help build efficient programs. You'll learn how software must be built for use with `gprof` and then have a chance to understand the data products that are provided. Finally, the chapter investigates the variety of options that `gprof` provides and how they can be used.

What Is Profiling?

Profiling is the art of analyzing the performance of an application. By identifying where a program spends the majority of its time, you can better isolate where your modifications can yield the biggest performance gains. The most common result of profiling is a better understanding of where a given program spends its time. By looking at where the program spends the majority of its time, you can yield significant gains by improving that portion of code, rather than fine-tuning code that doesn't affect the bottom line.

What Is gprof?

The gprof utility is the GNU profiler, a tool that identifies how much time is spent in a function of an operating program. The GNU profiler also identifies which functions were called by a given function. Similar to the gcov utility (the topic of Chapter 32, "Coverage Testing with GNU gcov"), the compiler introduces profiling code into the target image, which generates a statistics file upon execution. This file (gmon.out) contains histogram records, call-graph arc records, and basic-block execution records that illustrate the execution profile of an application. When read by the gprof utility, the performance behavior of the application can be readily understood.

This chapter discusses use of the gprof utility bundled with version 3.2.2 of the GNU compiler toolchain.

Preparing the Image

Now it's time to look at how an image is prepared for profiling with gprof. You first look at some basic uses of profiling with gprof, and in later sections you look at some of the other options available. For the profiling example, you use the following sorting demo shown in Listing 33.1. This sample source (sort.c) illustrates two sorting algorithms, the insert-sort (function insertSort, lines 5–21) and the bubble-sort (function bubbleSort, lines 23–50). Each is run with identical data to understand unambiguously their profiling properties for a given data set (as provided by function init_list, lines 53–62).

LISTING 33.1 Sample Source to Explore the gprof Utility (on the CD-ROM at ./source/ch33/sort.c)

```
 1:    #include <stdio.h>
 2:
 3:    #define MAX_ELEMENTS 10000
 4:
 5:    void insertSort( int list[], int size )
 6:    {
 7:      int i, j, temp;
 8:
 9:      for ( i = 1 ; i <= size-1 ; i++ ) {
10:
11:        temp = list[i];
12:
13:        for ( j = i-1 ; j >= 0 && (list[j] > temp) ; j— ) {
```

```
14:           list[j+1] = list[j];
15:         }
16:
17:         list[j+1] = temp;
18:
19:       }
20:
21:     }
22:
23:     void bubbleSort( int list[], int size )
24:     {
25:       int i, j, temp, swap = 1;
26:
27:       while (swap) {
28:
29:         swap = 0;
30:
31:         for ( i = (size-1) ; i >= 0 ; i— ) {
32:
33:           for ( j = 1 ; j <= i ; j++ ) {
34:
35:             if ( list[j-1] > list[j] ) {
36:
37:               temp = list[j-1];
38:               list[j-1] = list[j];
39:               list[j] = temp;
40:               swap = 1;
41:
42:             }
43:
44:           }
45:
46:         }
47:
48:       }
49:
50:     }
51:
52:
53:     void init_list( int list[], int size )
54:     {
55:       int i;
56:
57:       for ( i = 0 ; i < size ; i++ ) {
```

```
58:         list[i] = (size-i);
59:       }
60:
61:       return;
62:     }
63:
64:
65:     int main()
66:     {
67:       int list[MAX_ELEMENTS];  int i;
68:
69:       /* Invoke the bubble sort algorithm */
70:       init_list( list, MAX_ELEMENTS );
71:       bubbleSort( list, MAX_ELEMENTS );
72:       init_list( list, MAX_ELEMENTS );
73:       insertSort( list, MAX_ELEMENTS );
74:
75:     }
```

The `gprof` utility uses information from the executable image and the profiler output file, `gmon.out`, to generate its profiling data. To collect the profiling data, the image must be compiled and linked with a special set of compiler flags. These are illustrated in the following for compiling the sample source file, `sort.c`:

```
gcc sort.c -o sort -pg
```

The result is an image, `sort`, which is instrumented to collect profiling information. When the image is executed and completes normally, a file called `gmon.out` results, containing the profiling data.

The `gmon.out` *file is written upon normal exit of the application. If the program exits abnormally or the user forces an exit with a Ctrl+C, the* `gmon.out` *file is not written.*

Using the gprof Utility

Upon execution of the profiler-instrumented image, the `gmon.out` file is generated. This file is used in conjunction with the original image for `gprof` to generate human-readable statistics information. Take a look at a simple example and the data products that result. First, you invoke the image and then generate the `gprof` summary:

```
$ ./sort
$ gprof sort gmon.out > sort.gprof
```

The gprof utility writes its human-readable output to standard-out, so the user must redirect this to a file to save it. The first element of the gprof output is what's called the "flat profile." This provides the basic timing summary of the executable, as shown in Listing 33.2. Note that this is not the complete output of gprof. You can take a look at other data products shortly.

LISTING 33.2 Sample Flat Profile Output from gprof

```
$ gprof sort gmon.out | more
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 71.66      3.11     3.11        1     3.11     3.11  bubbleSort
 28.11      4.33     1.22        1     1.22     1.22  insertSort
  0.23      4.34     0.01        2     0.01     0.01  init_list
...
```

As you saw in Listing 33.1, your application is made up of three functions. Each function is represented here with a variety of timing data. The first column represents the percentage of time spent in each function in relation to the whole. What's interesting to note from this column is that the bubble-sort algorithm requires 2.5 times as much execution time to sort the identical list as the insert-sort. The next column, `cumulative seconds`, is a running sum of the number of seconds, and the next, `self seconds`, is the number of seconds taken by this function alone. Note that the table itself is sorted by this column in descending order. The column entitled `calls` represents the total number of times that this function was called, if the function itself was profiled; otherwise the element is blank. The `self s/call` represents the average number of seconds spent in this function (including functions that it calls), while the `total s/call` represents the total number of seconds spent in the function (including functions that it calls). Finally, the name of the function is provided.

The next element provided by gprof is the call graph. This summary (shown in Listing 33.3) shows each function and the calls that are made by it, including the time spent within each function. It illustrates the timing as a hierarchy, which is useful in understanding timing for individual functions down the chain and their effect on higher layers of the call chain.

LISTING 33.3 Sample Call Graph Output from `gprof`

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    2.11                 main [1]
                1.44    0.00       1/1           bubbleSort [2]
                0.67    0.00       1/1           insertSort [3]
                0.00    0.00       2/2           init_list [4]
-----------------------------------------------
                1.44    0.00       1/1           main [1]
[2]     68.2    1.44    0.00       1         bubbleSort [2]
-----------------------------------------------
                0.67    0.00       1/1           main [1]
[3]     31.8    0.67    0.00       1         insertSort [3]
-----------------------------------------------
                0.00    0.00       2/2           main [1]
[4]      0.0    0.00    0.00       2         init_list [4]
-----------------------------------------------
```

The `index` column is a unique number given to each element. The column marked `% time` represents the percentage of the total amount of time that is spent in the given function and its children calls. The `self` column is the amount of time spent in the function, with `children` as the amount of time spent in the children's functions. Note that in the first row, `children` is 2.11 (a sum of the two `sort` functions 1.44 + 0.67). This illustrates that the children functions took all of the time, and no meaningful time was spent in the `main` function itself. The `called` field identifies how many times the function was called by the parent. The first number is the number of times the particular parent called the function, and the second number is the total number of times that the child function was called altogether. Finally, the `name` column represents the names of the functions. Note that in the first row (after the row headings), the name `<spontaneous>` is used. This simply means that the parent could not be determined (very likely the C-start initialization).

Now that you have a performance baseline of your application, you can rebuild your application and can see how you can improve it. In this example, you build using `-O2` optimization to see how well it improves this very simple application:

```
$ gcc -o sort sort.c -pg -O2
$ ./sort
$ gprof sort gmon.out | more
```

From Listing 33.4, you see that the performance of the application improves significantly (a five times improvement for the insert-sort) from Listing 33.2

(pre-optimization). The function bubbleSort sees only a modest four times improvement.

LISTING 33.4 Sample Flat Profile Output from gprof for the Optimized Application

```
Flat profile:
Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
76.47      0.78     0.78        1   780.00   780.00  bubbleSort
23.53      1.02     0.24        1   240.00   240.00  insertSort
0.00       1.02     0.00        2     0.00     0.00  init_list
```

This clearly illustrates the usefulness of the gprof utility (and the gcc optimizer). You can also see how the -O2 optimization level improves the source.

Options Available for gprof

Now that you have covered the basic uses of gprof, you can look at the variety of options that are provided. While gprof provides a large number of options, this section discusses some of the more useful ones here. For a complete list of options, see the gprof help.

Source Annotation

The gprof utility can be used to annotate the source with frequency of execution. However, the image must be built for this purpose. The -g and -a options must be specified along with -pg as:

```
gcc -o sort sort.c -g -pg
```

This results in an image with not only the profile information (via the -pg option), but also debugging information (through the -g option). Upon executing the image and checking the resulting output, as follows:

```
gprof -A -l sort gmon.out
```

you get Listing 33.5. The -A option tells gprof to emit an annotated source listing. The -l option specifies to emit a function execution profile, as shown in the following:

Listing 33.5 Sample Source Annotation from `gprof` for the Sort Application (Incomplete)

```
                #include <stdio.h>
                #define MAX_ELEMENTS 10000
                void insertSort( int list[], int size )
            1 -> {
                  int i, j, temp;
                  for ( i = 1 ; i <= size-1 ; i++ ) {
                    temp = list[i];
                    for ( j = i-1 ; j >= 0 && (list[j] > temp) ; j– ) {
                      list[j+1] = list[j];
                    }
                    list[j+1] = temp;
                  }
                }
```

The `-x` option can also be used with `gprof` to extend execution counts to all source lines.

Ignoring Private Functions

For private functions that are statically declared, you can ignore the statistics for these through the use of the `-a` option (or `-no-static`). The time spent in the private function is attributed to the caller, with the private function never appearing in the flat profile or the call graph.

Recommending Function Ordering

The `gprof` utility can recommend a function ordering that can improve cache and translation lookaside buffer (TLB) performance on systems that require functions to be contiguous in memory. For systems that include multiway caches, this might not provide much improvement.

To recommend a function ordering (via `–function-ordering`), the following command sequences can be used:

```
$ gcc -o sort sort.c -pg -g
$ ./sort
$ gprof sort gmon.out –function-ordering
```

A list of functions is then suggested with specific ordering (using the source from Listing 33.1). The result is shown in Listing 33.6.

LISTING 33.6 Sample Function Ordering from `gprof`

```
        insertSort
        bubbleSort
        init_list
        _init
        _start
        __gmon_start__
    . . .
```

Note the grouping of the sorting and `init` functions, which are all called in proximity of one another. In some cases, reordering functions can be done as simply as coexisting contiguous functions within a single source file. The linker can also provide this capability.

The file ordering option (`–file-ordering`) provides a similar capability by recommending the order in which objects should be linked to the target image.

Minimizing `gprof` Summary

The `-b` option is useful to minimize the amount of superfluous description data that is emitted. Using `-b` removes the field discussion in the output, as shown in Listing 33.7.

LISTING 33.7 Sample Brief Output from `gprof`

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 71.59      3.10     3.10        1     3.10     3.10  bubbleSort
 28.41      4.33     1.23        1     1.23     1.23  insertSort
  0.00      4.33     0.00        2     0.00     0.00  init_list
            Call graph
granularity: each sample hit covers 4 byte(s) for 0.23% of 4.33
seconds
index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0    0.00    4.33                 main [1]
                3.10    0.00       1/1           bubbleSort [2]
                1.23    0.00       1/1           insertSort [3]
                0.00    0.00       2/2           init_list [4]
_____________________-
                3.10    0.00       1/1           main [1]
```

```
[2]     71.6    3.10    0.00        1          bubbleSort [2]
-----------------------------------------------
                1.23    0.00        1/1            main [1]
[3]     28.4    1.23    0.00        1          insertSort [3]
-----------------------------------------------
                0.00    0.00        2/2            main [1]
[4]      0.0    0.00    0.00        2          init_list [4]
-----------------------------------------------
Index by function name
 [2] bubbleSort (sort.c)  [4] init_list (sort.c)  [3] insertSort
 (sort.c)
```

To further minimize the output, `–no-flat-profile` can be used if the flat profile is not needed (the default is `–flat-profile`). The call graph can be disabled using the `–no-graph` option (default is `–graph`).

Finding Unused Functions

The `gprof` utility can identify functions that are not called in a given run. The `–display-unused-functions` is used in conjunction with the `–static-call-graph` option to list those functions, as follows:

```
gprof sort gmon.out –display-unused-functions –static-call-graph
```

Increasing `gprof` Accuracy

In some cases, a program might differ in timing or might represent such a small timing sample that its accuracy is left in question. To increase the accuracy of an application's profiling, the application can be performed numerous times and then averaged. A sample bash script that provides this capability is shown in Listing 33.8 [GNU`gprof`].

LISTING 33.8 Analyzing Multiple Invocations of an Application (on the CD-ROM at `./source/ch33/script`)

```
#!/bin/bash
for i in 'seq 1 5'; do
  ./sort
  mv gmon.out gmon.out.$i
done
gprof –sum sort gmon.out.*
gprof -b –no-graph sort gmon.sum
```

Running this script results in a single flat profile over five invocations of the sort application. This uses the `-sum` option of `gprof` to summarize the collection of input `gmon.out` files into a single summary file, `gmon.sum`. The per/call measurements of the flat profile can then be used as higher accuracy function timings.

Considerations

Profiling with `gprof` is a sampling process that is subject to statistical inaccuracies. Recall from Listing 33.7 that each sample counts as 0.01 seconds. This is the sampling period. The closer the sampling time is to this period, the larger the error is for the profile. For this reason, increasing the accuracy of the profile is recommended (see the previous section, "Increasing `gprof` Accuracy"). When `gprof` is enabled, it does introduce extra code into the image that can also affect its behavior and performance. Therefore, simply by measuring the performance of the code, you can affect it. This should always be kept in mind when you are using performance and coverage tools.

Summary

This chapter discussed profiling with GNU's `gprof`, identifying some of the most useful options that are provided. You walked through building an application for use with `gprof` and then gathering a profile from it, including options for the various `gprof` data products and recommendations for improving the performance of an application from a caching perspective.

References

[GNU`gprof`] The GNU Profiler, Jay Fenlason and Richard Stallman at http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_mono/gprof.html.

This page intentionally left blank

34 Advanced Debugging Topics

In This Chapter

- Memory Debugging with Open Source Tools
- Cross-Referencing Tools
- Tracing Tools
- Other Techniques

Introduction

This final chapter looks at some of the advanced techniques for debugging GNU/Linux applications and also shows you how to tune them. It looks at a variety of open source tools for memory debugging and cross-referencing, as well as other tools and techniques.

Memory Debugging

You can start this chapter with one of the most difficult to debug issues, which would be dynamic memory issues. These can take the form of invalid use of a freed buffer, buffer overruns, or leaks. Overruns tend to result in insidious bugs where the resulting behavior might not be deterministic and can change to a different behavior in a new run or after a compile. Leaks tend to result in different behaviors such as program freezes or severe reductions in performance.

This section reviews a number of open source solutions that can help you find memory issues in your programs.

VALGRIND

Valgrind is a suite of tools for debugging and profiling. One of the most well known tools in this suite is called memwatch; it can help find memory issues (such as memory leaks). Valgrind also includes other tools such as cachegrind, to annotate your program based upon its cache utilization, and massif, which is a heap profiler.

The easiest way to get the Valgrind suite is through the `apt` command:

```
$ sudo apt-get install valgrind
```

After installation, you have a binary called `valgrind` installed that contains the suite of debugging tools.

Out-of-bound memory errors are not only common, but tend to create problems that aren't easily traceable back to the out-of-bound error. When a write occurs out of bounds, another object is corrupted, so the impact can be unrelated.

Now it's time to take a look at a couple of uses of Valgrind. One of the most important uses of Valgrind is memcheck. This tool specifically monitors memory usage and allocation within a running application. You start with the sample program shown in Listing 34.1. This program has two specific errors. You can then use memcheck to find them.

LISTING 34.1 Sample Program with Memory Errors (`test.c`)

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:  #include <string.h>
 4:
 5:         int main()
 6: {
 7:   char *buffer;
 8:
 9:   buffer = (char *)malloc(5);
10:
11:   strcpy( buffer, "0123456" );
12:
13:   printf( "buffer is %s\n", buffer );
14:
15:   return 0;
16: }
```

To use memcheck, you first need to prepare your program. You compile the program normally, but with the `-g` option, which inserts debugging information into the resulting executable. Recall that the `-g` option is required to use GNU Debugger (GDB).

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

If you execute this program, it appears to operate normally (the errors are not immediately destructive). Now you can use Valgrind to find the errors. You use Valgrind as an application and your test program as an argument, as shown in Listing 34.2. As shown, the utility finds the buffer overrun at line 11 (in `strcpy`). Valgrind also finds the memory leak (reported at the end of the listing) indicated by an allocation but not an accompanying free.

LISTING 34.2 Sample Output from the Valgrind Utility

```
$ valgrind –tool=memcheck ./test
==7221== Memcheck, a memory error detector.
==7221== Copyright (C) 2002-2005, and GNU GPL'd, by Julian Seward et
al.
==7221== Using LibVEX rev 1471, a library for dynamic binary translation.
==7221== Copyright (C) 2004-2005, and GNU GPL'd, by OpenWorks LLP.
==7221== Using valgrind-3.1.0-Debian, a dynamic binary instrumentation
         framework.
==7221== Copyright (C) 2000-2005, and GNU GPL'd, by Julian Seward et al.
==7221== For more details, rerun with: -v
==7221==
==7221== Invalid write of size 4
==7221==    at 0x80483CC: main (test.c:11)
==7221==  Address 0x415502C is 4 bytes inside a block of size 5 alloc'd
==7221==    at 0x401B422: malloc (vg_replace_malloc.c:149)
==7221==    by 0x80483BF: main (test.c:9)
==7221==
...
buffer is 0123456
==7221==
==7221== ERROR SUMMARY: 8 errors from 5 contexts (suppressed: 11 from 1)
==7221== malloc/free: in use at exit: 5 bytes in 1 blocks.
==7221== malloc/free: 1 allocs, 0 frees, 5 bytes allocated.
==7221== For counts of detected errors, rerun with: -v
==7221== searching for pointers to 1 not-freed blocks.
```

```
==7221== checked 75,428 bytes.
==7221==
==7221== LEAK SUMMARY:
==7221==    definitely lost: 5 bytes in 1 blocks.
==7221==      possibly lost: 0 bytes in 0 blocks.
==7221==    still reachable: 0 bytes in 0 blocks.
==7221==         suppressed: 0 bytes in 0 blocks.
==7221== Use –leak-check=full to see details of leaked memory.
$
```

You can also use Valgrind to profile an application's cache usage. To do this you use the `cachegrind` option within Valgrind (which is shown in Listing 34.3). As you can see in the listing, the utility details both instruction and data cache usage for the Level 1 and Level 2 caches. This is extremely useful for profiling cache usage to improve it for greater performance.

LISTING 34.3 Valgrind's `cachegrind` Output

```
$ valgrind –tool=cachegrind ./test
==7331== Cachegrind, an I1/D1/L2 cache profiler.
==7331== Copyright (C) 2002-2005, and GNU GPL'd, by Nicholas Nethercote
         et al.
==7331== Using LibVEX rev 1471, a library for dynamic binary translation.
==7331== Copyright (C) 2004-2005, and GNU GPL'd, by OpenWorks LLP.
==7331== Using valgrind-3.1.0-Debian, a dynamic binary instrumentation
         framework.
==7331== Copyright (C) 2000-2005, and GNU GPL'd, by Julian Seward et al.
==7331== For more details, rerun with: -v
==7331==
buffer is 0123456
==7331==
==7331== I   refs:      168,074
==7331== I1  misses:      1,247
==7331== L2i misses:      1,187
==7331== I1  miss rate:    0.74%
==7331== L2i miss rate:    0.70%
==7331==
==7331== D   refs:       77,981  (58,028 rd + 19,953 wr)
==7331== D1  misses:      3,356  ( 2,971 rd +    385 wr)
==7331== L2d misses:      2,704  ( 2,383 rd +    321 wr)
==7331== D1  miss rate:     4.3% (   5.1%   +    1.9%  )
==7331== L2d miss rate:     3.4% (   4.1%   +    1.6%  )
==7331==
```

```
==7331== L2 refs:         4,603  ( 4,218 rd +    385 wr)
==7331== L2 misses:       3,891  ( 3,570 rd +    321 wr)
==7331== L2 miss rate:      1.5% (   1.5%   +    1.6%  )
$
```

When you are using Valgrind, your application slows by 20 or 30 times (as it's a CPU emulator), but that's a small price to pay for the benefits that it provides.

Electric Fence

Electric Fence is another memory debugger with the novel feature that it can identify the precise function of a malloc'd buffer where an overrun or underrun occurs. Electric Fence is able to do this because it uses the virtual memory subsystem to place zones around malloc'd buffers from which hardware can detect overrun/underrun conditions.

First, to get the latest Electric Fence, use `apt` with the following command line:

```
$ sudo apt-get install electric-fence
```

You can build Electric Fence into your application by linking in the library as follows:

```
$ gcc -o test test.c -lefence
```

After it is compiled, run your application under GDB, and Electric Fence indicates any overruns that occur. Electric Fence causes your program to run slower with more memory. Electric Fence is also an older tool, and you have better alternatives to choose (such as Valgrind).

yamd

The `yamd` (yet another malloc debugger) utility is another useful package for detecting memory-related bugs in both C and C++. The `yamd` package works, like most, by emulating the malloc and free calls with specially instrumented versions.

You can download the `yamd` package using the command line that follows. It can be built and installed using the package installation instructions explored in Chapter 28, "GNU/Linux Administration Basics."

```
$ wget http://www.cs.hmc.edu/~nate/yamd/yamd-0.32.tar.gz
```

After it is installed, you can use the utility to build an image from your source to be tested. It's time to use another test program now that exhibits erroneous freeing behaviors. For this example, you use the simple program shown in Listing 34.4.

This application exhibits two errors. The first is an attempt to free a buffer back to the heap twice (line 11), and the second is an attempt to release an erroneous buffer (line 12).

LISTING 34.4 Sample Program with Erroneous Freeing Behaviors

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:
 4:  int main()
 5: {
 6:   char *buffer;
 7:
 8:   buffer = (char *)malloc(50);
 9:
10:   free( buffer );
11:   free( buffer );
12:   free( ++buffer );
13:
14:   return 0;
15: }
```

To prepare your program with `yamd`, you use the provided utilities to build the program as follows:

```
$ yamd-gcc test.c
```

for your sample C program. If your application is written in C++, you can use `yamd-g++`. The result of this is an executable called `a.out`. You can execute this directly, as shown in Listing 34.5, which includes the output generated by `yamd`. For system calls, you see a variety of output indicating the traceback. The `INFO` blocks indicate normal operations (such as the malloc and first successful free). The `ERROR` blocks indicate abnormal operations, which include the multiple freeing of the buffer and the attempt to free an erroneous buffer. Note that some of the traceback information has been removed to shrink the output.

LISTING 34.5 Execution of the `yamd` Instrumented Image

```
$ ./a.out
YAMD version 0.32
Starting run: ./a.out
Executable: /home/mtj/memcheck/a.out
Virtual program size is 1580 K
Time is Sun Feb  3 00:21:01 2008
```

```
default_alignment = 1
min_log_level = 1
repair_corrupted = 0
die_on_corrupted = 1
check_front = 0

INFO: Normal allocation of this block
Address 0xb7ef8fce, size 50
Allocated by malloc at
BEGIN TRACEBACK
/lib/tls/i686/cmov/libc.so.6(malloc+0x35)[0xb7e2b3c5]
./a.out[0x8048b3c]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xd2)[0xb7ddaea2]
./a.out[0x8048a81]
END TRACEBACK

INFO: Normal deallocation of this block
Address 0xb7ef8fce, size 50
Allocated by malloc at
BEGIN TRACEBACK
/lib/tls/i686/cmov/libc.so.6(malloc+0x35)[0xb7e2b3c5]
./a.out[0x8048b3c]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xd2)[0xb7ddaea2]
./a.out[0x8048a81]
END TRACEBACK
Freed by free at
BEGIN TRACEBACK
/lib/tls/i686/cmov/libc.so.6(__libc_free+0x35)[0xb7e292f5]
./a.out[0x8048b4a]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xd2)[0xb7ddaea2]
./a.out[0x8048a81]
END TRACEBACK

ERROR: Multiple freeing
free of pointer already freed
Address 0xb7ef8fce, size 50
Allocated by malloc
Freed by free

ERROR: Free of errnoneous pointer
Freeing erroneous pointer 0xb7ef8fcf
Seems to be part of this block:
Address 0xb7ef8fce, size 50
Allocated by malloc
```

```
Freed by free
Address in question is at offset 1 (in bounds)

*** Finished at Sun Feb  3 00:21:01 2008
Allocated a grand total of 50 bytes
1 allocations
Average of 50 bytes per allocation
Max bytes allocated at one time: 50
12 K alloced internally / 8 K mapped now / 4 K max
Virtual program size is 1596 K
End.
$
```

As can be see in Listing 34.5, `yamd` generates a considerable amount of output. In addition to the runtime error checking, it also provides a summary of memory manipulations when the program exits.

mtrace

Finally, you now take a look at one of the simpler methods for detecting memory leaks in C or C++ programs. The `mtrace` utility can parse a trace file that is collected during the execution of an instrumented program. To instrument your program, you simply add the `mtrace()` function to log malloc and free calls, as shown in Listing 34.6. Line 9 contains the `mtrace` call.

Listing 34.6 Instrumenting Your Program with `mtrace` (`test.c`)

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:  #include <mcheck.h>
 4:
 5:         int main()
 6: {
 7:   char *buffer;
 8:
 9:   mtrace();
10:
11:   buffer = (char *)malloc(50);
12:
13:   return 0;
14: }
```

You can now compile your application (including debug information) as follows:

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

Next, you set an environment variable that defines where the trace data should be emitted (`MALLOC_TRACE`):

```
$ export MALLOC_TRACE=/home/mtj/memcheck/mtrace.txt
```

After executing the test program, the malloc/free tracing is written to your trace file defined by `MALLOC_TRACE`. You can then use the `mtrace` utility to parse the results as follows:

```
$ mtrace test $MALLOC_TRACE

Memory not freed:
-----------------
   Address     Size     Caller
0x0804a378     0x32  at /home/mtj/memcheck/test.c:11
$
```

What's shown is an allocation of a buffer at line 11 that has no accompanying free (a memory leak).

The `mtrace` approach to memory leak detection is part of the GNU C library and, therefore, is probably already on your system.

Cross-Referencing Tools

Modern integrated development environments (IDEs) include the capability to create a cross-reference of the source tree to enable simpler navigation. GNU/Linux supports a number of options that provide cross-referencing capabilities in the context of traditional GNU/Linux editing environments (such as vi). The next sections explore two options provided by GNU/Linux.

Cscope

Cscope is a great utility for browsing source code and integrates with GNU/Linux editors. Cscope is much more than a `grep` utility with results formatting, as it can parse the source to find the context of the object that you're looking for. For example, you can find any C symbol (variable name, function name, and so on),

global definitions, functions that are called by another function, functions calling a defined function, and simple text string and other search possibilities. Listing 34.7 shows a simple example of searching for a C symbol (in this case, `ruby_debug`) within the Ruby language source distribution. After providing the variable name (entered at the bottom of the listing), Cscope provides the references at the top. Each of these references can be selected, which takes the user to the particular source file in a given editor (default vi).

LISTING 34.7 Sample Output of the Cscope Utility

```
C symbol: ruby_debug

  File      Function          Line
0 ruby.c    <global>             52 VALUE ruby_debug = Qfalse;
1 eval.c    rb_longjmp         4576 if (RTEST(ruby_debug) &&
                                    !NIL_P(ruby_errinfo)
2 eval.c    rb_thread_start_0 12018 else if (th->safe < 4 &&
            (ruby_thread_abort
                                    || th->abort || RTEST(ruby_debug)))
{
3 ruby.c    proc_options         516 ruby_debug = Qtrue;
4 ruby.c    proc_options         709 ruby_debug = Qtrue;
5 ruby.c    ruby_prog_init      1161 rb_define_variable("$DEBUG",
                                    &ruby_debug);
6 ruby.c    ruby_prog_init      1162 rb_define_variable("$-d",
                                    &ruby_debug);
7 ruby.h    VALUE                552 RUBY_EXTERN VALUE ruby_verbose,
                                    ruby_debug;
8 sprintf.c rb_f_sprintf         801 if (RTEST(ruby_debug))
                                    rb_raise(rb_eArgError, mesg);


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

You can exit out of the utility by simply pressing Ctrl+D.

Other Cross-Referencing Tools

The `cxref` utility is a useful one for generating cross-references of a program. This utility can produce HTML listings of one or more source files, including all functions and symbols and their relationships (where defined and used).

Another cross-reference tool that provides a simple but easy to read function call hierarchy is called `cflow`. The output of `cflow` shows the functions defined in a file (and their source line), along with the functions that are called from each function. A sample output of `cflow` is shown in Listing 34.8.

The `cflow` output can be read very easily. The functions that are left indented (such as `ruby_getcwd`) are those functions present in the file (`util.c`) along with their source line. Those functions indented to the right in the following listing are those functions that are called (such as `ruby_xmalloc()`).

LISTING 34.8 Sample Output from the `cflow` Utility

```
$ cflow util.c
1       ruby_getcwd {util.c 644}
2               ruby_xmalloc {}
3               getcwd {}
4               __errno_location {}
5               free {}
6               rb_sys_fail {}
7               ruby_xrealloc {}
8       ruby_qsort {util.c 483}
9               mmswap_ {}
10              mmrot3_ {util.c 437}
11      ruby_scan_hex {util.c 52}
12              strchr {}
13      ruby_strdup {util.c 632}
14              strlen {}
15              ruby_xmalloc {}
16              memcpy {}
17      ruby_strtod {util.c 740}
18              __errno_location {}
19              __ctype_b_loc {}
20              __builtin_huge_val {}
$
```

System Call Tracing with `ltrace`

In Chapter 31, "Code Hardening," you got a first look at system call tracing using `strace`. But what if your interest lies in library calls made by a given executable?

That's where `ltrace` comes in. The `ltrace` utility emits a symbolic trace for an executable for a specific dynamic library. The `ltrace` utility can also be used just like `strace`, but this section focuses on its library trace capabilities.

Now it's time to take a look at a couple of examples that illustrate the capabilities of `ltrace`. First, say for a given command (in this case `ls`) you want to see what library calls are made to the standard C library (`libc`). Recall that you can understand which libraries an image is dependent upon using the `ldd` command. In this example, you specify the library of interest with the `-l` option and then provide your image to trace at the end (see Listing 34.9). As the image is run, the library calls are emitted (on the left) with the return status of each shown on the right (after the `=`).

This listing is informative because you can see the operation of the `ls` utility. The `ls` command attempts to read in a number of environment variables, none of which are found (each returns `NULL`). Note near the end of this partial listing the call to `getopt_long` (which is used to retrieve command-line options). As shown, this library call returns `-1`, indicating that all options have been parsed (which in this case are none). Also of interest is the third parameter to `getopt_long`. This is the list of command-line options that are supported to `ls` (where `w` and `I` require a parameter). This is sometimes useful to find options that are hidden (not listed in the help, but available in the image).

LISTING 34.9 Tracing Calls to `libc` with `ltrace`

```
$ ltrace -l /lib/libc.so.6 /bin/ls
__libc_start_main(0x804e5c5, 1, 0xbffee604, 0x80563a8, <unfinished ...>
setlocale(6, "")                                = "en_US.UTF-8"
bindtextdomain("coreutils", "/usr/share/locale") = "/usr/share/locale"
textdomain("coreutils")                         = "coreutils"
__cxa_atexit(0x804fcbd, 0, 0, 0xb7fabadc, 0xbffee578) = 0
isatty(1)                                       = 1
getenv("QUOTING_STYLE")                         = NULL
getenv("LS_BLOCK_SIZE")                         = NULL
getenv("BLOCK_SIZE")                            = NULL
getenv("BLOCKSIZE")                             = NULL
getenv("POSIXLY_CORRECT")                       = NULL
getenv("BLOCK_SIZE")                            = NULL
getenv("COLUMNS")                               = NULL
ioctl(1, 21523, 0xbffee130)                     = 0
getenv("TABSIZE")                               = NULL
getopt_long(1, 0xbffee604, "abcdfghiklmnopqrstuvw:xABCDFGHI:"... ) = -1
...
```

```
exit(0 <unfinished ...>
__fpending(0xb7fac0e0, 0xb7e6e8e8, 1, 1, 0)      = 0
fclose(0xb7fac0e0)                               = 0
+++ exited (status 0) +++
$
```

Another useful option of `ltrace` is `-r`, which emits the relative timestamp of each library call (see Listing 34.10). Also shown in this listing is the indent option (`-n`), which indents *n* spaces for each nested system call.

LISTING 34.10 Nested System Call Tracing with Relative Timestamps

```
$ ltrace -r -n 5 -l /lib/libc.so.6 /bin/ls
...
  0.002015      fwrite_unlocked("Test.pm", 1, 7, 0xb7eda0e0) = 7
  0.001010      __overflow(0xb7eda0e0, 10, 0, 0, 0test  test.pl  Test.pm
) = 10
  0.001143      free(0x80763f0)                  = <void>
  0.001307      free(NULL)                       = <void>
  0.001258      free(0x80763d8)                  = <void>
  0.000834      exit(0 <unfinished ...>
  0.000513           __fpending(0xb7eda0e0, 0xb7d9c8e8, 1, 1, 0) = 0
  0.001314           fclose(0xb7eda0e0)          = 0
  0.002497 +++ exited (status 0) +++

$
```

Instead of tracing information, you can explore the numbers of system calls made by a program and how much time is spent in each. This is achieved using the `-c` option (for count). This is shown in Listing 34.11.

LISTING 34.11 System Call Counts with `ltrace`

```
$ ltrace -c -l /lib/libc.so.6 /bin/ls
test  test.pl  Test.pm
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
 68.15    0.019121       19121         1 setlocale
  4.11    0.001153          64        18 __ctype_get_mb_cur_max
  3.95    0.001109          79        14 readdir64
  3.46    0.000972         972         1 qsort
  3.45    0.000967          64        15 __errno_location
  2.87    0.000804         160         5 __overflow
```

```
 2.26    0.000633          70         9 malloc
 1.99    0.000557          69         8 getenv
 1.38    0.000388         129         3 fwrite_unlocked
 1.05    0.000296         296         1 opendir
 0.94    0.000265          66         4 memcpy
 0.84    0.000237          79         3 strcoll
 0.82    0.000230         230         1 fclose
 0.78    0.000220          73         3 free
 0.58    0.000162         162         1 isatty
 0.55    0.000153         153         1 closedir
 0.50    0.000140         140         1 ioctl
 0.40    0.000113         113         1 bindtextdomain
 0.39    0.000110         110         1 realloc
 0.35    0.000098          98         1 getopt_long
 0.31    0.000087          87         1 __cxa_atexit
 0.30    0.000083          83         1 __fpending
 0.29    0.000082          82         1 textdomain
 0.28    0.000079          79         1 _setjmp
------ ----------- ----------- --------- ----------
100.00    0.028059                    96 total
$
```

Other useful options for `ltrace` include tracing child processes created by the image (`-f`), emitting the instruction pointer at call time (`-i`), showing the time spent inside a system call (`-T`), and increasing the debug level (`-d`).

Dynamic Attachment with GDB

A very interesting feature of GDB is its ability to attach to an already running process. You can start with the simple program shown in Listing 34.12 as the program to which you want to attach.

LISTING 34.12 Attaching to a Running Process with GDB

```
1:  #include <stdio.h>
2:  #include <unistd.h>
3:
4:  int counter = 0;
5:
6:  int main()
7:  {
```

```
 8:    while(1) {
 9:
10:      sleep(1);
11:
12:      counter++;
13:
14:    }
15:
16:    return 0;
17:  }
```

Listing 34.13 shows your sample session with GDB. You begin by compiling the sample program (note the use of `-g`, which is required here to use symbol information). With an executable image created, you start the image in the background. You then use GDB, specifying the process ID (PID) to which you want to attach. GDB attaches to the process (using a signal to stop it, but keep its memory intact) and then loads the symbols from the various libraries to which the image was compiled. At this point, you are presented with the `gdb` prompt and can look at the source, inspect variables, and step through the source. When you are done, you use the `detach` command to detach from the process and allow it to continue running.

LISTING 34.13 Attaching to a Running Process with GDB

```
$ gcc -g -o proc proc.c
$ ./proc &
[1] 16999
$ gdb -q - 16999
Attaching to process 16999
Reading symbols from /home/mtj/memcheck/proc...done.
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Reading symbols from /lib/tls/i686/cmov/libc.so.6...done.
Loaded symbols for /lib/tls/i686/cmov/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0xffffe410 in __kernel_vsyscall ()
(gdb) list
1       #include <stdio.h>
2       #include <unistd.h>
3
4       int counter = 0;
5
6       int main()
7       {
```

```
8         while(1) {
9
10          sleep(1);
(gdb) p counter
$1 = 9
(gdb) step
Single stepping until exit from function __kernel_vsyscall,
which has no line number information.
0xb7e6c110 in nanosleep () from /lib/tls/i686/cmov/libc.so.6
(gdb) step
Single stepping until exit from function nanosleep,
which has no line number information.
0xb7e6bf3c in sleep () from /lib/tls/i686/cmov/libc.so.6
(gdb) step
Single stepping until exit from function sleep,
which has no line number information.
main () at proc.c:12
12          counter++;
(gdb) step
14        }
(gdb) p counter
$2 = 10
(gdb) detach
Detaching from program: /home/mtj/memcheck/proc, process 16999
(gdb) quit
You have new mail in /var/mail/mtj
mtj@camus:~/memcheck$
```

You must have adequate permissions to debug processes, and this is mostly useful only if the symbols are retained in the image (through the `-g` option during compile). But if your process is stuck or is dominating CPU usage, attaching through GDB is a great way to find out why.

Summary

GNU/Linux supports a wide range of advanced debugging options, including a variety of solutions to support memory debugging. This chapter has presented a number of memory debugging options that can help identify memory leaks, buffer overruns, and buffer under-runs. Also explored in this chapter were cross-referencing tools and tools to assist in system-level tracing.

Resources

ccmalloc at http://www.inf.ethz.ch/personal/biere/projects/ccmalloc/.
Cscope at http://cscope.sourceforge.net/.
cxref (Cross-Reference) at http://www.gedanken.demon.co.uk/cxref/.
Electric Fence at http://perens.com/FreeSoftware/.
ltrace at http://linux.die.net/man/1/ltrace.
mtrace at http://en.wikipedia.org/wiki/Mtrace.
Valgrind Instrumentation Framework at http://www.valgrind.org.

This page intentionally left blank

Appendix

A Acronyms and Partial Acronyms

AMD	Advanced Micro Devices
API	Application Programmer's Interface
APT	Advanced Package Tool
ASCII	American Standard Code for Information Interchange
AT&T	American Telephone and Telegraph
AWK	Aho-Weinberger-Kernighan
BASH	Bourne-Again SHell
BB	Basic Block
BBG	Basic Block Graph
BIOS	Basic Input Output System
BNF	Backus-Naur Form
BSD	Berkeley Software Distribution
BSS	Block Started by Symbol
CMU	Carnegie Mellon University
COW	Copy-On-Write
CPU	Central Processing Unit
CSE	Common Sub-expression Elimination
CVS	Concurrent Versions System
DEC	Digital Equipment Corporation
DMA	Direct Memory Access
DNS	Domain Name Server
DL	Dynamically Loaded
DWARF	Debugging with Attribute Record Format
EMACS	Editing MACroS
ELF	Executable and Linking Format
EOF	End of File
EXT2	Second Extended Filesystem
EXT3	Third Extended Filesystem
FHS	Filesystem Hierarchy Standard

FIFO	First-In First-Out
FQDN	Fully Qualified Domain Name
FS	Field Separator
FSF	Free Software Foundation
GCC	GNU Compiler Collection
GCOV	GNU Coverage
GDB	GNU DeBugger
GID	Group ID
GLIBC	GNU C Library
GMT	Greenwich Mean Time
GNU	GNU's Not UNIX
GOT	Global Offset Table
GPROF	GNU Profiler
HTML	HyperText Markup Language
HTONL	Host TO Network Long
HTONS	Host TO Network Short
HUP	HangUP
IBM	International Business Machines
IP	Internet Protocol
IPC	Inter-Process Communication
IPO	Initial Public Offering
IPv4	Internet Protocol Version 4
ISS	Instruction Set Simulator
JVM	Java Virtual Machine
KB	KiloByte
KVM	Kernel Virtual Machine
LIFO	Last-In First-Out
LISP	List Processor
MAME	Multiple Arcade Machine Emulator
MINIX	Miniature UNIX
MIT	Massachusetts Institute of Technology
MMU	Memory Management Unit
MUTEX	Mutual Exclusion
NF	Number of Fields
NFS	Network File System
NIC	Network Interface Card
NPTL	Native POSIX Thread Library
NR	Number of Record
NTOHL	Network TO Host Long
NTOHS	Network TO Host Short
OFS	Output Field Separator

ORS	Output Record Separator
OSI	Open Source Initiative
PDP	Programmed Data Processor
PGRP	Process Group
PIC	Position Independent Code
PID	Process Identifier
POSIX	Portable Operating System Interface
PWD	Present Working Directory
QID	Queue Identifier
QPL	Qt Public License
RAID	Redundant Array of Inexpensive Disks
RAM	Random Access Memory
REGEX	Regular Expression
ROM	Read Only Memory
RS	(input) Record Separator
SCM	Source Control Management
SCSH	Scheme Shell
SED	Stream Editor
SPLINT	Secure Programming Lint
STDERR	Standard Error
STDIN	Standard Input
STDOUT	Standard Output
SVN	Subversion
SYSV	System 5
TAR	Tape Archive
TCL	Tool Command Language
TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
UDP	User Datagram Protocol
UID	User ID
UTC	Coordinated Universal Time
VFS	Virtual File System
VI	Visual Interface
VM	Virtual Machine
VMM	Virtual Machine Monitor
VPATH	Virtual Path
YAMD	Yet Another Malloc Debugger

This page intentionally left blank

Appendix B About the CD-ROM

The CD-ROM included with *GNU/Linux Application Programming, Second Edition* includes all sample applications found in the book.

CD-ROM Folders

- Source: Contains all the code from examples in the book, by chapter.
- Figures: Contains all the figures in the book, by chapter.

Overall System Requirements

- Linux with a 2.6 Kernel (tested with Fedora and Ubuntu)
- Pentium I processor or greater
- CD-ROM drive
- Hard drive
- 256 MB of RAM
- 1 MB of hard drive space for the code examples

This page intentionally left blank

Index

M

Q

T

License Agreement/Notice of Limited Warranty

By opening the sealed disc container in this book, you agree to the following terms and conditions. If, upon reading the following license agreement and notice of limited warranty, you cannot agree to the terms and conditions set forth, return the unused book with unopened disc to the place where you purchased it for a refund.

License:
The enclosed software is copyrighted by the copyright holder(s) indicated on the software disc. You are licensed to copy the software onto a single computer for use by a single user and to a backup disc. You may not reproduce, make copies, or distribute copies or rent or lease the software in whole or in part, except with written permission of the copyright holder(s). You may transfer the enclosed disc only together with this license, and only if you destroy all other copies of the software and the transferee agrees to the terms of the license. You may not decompile, reverse assemble, or reverse engineer the software.

Notice of Limited Warranty:
The enclosed disc is warranted by Course Technology to be free of physical defects in materials and workmanship for a period of sixty (60) days from end user's purchase of the book/disc combination. During the sixty-day term of the limited warranty, Course Technology will provide a replacement disc upon the return of a defective disc.

Limited Liability:
THE SOLE REMEDY FOR BREACH OF THIS LIMITED WARRANTY SHALL CONSIST ENTIRELY OF REPLACEMENT OF THE DEFECTIVE DISC. IN NO EVENT SHALL COURSE TECHNOLOGY OR THE AUTHOR BE LIABLE FOR ANY OTHER DAMAGES, INCLUDING LOSS OR CORRUPTION OF DATA, CHANGES IN THE FUNCTIONAL CHARACTERISTICS OF THE HARDWARE OR OPERATING SYSTEM, DELETERIOUS INTERACTION WITH OTHER SOFTWARE, OR ANY OTHER SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES THAT MAY ARISE, EVEN IF COURSE TECHNOLOGY AND/OR THE AUTHOR HAS PREVIOUSLY BEEN NOTIFIED THAT THE POSSIBILITY OF SUCH DAMAGES EXISTS.

Disclaimer of Warranties:
COURSE TECHNOLOGY AND THE AUTHOR SPECIFICALLY DISCLAIM ANY AND ALL OTHER WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY, SUITABILITY TO A PARTICULAR TASK OR PURPOSE, OR FREEDOM FROM ERRORS. SOME STATES DO NOT ALLOW FOR EXCLUSION OF IMPLIED WARRANTIES OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THESE LIMITATIONS MIGHT NOT APPLY TO YOU.

Other:
This Agreement is governed by the laws of the State of Massachusetts without regard to choice of law principles. The United Convention of Contracts for the International Sale of Goods is specifically disclaimed. This Agreement constitutes the entire agreement between you and Course Technology regarding use of the software.